Formalization of double-word arithmetic, and comments on "Tight and rigorous error bounds for basic building blocks of double-word arithmetic"

Jean-Michel Muller, Laurence Rideau

Formalization of double-word arithmetic, and comments on "Tight and rigorous error bounds for basic building blocks of double-word arithmetic"

Jean-Michel Muller
CNRS, LIP, Université de Lyon
France

Laurence Rideau
Inria Sophia Antipolis, Université Côte d'Azur
France

August 31, 2021

keywords Floating-point arithmetic; double-word arithmetic; double-double arithmetic; formalization; proof assistant; Coq.

Abstract

Recently, a complete set of algorithms for manipulating double-word numbers (some classical, some new) was analyzed [16]. We have formally proven all the theorems given in that paper, using the Coq proof assistant. The formal proof work led us to: i) locate mistakes in some of the original paper proofs (mistakes that, however, do not hinder the validity of the algorithms), ii) significantly improve some error bounds, and iii) generalize some results by showing that they are still valid if we slightly change the rounding mode. The consequence is that the algorithms presented in [16] can be used with high confidence, and that some of them are even more accurate than what was believed before. This illustrates what formal proof can bring to computer arithmetic: beyond mere (yet extremely useful) verification, correction and consolidation of already known results, it can help to find new properties. All our formal proofs are freely available.

1 Introduction

Double-word arithmetic, frequently called "double-double" arithmetic, consists of representing a real number as the unevaluated sum of two floating-point (FP) numbers, with some additional constraints needed to make sure that this is an accurate representation (see Definition 1 below). Double-word (and, more generally, multiple-word)

arithmetic is useful when the largest FP format available in hardware on the system being used is not wide enough for some critical parts of a numerical algorithm. Typical examples occur in computational geometry [28], in the design of accurate BLAS [20], and the design of libraries for correctly rounded elementary functions [7].

In all widely available distributions of double-word arithmetic, the underlying FP format is the *binary64*, a.k.a. "double precision" format of the IEEE 754-2019 Standard for Floating-Point Arithmetic [13], which explains the name "double-double". The first double-word algorithms were given by Dekker [8] as early as 1971. A largely used library that provides efficient double-word (and quad-word) arithmetic is the QD ("quad-double") library by Hida, Li, and Bailey [11, 12]. A more recent library is Campary [17] (see also [14] for a performance evaluation of Campary).

In [16], the authors (including one of us) consider several arithmetic algorithms for manipulating double-word numbers (some taken from the literature, some new), and give paper-and-pencil proofs of error bounds for these algorithms. We felt that building formal proofs of these algorithms was necessary for several reasons:

- These algorithms are specifically designed to be used in *critical parts* of numerical programs, where a careful control of numerical error is important, so we need to be *absolutely certain* about the claimed error bounds;
- despite many "without loss of generality we assume that…" and the use of symmetries whenever possible, the paper-and-pencil proofs remain rather long, complex, and at times rather tedious to read (mainly because of unavoidable enumerations of many possible cases). This has two consequences: first, it is quite possible that some errors remain hidden in the proofs and, second, few people will carefully read them;
- with this class of algorithms, testing is a doubtful option. Even if this does not provide absolute certainty, a frequent way of increasing confidence in published error bounds is to perform extensive tests. This does not work well with this family of algorithms, because the cases for which the relative error is close to maximum are extremely rare. Hence even when performing millions of random tests, one may miss the worst case by a very significant margin. An example, discussed in Section 2.2, is Algorithm 4 below ("accurate sum" of two double-word numbers). It computes the sum of two double-word numbers. It was presented in [21, 20], and it is Algorithm 6 in [16]. For that algorithm, the authors of [16] could build a "bad case" with error around $2.25u^2$ (where u, the *rounding unit*, is defined later on). We give an even worse case (close to $3u^2$) in this paper. To obtain such bad cases, one needs to build them: they are so rare that one can perform millions of random tests without finding relative errors larger than $2u^2$.

It clearly appeared, when working on these formal proofs, that there are further benefits. As we are going to see, the formal proof work also allows one to do more than just checking the validity of a proof or pointing-out and correcting the errors:

- beyond simply checking a paper-and-pencil proof, formalization helps to understand all details of a proof, sometimes it allows one to prove a stronger re-

sult, or to improve or simplify the proof. Especially, one can at times *generalize* the results, by checking if the proof (possibly with small modifications) still holds with more general assumptions: in that case, it is not mere (yet useful!) verification of existing results, but discovery of new results.

- even once the paper proof is verified (and corrected if needed), the formal proof serves as a detailed appendix to the paper proof: shortcuts such as "the second case is symmetrical to the first one…", or "without loss of generality, we assume that…" are frequently necessary in a paper proof for the sake of clarity and/or brevity. The formal proof contains all the details (exhibition of symmetries, changes of values, …) that prove that these shortcuts were valid.

