Alternative Split Functions and Dekker's Product

Stef Graillat, Vincent Lefèvre, Jean-Michel Muller

▶ **To cite this version:**

Stef Graillat, Vincent Lefèvre, Jean-Michel Muller. Alternative Split Functions and Dekker's Product. ARITH-2020 - IEEE 27th Symposium on Computer Arithmetic, Jun 2020, Portland, United States. pp.1-7, 10.1109/ARITH48897.2020.00015 . hal-02470782v2

HAL Id: hal-02470782
https://hal.science/hal-02470782v2

Submitted on 14 May 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Alternative Split Functions and Dekker's Product

Stef Graillat
Sorbonne Université
CNRS, LIP6
F-75005 Paris, France
stef.graillat@sorbonne-universite.fr

Vincent Lefèvre
Univ Lyon, Inria
ENS de Lyon, CNRS
Univ. Claude Bernard Lyon 1, LIP
F-69007 Lyon, France
vincent.lefevre@ens-lyon.fr

Jean-Michel Muller
Univ Lyon, CNRS
ENS de Lyon, Inria
Univ. Claude Bernard Lyon 1, LIP
F-69007 Lyon, France
jean-michel.muller@ens-lyon.fr

***Abstract*—We introduce algorithms for splitting a positive binary floating-point number into two numbers of around half the system precision, using arithmetic operations all rounded either toward $-\infty$ or toward $+\infty$. We use these algorithms to compute "exact" products (i.e., to express the product of two floating-point numbers as the unevaluated sum of two floating-point numbers, the rounded product and an error term). This is similar to the classical Dekker product, adapted here to directed roundings.**

***Index Terms*—Floating-point arithmetic, split functions, accurate products.**

I. Introduction

It is sometimes useful to express the exact product of two floating-point numbers a and b as an unevaluated sum $r_1 + r_2$ of two floating-point numbers. This for instance makes it possible to implement a "double word" (sometimes called "double-double") arithmetic, i.e., to mimic an arithmetic that has roughly twice the precision of the underlying floating-point arithmetic [1], [2], [3]. This also is a basic building block of accurate algorithms for computing dot products of vectors (see for instance [4]) and evaluating polynomials [5].

If a fused multiply-add (FMA) instruction is available, obtaining r_1 and r_2 from a and b is very easily done [6], [7, Section 4.4]: just compute first the floating-point (hence, rounded) product r_1 of a and b. One can show that if no underflow or overflow occurred when computing r_1 and the sum of the exponents of a and b is greater than or equal to the minimum exponent plus the precision minus 1, the number $r_2 = ab - r_1$ is a floating-point number, therefore it is exactly computed with one FMA operation.

If no FMA instruction is available, one needs to "split" the input operands into parts small enough, so that the product of two such parts fit exactly in a floating-point number. Two classical algorithms for doing that are Veltkamp's splitting [8], [9] and Dekker's product [10]. They require rounded-to-nearest operations (for a recent presentation of splitting algorithms with rounded to nearest operations, see [11]). In case of directed roundings, those algorithms are no more valid.

The purpose of this paper is to show that calculations similar to Veltkamp's splitting and Dekker's product can be done with directed roundings (more precisely, with operations either rounded toward $+\infty$ or rounded toward $-\infty$). This can be of interest on systems on which changing the rounding mode is an expensive operation, or even an impossible one (for instance, with the GCC compiler, the directed roundings are not supported correctly unless the `-frounding-math` switch is provided; but the implementation of this switch is incomplete, so that in practice, GCC currently assumes that the same rounding mode is used everywhere[1]). Moreover, directed roundings are heavily used in interval arithmetic or stochastic arithmetic [12].

In the following, we assume a radix-2, precision-p floating-point (FP) arithmetic. We assume an unbounded exponent range (which means that our results apply to "real life" floating-point arithmetic such as the one specified by the IEEE 754-2019 standard [13], provided that underflow and overflow do not occur). Hence, throughout this paper, the floating-point numbers are the numbers of the form

$$x = M_x \cdot 2^{e_x - p + 1},$$

where M_x and e_x are integers, with $|M_x| \leq 2^p - 1$. The number e_x is the floating-point exponent of x. We denote, as usual, $\mathrm{ulp}(x) = 2^{e_x - p + 1}$ and $u = 2^{-p}$ (the so-called "rounding unit"). $\mathrm{RU}(t)$ (a.k.a. t rounded toward $+\infty$) is the smallest floating-point number larger than or equal to t, $\mathrm{RD}(t)$ (a.k.a. t rounded toward $-\infty$) is the largest floating-point number less than or equal to t, and RN is the round-to-nearest function (with any choice in case of a tie).

We will need the following definition.

Definition 1: A real number x *fits in* t *bits* if there exist two integers L_x and N_x such that $|L_x| \leq 2^t - 1$ and $x = L_x \cdot 2^{N_x}$.

For instance, a real number is a floating-point number if it fits in p bits.

Section II presents a splitting algorithm for rounded toward $-\infty$ arithmetic (i.e., when `a T b` is called, with `T` $\in \{+, -, \times\}$, the value that is effectively calculated is $\mathrm{RD}(a \mathtt{T} b)$). In Section III, we adapt the previous splitting algorithm to rounded toward $+\infty$ arithmetic. In Section IV, we show that Dekker's original strategy can be used, with the splitting algorithms presented in this paper, for computing the exact product of two FP numbers.