The use of formal proof tools for giving more confidence in computer arithmetic algorithms has a long history, that goes back at least to the years that followed the Pentium FDIV Bug [5]. This is not surprising: all of numerical computing is built upon the underlying arithmetic of our computers. If the arithmetic cannot be trusted, nothing can be trusted. We therefore must have full confidence in that underlying arithmetic. In some cases (unary functions and very small values of the precision p), exhaustive testing of the programs is possible,[1] but in general, formal proof is the only way of obtaining that confidence.

Among these first works on formal proof of floating-point algorithms, let us mention works by Harrison [9, 10], Moore et al. [23], Daumas, Rideau and Théry [6]. Our formal proofs are built using Boldo and Melquiond's Flocq library [3, 4], built on the Coq proof assistant.[2] Coq (see for instance [2]) is based on the calculus of inductive constructions. It provides interactive proof methods and a tactic language to help the user to define new proof methods. It also makes it possible to extract programs from proofs. The Flocq library was for instance used by Boldo et al. to show the correctness of the floating-point passes of the verified CompCert C compiler [1].

In the following, we assume that the floating-point arithmetic upon which we build a double-word arithmetic is a radix-2, precision-p FP arithmetic system, with unlimited exponent range (which means that the results presented here apply to "real life" floating-point arithmetic provided that underflows and overflows do not occur). This means that an FP number x is a number of the form

$$x = M \cdot 2^{e-p+1}, \tag{1}$$

where M and e are integers, with $|M| \leq 2^p - 1$. If $x \neq 0$, there is a unique pair (M_x, e_x) that satisfies both (1) and the additional requirement

$$2^{p-1} \leq |M| \leq 2^p - 1.$$

The number e_x from that pair is called the *floating-point exponent* of x, and M_x is called the *integral significand* of x. We will say that a FP number is *even* if its integral significand is even.

[1] For instance, the simplest way of verifying a single-precision implementation of the sine or exponential function is to check what it returns for each of the 2^{32} possible input values. On a recent laptop, this takes at most a few hours.

[2] `http://coq.inria.fr/`

The notation RN(t) stands for t rounded to the nearest FP number, and unless stated otherwise, we assume that we use the *ties-to-even* tie-breaking rule. It is defined as follows:

- if there is only one FP number nearest to t then RN(t) is that number,
- and if t is exactly halfway between two consecutive FP numbers, then RN(t) is the one of these two numbers whose integral significand is even.

Round-to-nearest ties-to-even is the default rounding mode in the IEEE 754-2019 Standard [13], and it is by far the most used (few people consider changing the rounding mode, and programming environments do not always make that an easy task). Hence, when an arithmetic operation $\mathtt{c} \top \mathtt{d}$ is performed, the result that is actually returned is RN($c \top d$). Interestingly enough, the IEEE 754-2019 standard also defines two other round-to-nearest functions: round-to-nearest *ties-to-away* (mainly needed in decimal arithmetic, for financial applications), and round-to-nearest *ties-to-zero* (whose major purpose is to help implementing fast reproducible summation [27]). The paper proofs of the double-word algorithms in [16] assume round-to-nearest, ties-to-even. It might be interesting to see if the proofs remain valid (possibly with minor modifications) with the other round-to-nearest functions: we consider this later on in this paper. We will even see that changing the tie-breaking rule can change the error bound of an algorithm (see Theorem 2.6).

Finally, two quantities are commonly used to express errors in floating-point arithmetic:

- the number ulp(x), for $x \neq 0$ is $2^{\lfloor \log_2 |x| \rfloor - p + 1}$. Roughly speaking, ulp($x$) is the distance between two consecutive FP numbers in the neighborhood of x. If a scalar function f is *correctly rounded*—i.e., if for any x we always return RN $(f(x))$—, then the absolute error when computing $f(x)$ is bounded by $\frac{1}{2}\text{ulp}\left(f(x)\right)$.
- $u = 2^{-p}$ denotes the roundoff error unit. If a scalar function is correctly rounded then it is computed with relative error less than $u/(1+u)$ [15], which is very slightly less than u. A floating-point number between 2^k and 2^{k+1} is a multiple of $u \cdot 2^{k+1}$.

The bounds given in this paper and in [16] are given as functions of u. If the bound for some algorithm is $B(u)$ and if we are able to show that there exist some inputs parametrized by u for which the relative error $E(u)$ satisfies $E(u)/B(u) \to 1$ as $u \to 0$, we will say that the bound $B(u)$ is *asymptotically optimal*.

2 Algorithms for double-word arithmetic

Let us now give a formal definition of a double-word number.