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678#c1

II. Splitting algorithm for round-toward $-\infty$

A. From Veltkamp's splitting to the new algorithm

We aim at splitting a floating-point number a into two numbers a_h and a_ℓ such that $a = a_h + a_\ell$ and a_h is an approximation to a that fits in a given (significantly less than p) number of bits, with the consequence that a_ℓ also fits in a small number of bits. For computing the exact product of two FP numbers, we will actually need a_h to fit in $\lfloor p/2 \rfloor$ bits, and if we write $a_\ell = A_\ell \cdot \mathrm{ulp}(a)$, we will try to have a maximum value of $|A_\ell|$ as small as possible in order to be less than some upper bound (in practice, we will need A_ℓ^2 to be less than 2^p).

Veltkamp's splitting algorithm achieves this goal with operations rounded to nearest. With Veltkamp's splitting, a_h is the number a rounded to nearest in precision $\lfloor p/2 \rfloor$, which allows the maximum possible value of $|A_\ell|$ to be minimized. We wish to obtain a similar behavior with operations rounded toward $-\infty$ (a_h will still fit in $\lfloor p/2 \rfloor$ bits, but it will not necessarily be the $\lfloor p/2 \rfloor$-bit number nearest to a).

The proof of our algorithm will be rather involved (see Section II-C), but let us first explain the rough idea behind it.

Veltkamp's algorithm works by first building a number

$$c = \mathrm{RN}\left(\left(2^{\lceil p/2 \rceil} + 1\right) \cdot a\right),$$

in order to round the input a in a smaller precision, as follows:

$$a_h = \mathrm{RN}\big(\mathrm{RN}(a - c) + c\big) = \mathrm{RN}(a - c) + c.$$

Note that for a given a, the exact value of c does not necessarily matter: in short, only the exponent of $a - c$ matters, so that the rounding occurs at the expected ulp.

In our case, the operations are rounded toward $-\infty$ instead of to nearest. Just replacing the rounding function RN by RD in Veltkamp's algorithm does not always work, at least if p is odd, which is the case in binary64 ($p = 53$) and binary128 ($p = 113$) arithmetics: with $p = 11$ and

$$a = 2047 = 11111111111_2,$$

we obtain

$$c = 100000011110000000_2,$$

$$a_h = 1984 = 11111000000_2,$$

and

$$a_\ell = 111111_2,$$

which does not fit in 5 bits. As a matter of fact, replacing RN by RD in Veltkamp's algorithm has two effects: First, the value of c, used to obtain the wanted precision for a_h, is slightly different; but it turns out that this is not a big issue. And more importantly, the rounding of $a - c$ is no longer done to nearest, which frequently suffices for obtaining a wrong result. However, since in general (if we forget about values of x halfway between two floating-point numbers and negative powers of two), $\mathrm{RN}(x)$ is equal to

$$\mathrm{RD}\left(x + \frac{\mathrm{ulp}(x)}{2}\right), \tag{1}$$

we can try to "emulate" the calculation of $\mathrm{RN}(a - c)$ that appears in Veltkamp's algorithm by replacing it by $\mathrm{RD}(a^* - c)$, where a^* is slightly larger than a (with a difference that roughly corresponds to the term "$\mathrm{ulp}(x)/2$" in (1)). This is done by choosing

$$a^* = \mathrm{RD}\left(a \cdot k\right),$$

where k is some adequately chosen constant slightly larger than 1. This will not always round the number a to the nearest $\lfloor p/2 \rfloor$-bit number, but this will be enough to get an acceptable bound on $|A_\ell|$. This gives Algorithm 1 and Theorem 1 below.

Note: A "quick-and-dirty" reasoning to minimize the bound on $|A_\ell|$ allowed us to hint the adequate value of k; that reasoning was close to what is rigorously explained at the end of the proof of Theorem 1.

B. The splitting algorithm

Consider Algorithm 1 below. Note that the requirement $a \geq 0$ is important: there are negative values for which the algorithm does not work (an example is $p = 24$ and $a = -8391339$). This is not a problem for implementing high-precision multiplication: it suffices to handle the signs separately.

ALGORITHM 1: SplitRD. Splitting algorithm for round-toward $-\infty$. We assume $a \geq 0$.

1: **uses** $s = \lceil p/2 \rceil$ **and** $k = \mathrm{RN}\left(1 + \frac{2}{3} \cdot 2^{-\lfloor p/2 \rfloor}\right)$.
2: $a^* \leftarrow \mathrm{RD}(a \cdot k)$
3: $c \leftarrow \mathrm{RD}((2^s + 1) \cdot a^*)$
4: $d \leftarrow \mathrm{RD}(a^* - c)$
5: $a_h \leftarrow \mathrm{RD}(c + d)$
6: $a_\ell \leftarrow \mathrm{RD}(a - a_h)$
7: **return** (a_h, a_ℓ)

Algorithm 1 satisfies:

Theorem 1: Assuming $p \geq 2$, the two numbers a_h and a_ℓ returned by Algorithm 1 satisfy:

- $a_h + a_\ell = a$;
- a_h fits in $p - s = \lfloor p/2 \rfloor$ bits, it is a multiple of $2^s \mathrm{ulp}(a)$, and $a_h \leq 2^{e_a + 1}$, where e_a is the floating-point exponent of a;
- a_ℓ is of the form

  $$A_\ell \cdot \mathrm{ulp}(a),$$

  where $|A_\ell|$ is an integer satisfying

  $$|A_\ell| \leq \frac{4}{3} \cdot 2^{\lceil p/2 \rceil - 1} + \frac{5}{2},$$

  $A_\ell^2 < 2^p$ and $|a_\ell| < 2^{e_a - p/2 + 1}$.

□

The proof given in Section II-C assumes $p \geq 6$ for p even and $p \geq 11$ for p odd (in particular for the last properties of Theorem 1). For smaller precisions p, we have run exhaustive

tests showing that all these properties remain satisfied as soon as $p \geq 2$.

Before giving this proof, let us give an example to illustrate how the algorithm works. Consider the case $p = 11$, for which $k = 1045/1024 = 1.0000010101 \times 2^0$, and take

$$a = 2047 = 11111111111_2.$$

We successively obtain:

- $a^* = 2088 = 100000101000_2$;
- $c = 135680$;
- $d = -133632$;
- $a_h = 2^{11}$;
- $a_\ell = -1$.

Finally, in Algorithm 1, the fact that the rounding is toward $-\infty$ is important. For instance, if we replace in the algorithm the round-toward $-\infty$ rounding function RD by the round-toward $+\infty$ rounding function RU, the algorithm does not work for $p = 53$ and $a = 2^{52} + 1$: the returned value of a_ℓ is $-2^{27} + 1$, whose square is larger than 2^{53}.

C. Proof of Theorem 1

Let λ be such that

$$k = 1 + \lambda \cdot 2^{s-p} = 1 + \lambda \cdot 2^{-\lfloor p/2 \rfloor}.$$

The choice of k in Algorithm 1 implies $\lambda \approx 2/3$ but temporarily, let us just assume $0 < \lambda \leq 1$ (we will explain the choice of k and λ later on). Note that this implies $1 < k < 2$, and therefore the number $\mathrm{ulp}(a^*)$ is equal to $\mathrm{ulp}(a)$ or to $2 \cdot \mathrm{ulp}(a)$.

Line 2 of the algorithm gives

$$\begin{aligned} 0 \leq a \cdot k - a^* &< \mathrm{ulp}(a \cdot k) \\ &\leq 2u \cdot a \cdot k = a \cdot k \cdot 2^{1-p}. \end{aligned}$$

Now, consider **Line 3**. We have

$$0 \leq (2^s + 1) \cdot a^* - c < \mathrm{ulp}(c),$$

so that

$$-2^s a^* \leq a^* - c < -2^s a^* + \mathrm{ulp}(c).$$

The number a^* can be written

$$a^* = 2^e \cdot (1 + n \cdot 2^{1-p}), \tag{2}$$

where $e = e_{a^*}$ is the floating-point exponent of a^* and n is an integer such that $0 \leq n \leq 2^{p-1} - 1$. We have:

- if $n = 0$ then a^* is a power of 2. Hence

  $$c = (2^s + 1) \cdot a^*,$$

  so that $a^* - c = -2^s a^*$, which implies

  $$\mathrm{ulp}(a^* - c) = 2^s \mathrm{ulp}(a^*);$$

- if $n = 1$ then $(2^s + 1) \cdot a^*$ is equal to

  $$2^e \cdot \left(2^s + 1 + 2^{1-\lfloor p/2 \rfloor} + 2^{1-p}\right),$$

  which is less than $2^e \cdot (2^s + 3)$ as soon as $p \geq 2$. If $p \geq 3$, then $2^s + 3 < 2^{s+1}$, therefore

  $$\mathrm{ulp}(c) = \mathrm{ulp}(2^s a^*) = 2^{e+s+1-p};$$

- if $n \geq 2$ then, since $c = \mathrm{RD}((2^s + 1) \cdot a^*) \leq 2^{s+1} \cdot a^*$, we have

  $$\mathrm{ulp}(c) \leq 2^{s+1} \mathrm{ulp}(a^*) = 2^{s+e+2-p} \leq n \cdot 2^{s+e+1-p}.$$

Therefore, if $n \geq 1$ and $p \geq 3$, we have

$$-2^s a^* \leq a^* - c \leq -2^s a^* + n \cdot 2^{s+e+1-p}. \tag{3}$$

Replacing a^* in the right-hand part of (3) by the expression given in (2), we obtain -2^{e+s}. Hence, in all cases, we have

$$\mathrm{ulp}(a^* - c) = 2^{e+s+1-p} = \mathrm{ulp}(2^s a^*) = 2^s \mathrm{ulp}(a^*). \tag{4}$$