Definition 1. *[16]. A* double-word *(DW) number x is the unevaluated sum $x_h + x_\ell$ of two floating-point numbers x_h and x_ℓ such that $x_h = \text{RN}(x)$.*

In other words, a DW number is a real number equal to the sum of its rounded to the nearest FP number x_h and an error term x_ℓ that is also a floating-point number.

In [16], algorithms are given for adding, multiplying and dividing DW numbers, adding a FP number to a DW number, and multiplying or dividing a DW number by a FP number. All these algorithms return DW numbers. They are given with an error bound and a proof. They all use the following three basic building blocks, named "error-free transforms" by Ogita, Rump and Oishi [26], that are well-known in the FP literature, and that return DW numbers equal to the sum or product of two input FP numbers.

2.1 The basic building blocks: "error-free transforms"

The Fast2Sum algorithm (Algorithm 1) is due to Dekker [8].

Algorithm 1 – Fast2Sum(a, b). The Fast2Sum algorithm [8].

$s \leftarrow \text{RN}(a + b)$
$z \leftarrow \text{RN}(s - a)$
$t \leftarrow \text{RN}(b - z)$

If a and b are FP numbers that can be written $M_a \cdot e_a$ and $M_b \cdot e_b$, with $|M_a|, |M_b| \leq 2^p - 1$ and $e_a \geq e_b$, then the result (s, t) returned by Algorithm 1 satisfies $s + t = a + b$. Hence, t is the error of the FP addition $s \leftarrow \text{RN}(a + b)$. In practice, the above given condition on e_a and e_b may be hard to check. However, if $|a| \geq |b|$ then that condition is satisfied. One can avoid having to perform a comparison of $|a|$ and $|b|$ by using the more complex Algorithm 2 below.

Algorithm 2 – 2Sum(a, b). The 2Sum algorithm [22, 19].

$s \leftarrow \text{RN}(a + b)$
$a' \leftarrow \text{RN}(s - b)$
$b' \leftarrow \text{RN}(s - a')$
$\delta_a \leftarrow \text{RN}(a - a')$
$\delta_b \leftarrow \text{RN}(b - b')$
$t \leftarrow \text{RN}(\delta_a + \delta_b)$

The result (s, t) returned by Algorithm 2 satisfies $s + t = a + b$ for all FP inputs a and b.

Finally, the following algorithm allows one to compute the error of a FP multiplication.

Algorithm 3 – 2Prod(a, b)**.** The 2Prod algorithm (called Fast2Mult in [18, 25, 24]). It requires the availability of a fused multiply-add (FMA) instruction for computing $\text{RN}(a \cdot b - \pi)$.

$\pi \leftarrow \text{RN}(a \cdot b)$
$\rho \leftarrow \text{RN}(a \cdot b - \pi)$

The result (π, ρ) returned by Algorithm 3 satisfies $\pi + \rho = a \times b$ for all FP inputs a and b. That algorithm requires the availability of an FMA (fused multiply-add) instruction.

2.2 An example

Algorithm 4 below (which was Algorithm 6 in [16]) was presented in [21, 20]. It approximates the sum of two DW numbers by a DW number, with a relative error less than $3u^2 + 13u^3$ as soon as $p \geq 6$, as shown in [16].

Algorithm 4 – AccurateDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$. Calculation of $(x_h, x_\ell) + (y_h, y_\ell)$ in binary, precision-p, floating-point arithmetic.

1: $(s_h, s_\ell) \leftarrow \mathsf{2Sum}(x_h, y_h)$
2: $(t_h, t_\ell) \leftarrow \mathsf{2Sum}(x_\ell, y_\ell)$
3: $c \leftarrow \mathsf{RN}(s_\ell + t_h)$
4: $(v_h, v_\ell) \leftarrow \mathsf{Fast2Sum}(s_h, c)$
5: $w \leftarrow \mathsf{RN}(t_\ell + v_\ell)$
6: $(z_h, z_\ell) \leftarrow \mathsf{Fast2Sum}(v_h, w)$
7: **return** (z_h, z_ℓ)

Although the proved error bound for Algorithm 4 is close to $3u^2$, we were not aware before this study, when the RN function is *round-to-nearest ties to even*, of examples for which the relative error is larger than $2.25u^2$. The authors of [16] build an example, for $p = 53$, for which the relative error is $2.24999999999999956\cdots \times 2^{-106}$. Failing to obtain larger errors in extensive simulations, we were inclined to conjecture that the actual error bound for Algorithm 4 is $2.25u^2$, but such conjectures are dangerous: the authors of [21, 20] already claimed that the bound, for $p = 53$, was 2×2^{-106} (i.e., $2u^2$).

Indeed, the error can be larger, more precisely, we have the following.