This allows us to deal with **Line 4** of the algorithm. From (4), we deduce that the floating-point number d computed at that line is the largest multiple of $2^s \mathrm{ulp}(a^*)$ less than or equal to $a^* - c$. Therefore, one can write

$$a^* - c = d + f,$$

with $0 \leq f < 2^s \mathrm{ulp}(a^*)$. When $\mathrm{ulp}(a^*) = \mathrm{ulp}(a)$, this obviously gives

$$0 \leq f < 2^s \mathrm{ulp}(a). \tag{5}$$

Let us show that (5) remains true even when $\mathrm{ulp}(a^*) = 2 \cdot \mathrm{ulp}(a)$. When this is the case, we have

$$2^{e-1} < a < 2^e \text{ and } \mathrm{ulp}(a) = 2^{e-p}. \tag{6}$$

$$2^e \leq a^* < 2^e \cdot k = 2^e \cdot (1 + \lambda \cdot 2^{s-p}),$$

so that

$$2^e - c \leq a^* - c < 2^e - c + \lambda \cdot 2^{e+s-p}. \tag{7}$$

Since $2^s \mathrm{ulp}(a^*) = 2^{e+s-p+1}$, the number 2^e is a multiple of $2^s \mathrm{ulp}(a^*)$. Also, $c \geq 2^s a^*$ implies $\mathrm{ulp}(c) \geq 2^s \mathrm{ulp}(a^*)$, therefore c is a multiple of $2^s \mathrm{ulp}(a^*)$. As a consequence, $2^e - c$ is a multiple of $2^s \mathrm{ulp}(a^*)$. Therefore, from (7), the number

$$f = (a^* - c) \bmod 2^s \mathrm{ulp}(a^*)$$

is less than $\lambda \cdot 2^{e+s-p}$, which, from (6), is equal to $\lambda \cdot 2^s \mathrm{ulp}(a)$, which (since $\lambda \leq 1$) is less than or equal to $2^s \mathrm{ulp}(a)$, so that (5) still holds.

Now, consider **Line 5** of Algorithm 1. From

$$2^s a^* \leq c \leq (2^s + 1) \cdot a^*,$$

we deduce

$$(2^s - 1) \cdot a^* \leq c - a^* \leq -d = |d|, \tag{8}$$

Also, since $a^* \geq a \geq 0$, we have

$$d = \mathrm{RD}(a^* - c) \geq \mathrm{RD}(-c) = -c,$$

so that

$$|d| = -d \leq c \leq (2^s + 1) \cdot a^*. \tag{9}$$

From (8) and (9), we obtain

$$1 \le \frac{c}{|d|} \le \frac{2^s+1}{2^s-1}. \tag{10}$$

If $p \ge 3$, then $s \ge 2$, so that (10) gives

$$1 \le \frac{c}{|d|} \le \frac{5}{3}.$$

Therefore, Sterbenz Lemma [14], [7] implies that the operation performed at Line 5 is exact, i.e., we have

$$a_h = c + d = a^* - f. \tag{11}$$

Since c and d are multiples of $2^s\text{ulp}(a^*)$, a_h is a multiple of $2^s\text{ulp}(a^*)$ (and therefore a multiple of $2^s\text{ulp}(a)$). Also $a_h = a^* - f$ is less than or equal to a^*. It follows that a_h fits in $p - s = \lfloor p/2 \rfloor$ bits.

There remains to consider the computation of a_ℓ at **Line 6** of Algorithm 1. We have

$$a_\ell = \text{RD}(a - a_h) = \text{RD}(a - a^* + f). \tag{12}$$

From (9) we obtain $c + d \ge 0$, so that $a_h \ge 0$.

If $a_h = 0$, then a_ℓ is computed exactly (incidentally, in such a case, one can show that $a = 0$, so that $a_\ell = 0$ too).

Now assume $a_h > 0$. We have

$$\begin{array}{rcl} a - a_h & = & a - a^* + f \\ & = & (a - a\cdot k) + (a\cdot k - a^*) + f. \end{array} \tag{13}$$

In (13), the term $(a - a\cdot k)$ is equal to $-\lambda\cdot a\cdot 2^{-\lfloor p/2\rfloor}$, the term $(a\cdot k - a^*)$ lies in the interval $[0, \text{ulp}(a^*))$, and the term f lies in $[0, 2^s\text{ulp}(a))$. Therefore,

$$\begin{array}{c} -\lambda\cdot a\cdot 2^{-\lfloor p/2\rfloor} \\ \le a - a_h \\ < -\lambda\cdot a\cdot 2^{-\lfloor p/2\rfloor} + \text{ulp}(a^*) + 2^s\text{ulp}(a) \\ \le -\lambda\cdot a\cdot 2^{-\lfloor p/2\rfloor} + (2^s+2)\,\text{ulp}(a). \end{array} \tag{14}$$

The number $a - a_h$ is a multiple of $\text{ulp}(a)$. From (14), its absolute value is less than or equal to

$$\max\left\{\lambda\cdot a\cdot 2^{-\lfloor p/2\rfloor}; -\lambda\cdot a\cdot 2^{-\lfloor p/2\rfloor} + (2^s+2)\text{ulp}(a)\right\},$$

which is less that $2^p\text{ulp}(a)$ as soon as $2^s+2 \le 2^p$, i.e., as soon as $p \ge 2$. Hence, under the hypothesis $p \ge 3$ of the theorem, $a - a_h$ is a floating-point number, therefore

$$a_\ell = a - a_h.$$

Now, there remains to explain the choice of k (hence, the choice of λ) in Algorithm 1, and to bound $|a_\ell|$ as tightly as possible. Let e_a be the floating-point exponent of a. If e_a is fixed, in (14), the left-hand bound attains its maximum absolute value, slightly less than

$$2\lambda\cdot 2^{-\lfloor p/2\rfloor}\cdot 2^{e_a},$$

when

$$a = \left(2 - 2^{-p+1}\right)\cdot 2^{e_a},$$

and the right-hand bound attains its maximum absolute value, equal to

$$\left(-\lambda\cdot 2^{-\lfloor p/2\rfloor} + (2^s+2)\cdot 2^{1-p}\right)\cdot 2^{e_a},$$

when $a = 2^{e_a}$. Therefore, we have

$$\begin{array}{c} -2\lambda\cdot 2^{-\lfloor p/2\rfloor} \le \dfrac{a_\ell}{2^{e_a}} \\ \le -\lambda\cdot 2^{-\lfloor p/2\rfloor} + (2^s+2)\cdot 2^{1-p}. \end{array} \tag{15}$$

The best choice of λ (i.e., the one for which the bound on $|a_\ell|$ is as small as possible) is when the absolute values of both bounds of (15) are equal, i.e. when

$$3\lambda\cdot 2^{s-p} = (2^s+2)\cdot 2^{1-p},$$

i.e., when

$$\lambda = \frac{2+2^s}{3}\cdot 2^{1-s} \approx \frac{2}{3}.$$

This explains the choice $k = \text{RN}\left(1 + \frac{2}{3}\cdot 2^{-\lfloor p/2\rfloor}\right)$ in Algorithm 1. In the following, we assume that k has this value. We have

$$1 + \frac{2}{3}\cdot 2^{-\lfloor p/2\rfloor} - 2^{-p} \le k \le 1 + \frac{2}{3}\cdot 2^{-\lfloor p/2\rfloor} + 2^{-p},$$

therefore,

$$\frac{2}{3} - 2^{-\lceil p/2\rceil} \le \lambda \le \frac{2}{3} + 2^{-\lceil p/2\rceil}. \tag{16}$$

Using (15) and (16), we obtain

$$\begin{array}{c} \dfrac{|a_\ell|}{2^{e_a}} \le \max\left\{\dfrac{4}{3}\cdot 2^{-\lfloor p/2\rfloor} + 2^{-p+1};\right. \\ \left. -\frac{2}{3}\cdot 2^{-\lfloor p/2\rfloor} + 2^{-p} + 2^{-\lfloor p/2\rfloor+1} + 2^{2-p}\right\}, \end{array}$$

i.e.,

$$\begin{array}{rcl} \dfrac{|a_\ell|}{2^{e_a}} & \le & \max\left\{\dfrac{4}{3}\cdot 2^{-\lfloor p/2\rfloor} + 2^{-p+1};\right. \\ & & \left.\dfrac{4}{3}\cdot 2^{-\lfloor p/2\rfloor} + 5\cdot 2^{-p}\right\} \\ & = & \dfrac{4}{3}\cdot 2^{-\lfloor p/2\rfloor} + 5\cdot 2^{-p}. \end{array}$$

Hence,

$$\begin{array}{c} a_\ell = A_\ell\cdot\text{ulp}(a) = A_\ell\cdot 2^{e_a-p+1}, \\ \text{with } |A_\ell| \le \dfrac{4}{3}\cdot 2^{\lceil p/2\rceil-1} + \dfrac{5}{2}. \end{array} \tag{17}$$

- If p is even, (17) gives

  $$|A_\ell| \le \frac{4}{3}\cdot 2^{p/2-1} + \frac{5}{2},$$

  hence

  $$A_\ell^2 \le \frac{16}{9}\cdot 2^{p-2} + \frac{20}{3}\cdot 2^{p/2-1} + \frac{25}{4},$$

  which is less than 2^p as soon as $p \ge 6$.
- If p is odd, (17) gives

  $$|A_\ell| \le \frac{4}{3}\cdot 2^{(p-1)/2} + \frac{5}{2},$$

hence

$$A_\ell^2 \le \frac{16}{9}\cdot 2^{p-1} + \frac{20}{3}\cdot 2^{(p-1)/2} + \frac{25}{4},$$

which is less than 2^p as soon as $p \ge 11$.

It remains to show that $a_h \le 2^{e_a+1}$, where e_a is the floating-point exponent of a. This is an immediate consequence of the fact that

$$|A_\ell| \le \frac{2}{3}\cdot 2^s + \frac{5}{2} < 2^s \quad \text{as soon as } p \ge 5,$$

and

$$a_h \le a + |a_\ell| < 2^{e_a+1} + |A_\ell|\cdot \mathrm{ulp}(a) < 2^{e_a+1} + 2^s \mathrm{ulp}(a),$$

where both a_h and 2^{e_a+1} are multiples of $2^s\mathrm{ulp}(a)$.

□

III. Splitting algorithm assuming round-toward $+\infty$

One easily derives from Algorithm 1 a splitting algorithm for the round-toward $+\infty$ rounding function RU: this is Algorithm 2 below. The proof immediately follows from Theorem 1, by using the relation

$$\mathrm{RU}(x) = -\mathrm{RD}(-x),$$

by noting that the constant k' of Algorithm 2 is the opposite of the constant k of Algorithm 1, and by using the fact (shown when proving Theorem 1) that the last operation is exact.

ALGORITHM 2: SplitRU. Splitting algorithm for round-toward $+\infty$. We assume $a \ge 0$.

1: **uses** $s = \lceil p/2 \rceil$ **and** $k' = -\mathrm{RN}\left(1 + \frac{2}{3}\cdot 2^{-\lfloor p/2 \rfloor}\right)$.
2: $a^* \leftarrow \mathrm{RU}(a \cdot k')$
3: $c \leftarrow \mathrm{RU}((2^s+1)\cdot a^*)$
4: $d \leftarrow \mathrm{RU}(a^* - c)$
5: $a_h \leftarrow -\mathrm{RU}(c + d)$
6: $a_\ell \leftarrow \mathrm{RU}(a - a_h)$
7: **return** (a_h, a_ℓ)

We have,

Theorem 2: Assuming $p \ge 2$, the two numbers a_h and a_ℓ returned by Algorithm 2 satisfy:

- $a_h + a_\ell = a$;
- a_h fits in $p - s = \lfloor p/2 \rfloor$ bits, it is a multiple of $2^s\mathrm{ulp}(a)$, and $a_h \le 2^{e_a+1}$, where e_a is the floating-point exponent of a;
- a_ℓ is of the form

  $$A_\ell \cdot \mathrm{ulp}(a),$$

  where $|A_\ell|$ is an integer satisfying

  $$|A_\ell| \le \frac{4}{3}\cdot 2^{\lceil p/2 \rceil - 1} + \frac{5}{2},$$

  $A_\ell^2 < 2^p$ and $|a_\ell| < 2^{e_a - p/2 + 1}$.

□

IV. Exact multiplication using Algorithm 1 for splitting the operands

We have slightly adapted Dekker's classical multiplication algorithm [10], by using the round-to $-\infty$ rounding function RD (instead of the round-to-nearest function RN), and by using Algorithm 1 for splitting the operands. This gives Algorithm 3 below.

ALGORITHM 3: Dekker's product with rounding toward $-\infty$. It returns two FP numbers r_1 and r_2 such that $r_1 + r_2 = ab$.

We assume $a \ge 0$ and $b \ge 0$.

Require: $s = \lceil p/2 \rceil$
Ensure: $r_1 + r_2 = a \cdot b$
1: $(a_h, a_\ell) \leftarrow \mathrm{SplitRD}(a)$
2: $(b_h, b_\ell) \leftarrow \mathrm{SplitRD}(b)$
3: $r_1 \leftarrow \mathrm{RD}(a \cdot b)$
4: $t_1 \leftarrow \mathrm{RD}(-r_1 + \mathrm{RD}(a_h \cdot b_h))$
5: $t_2 \leftarrow \mathrm{RD}(t_1 + \mathrm{RD}(a_h \cdot b_\ell))$
6: $t_3 \leftarrow \mathrm{RD}(t_2 + \mathrm{RD}(a_\ell \cdot b_h))$
7: $r_2 \leftarrow \mathrm{RD}(t_3 + \mathrm{RD}(a_\ell \cdot b_\ell))$
8: **return** (r_1, r_2)

We have,

Theorem 3: If $p \ge 11$, the two floating-point numbers r_1 and r_2 returned by Algorithm 3 satisfy

$$r_1 + r_2 = a \cdot b.$$

□

Proof. Without loss of generality, we assume $1 \le a < 2$ and $1 \le b < 2$. If $p \ge 11$, the analysis of Algorithm 1 shows that a_h and b_h are multiples of 2^{s-p+1}, $a_h \le 2$, $b_h \le 2$, $|a - a_h| < 2^{-p/2+1}$, and $|b - b_h| < 2^{-p/2+1}$. Theorem 1 implies that $a_h \cdot b_h$, $a_h \cdot b_\ell$, $a_\ell \cdot b_h$, and $a_\ell \cdot b_\ell$ are exactly computed, i.e.,

$$\mathrm{RD}(a_h \cdot b_h) = a_h \cdot b_h,$$

$$\mathrm{RD}(a_h \cdot b_\ell) = a_h \cdot b_\ell,$$

$$\mathrm{RD}(a_\ell \cdot b_h) = a_\ell \cdot b_h,$$

and

$$\mathrm{RD}(a_\ell \cdot b_\ell) = a_\ell \cdot b_\ell.$$

We have

$$ab - a_h b_h = (a - a_h)\cdot b + (b - b_h)\cdot a_h,$$

therefore

$$|ab - a_h b_h| \le 2^{-p/2+3}.$$

Since $|ab - r_1| < \mathrm{ulp}(ab) \le 2^{-p+2}$, we deduce

$$|-r_1 + a_h b_h| \le 2^{-p/2+3} + 2^{-p+2}. \tag{18}$$

Since r_1 is a floating-point number such that $|r_1| \geq 1$, it is a multiple of $\mathrm{ulp}(1) = 2^{-p+1}$. Since a_h and b_h are multiples of 2^{s-p+1}, their product $a_h b_h$ is a multiple of $2^{2s-2p+2} \geq 2^{-p+2}$. Therefore, $-r_1 + a_h b_h$ is a multiple of 2^{-p+1}, and (18) with $p \geq 5$ implies that it is a floating-point number. As a consequence,

$$t_1 = -r_1 + a_h b_h.$$

Now, we have

$$ab = a_h b_h + a_h b_\ell + a_\ell b_h + a_\ell b_\ell,$$

therefore,

$$\begin{aligned} &|t_1 + a_h b_\ell| \\ &= |-r_1 + a_h b_h + a_h b_\ell| \\ &= |-r_1 + ab + (a_h b_h + a_h b_\ell - ab)| \\ &\leq |-r_1 + ab| + |a_\ell (b_h + b_\ell)|\,. \end{aligned} \tag{19}$$

For $p \geq 11$, Theorem 1 implies that $|a_\ell|$ is bounded by $2^{-p/2+1/2}$ if p is even, and by $2^{-p/2+1}$ if p is odd. Therefore, from (19), we conclude that $|t_1 + a_h b_\ell|$ is less than

$$2^{-p+2} + 2^{-p/2+3/2}$$

if p is even, and less than

$$2^{-p+2} + 2^{-p/2+2}$$

if p is odd. Furthermore, since a_h is a multiple of 2^{s-p+1} and b_ℓ is a multiple of 2^{-p+1}, we deduce that $|t_1 + a_h b_\ell|$ is a multiple of 2^{s-2p+2}. Therefore:

- if p is even, then

  $$2^{s-2p+2} = 2^{-3p/2+2}.$$

  Hence the number $|t_1 + a_h b_\ell|$ is of the form

  $$K \cdot 2^{s-2p+2},$$

  where K is an integer and

  $$|K| \leq 2^{p-1/2} + 2^{p/2} < 2^p$$

  (as soon as $p \geq 4$);
- if p is odd, then

  $$2^{s-2p+2} = 2^{-3p/2+5/2}.$$

  Hence the number $|t_1 + a_h b_\ell|$ is of the form

  $$K \cdot 2^{s-2p+2},$$

  where K is an integer and

  $$|K| \leq 2^{p-1/2} + 2^{p/2-1/2} < 2^p$$

  (as soon as $p \geq 3$).

Therefore, in all cases, $t_1 + a_h b_\ell$ is a floating-point number. This gives

$$t_2 = t_1 + a_h b_\ell = -r_1 + a_h b_h + a_h b_\ell.$$

Now,

$$t_2 + a_\ell b_h = (-r_1 + ab) - a_\ell b_\ell.$$

Hence,

$$\begin{aligned} |t_2 + a_\ell b_h| &\leq |-r_1 + ab| + |a_\ell b_\ell| \\ &\leq 2^{-p+2} + 2^{-p+2} \\ &\leq 2^{-p+3}. \end{aligned} \tag{20}$$

Again, since t_2 is a multiple of 2^{s-2p+2}, b_h is a multiple of 2^{s-p+1} and a_ℓ is a multiple of 2^{-p+1}, the number $t_2 + a_\ell b_h$ is a multiple of 2^{s-2p+2}. Therefore, $t_2 + a_\ell b_h$ is of the form

$$K \cdot 2^{s-2p+2},$$

where K is an integer of absolute value less than $2^{-p+3}/2^{s-2p+2} < 2^p$. Therefore, $t_2 + a_\ell b_h$ is a floating-point number, which gives

$$t_3 = t_2 + a_\ell b_h = -r_1 + a_h b_h + a_h b_\ell + a_\ell b_h.$$

Finally,

$$t_3 + a_\ell b_\ell = -r_1 + ab.$$

From this, we deduce that $|t_3 + a_\ell b_\ell|$ is less than 2^{-p+2}. Since it is a multiple of 2^{-2p+2}, it is a floating-point number, therefore

$$r_2 = -r_1 + ab,$$

which is what we wanted to show.

□

Important remark: we have seen in the proof of Theorem 3 that the operations performed at lines 4 to 7 of Algorithm 3 are exact operations. Therefore, one obtains exactly the same results if the rounding function RD is replaced by another one. From this, one easily deduces that if the available rounding function is RU, it suffices to replace the calls to SplitRD (i.e., Algorithm 1) at lines 1-2 by calls to SplitRU (i.e., Algorithm 2), and to replace RD by RU at lines 3-7, to obtain an exact multiplication algorithm that only uses rounding toward $+\infty$.

V. Timings

Veltkamp's algorithm (using rounding function RN), Algorithm 1 (using rounding function RD) and Algorithm 2 (using rounding function RU) have been implemented in C using the floating-point type `double`. The x86_64 assembly code, generated under a Debian/unstable machine with the `-frounding-math -std=c11 -O3 -march=native` GCC-compatible options, has been analyzed. For each algorithm, in the code of the function, one can see the instructions matching the operations from the algorithm. The negation is implemented by a XOR, i.e., an integer operation. Since the code of the function needs to follow the ABI, the assembly code also contains some "move" instructions. A GCC 10 preversion (provided by the `gcc-10` Debian package) and Clang 9 generate similar code, while GCC 9.2.1 can generate an additional move instruction. But when the function is inlined, these move instructions are no longer needed.

We obtain timings in the following way: an array of random inputs has been generated (not too many so that they can fit

into the cache), and each input is tested in an internal loop; the function should automatically be inlined by the compiler. An external loop runs the internal loop a large number of times so that the test lasts long enough to get meaningful timings. Two multiplications are done on the floating-point values returned by the function, and the result is accumulated, in order to make sure that all results are used, and avoid some optimizations related to the context. The running time is measured with the `clock` function.

The code has been tested on an Intel Xeon CPU E5-2609 v3 with the GCC 10 preversion. We could also see that the compiler automatically vectorized the code (the multiplications used for the test are also vectorized, but the accumulation is not, in order to conform to IEEE 754). The `-fno-tree-vectorize` option can be provided to avoid automatic vectorization; we checked that in the generated code. With an array of 32, 64, 128 or 256 inputs, without vectorization, Algorithm 1 takes about 10% more time than Veltkamp's algorithm (the overhead is included in the timings) and Algorithm 2 takes about 17% more time; with vectorization, all 3 algorithms take the same time. The code has been tested on other machines, and the following increases with our algorithms have been observed: $(+4\%, +7\%)$ on a POWER9 machine, $(+15\%, +15\%)$ on an AArch64 (64-bit ARM) machine, up to $(+17\%, +34\%)$ on an AMD Opteron machine.

Importantly enough, modifying the test code just by adding a `fesetround(FE_TONEAREST)` before executing Veltkamp's algorithm instead of using our algorithms can yield a loss of a factor 5.

In short, the timings depend very much on the context (the caller code, the processor, the compiler options, etc.), but in any case, there are no significant differences in terms of delay between the classical Veltkamp-Dekker algorithms and the algorithms introduced in this paper, which makes our algorithms attractive when used in applications that already use a directed rounding.

The source code for the exhaustive tests mentioned in Section II-B and for the timings is available at

https://hal.archives-ouvertes.fr/hal-02470782

Conclusion

We have proposed alternatives to Veltkamp's splitting and Dekker's product for rounded toward $-\infty$ and rounded toward $+\infty$ arithmetics. This can be of interest on architectures on which changing the rounding mode is an expensive operation, or even an impossible one.

According to exhaustive tests in small precisions, the chosen value $k = \mathrm{RN}\left(1 + \frac{2}{3} \cdot 2^{-\lfloor p/2 \rfloor}\right)$ in Algorithm 1 seems to be the best possible one together with $\mathrm{RD}\left(1 + \frac{2}{3} \cdot 2^{-\lfloor p/2 \rfloor}\right)$, which is different from k for some values of p. Both of these choices yield a maximum value of $|A_\ell|$ equal to $\lfloor \frac{4}{3} \cdot 2^{\lceil p/2 \rceil - 1} \rfloor$. Future work could consist in proving these properties in any precision.

Acknowledgments

The authors wish to thank the reviewers for their helpful comments.

References

[1] Y. Hida, X. S. Li, and D. H. Bailey, "C++/Fortran-90 double-double and quad-double package, release 2.3.22," Feb. 2019, accessible electronically at https://www.davidhbailey.com/dhbsoftware/.

[2] ——, "Algorithms for quad-double precision floating-point arithmetic," in *15th IEEE Symposium on Computer Arithmetic (ARITH-15)*, Jun. 2001, pp. 155–162.

[3] M. Joldeş, J.-M. Muller, and V. Popescu, "Tight and rigourous error bounds for basic building blocks of double-word arithmetic," *ACM Transactions on Mathematical Software*, vol. 44, no. 2, Oct. 2017.

[4] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005.

[5] S. Graillat, P. Langlois, and N. Louvet, "Algorithms for accurate, validated and fast computations with polynomials," *Japan Journal of Industrial and Applied Mathematics*, vol. 26, no. 2, pp. 215–231, 2009.

[6] W. Kahan, "Lecture notes on the status of IEEE-754," 1997, available at https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF.

[7] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2018.

[8] G. W. Veltkamp, "ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie," RC-Informatie, Technishe Hogeschool Eindhoven, Tech. Rep. 22, 1968.

[9] ——, "ALGOL procedures voor het rekenen in dubbele lengte," RC-Informatie, Technishe Hogeschool Eindhoven, Tech. Rep. 21, 1969.

[10] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[11] C.-P. Jeannerod, J. Muller, and P. Zimmermann, "On various ways to split a floating-point number," in *25th IEEE Symposium on Computer Arithmetic, Amherst, MA, USA*, June 2018, pp. 53–60.

[12] S. Graillat and F. Jézéquel, "Tight interval inclusions with compensated algorithms," *IEEE Transactions on Computers*, pp. 1–1, 2019.

[13] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, Jul. 2019. [Online]. Available: https://doi.org/10.1109/IEEESTD.2019.8766229

[14] P. H. Sterbenz, *Floating-Point Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1974.