Property 2.1. *Assuming that $p \geq 3$ and* RN *is round-to-nearest* ties-to-even*, the error bound $3u^2/(1-4u) = 3u^2 + 12u^3 + 48u^4 + \cdots$ for Algorithm 4, given by Joldes et al. in [16], is asymptotically optimal.*

Proof. Consider

$$\begin{aligned} x_h &= 1 \\ x_\ell &= u - u^2 \\ y_h &= -\tfrac{1}{2} + \tfrac{u}{2} \\ y_\ell &= -\tfrac{u^2}{2} + u^3. \end{aligned}$$

We successively obtain:

$$s_h = \mathsf{RN}(x_h + y_h) = \frac{1}{2}$$

(thanks to the ties-to-even tie-breaking rule)

$$s_\ell = x_h + y_h - s_h = \frac{u}{2},$$

$$t_h = \mathrm{RN}(x_\ell + y_\ell) = u - u^2,$$

$$t_\ell = x_\ell + y_\ell - t_h = -\frac{u^2}{2} + u^3,$$

$$c = \mathrm{RN}(s_\ell + t_h) = \frac{3u}{2},$$

(thanks to the ties-to-even tie-breaking rule, and because $p \geq 3$ implies that $3u/2$ is an "even" FP number)

$$v_h = \mathrm{RN}(s_h + c) = \frac{1}{2} + 2u,$$

$$v_\ell = s_h + c - v_h = -\frac{u}{2},$$

and finally,

$$z = z_h + z_\ell = v_h + w = \frac{1}{2} + \frac{3u}{2},$$

whereas the exact result is

$$x + y = (x_h + x_\ell) + (y_h + y_\ell) = \frac{1}{2} + \frac{3u}{2} - \frac{3u^2}{2} + u^3.$$

Therefore the relative error is

$$\frac{|(x+y) - (z_h + z_\ell)|}{x+y} = \frac{3u^2 - 2u^3}{1 + 3u - 3u^2 + 2u^3} = 3u^2 - 11u^3 + 42u^4 + \cdots$$

□

For instance, if $p = 53$ (double-precision arithmetic), the generic example used in the proof leads to a relative error equal to

$$2.9999999999999877875 \cdots \times u^2.$$

That example has an interesting history. Being puzzled by the gap between the bound shown in [16] and the largest observed error, we tried to show a smaller bound. We almost succeeded: the only case for which we could not prove that the relative error is less than $2.5u^2$ was when c is of the form $3 \cdot 2^k$. This led us to focus only on cases for which c has that form, and finally to build the example used in the proof. Obtaining such an example by random testing is hopeless. Figure 1 gives the repartition of the observed relative errors for a random sample of 4000000 input values: it is almost impossible to observe errors larger than around $2.5u^2$.

2.3 The various algorithms of [16] we have formally proven

We have formally proven the 15 algorithms presented in [16]. All these algorithms return DW numbers. All error bounds claimed in [16] are correct (but we have found improvements for three of them). Table 1 summarizes the obtained results.

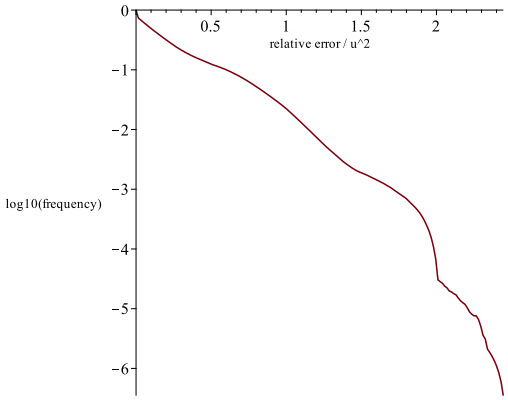

Figure 1: Radix-10 logarithm of the frequency of cases for which the relative error of Algorithm 4 is larger than λu^2 as a function of λ, for a random sample of 4000000 input values.

Table 1: Summary of the results presented in [16], and now formally proven, and of our own results. For each algorithm, we give the bound given in [16], the bound we have formally proven, and the largest relative error experimentally observed. The bold face indicates improvements over the bounds given in [16]. Unless stated otherwise, the largest errors observed in experiments are for RN being round-to-nearest *ties-to-even*.

Operation	Name of the Algorithm in [16]	Bound given in [16]	Bound formally proven	Largest relative error built or observed in experiments
DW + FP	DWPlusFP	$2u^2$	$2u^2$	$2u^2 - 6u^3$
DW + DW	SloppyDWPlusDW	N/A	N/A	1
	AccurateDWPlusDW	$3u^2 + 13u^3$	$3u^2 + 13u^3$	$3u^2 - 11u^3$ $+\mathcal{O}(u^4)$
DW × FP	DWTimesFP1	$\frac{3}{2}u^2 + 4u^3$	$\frac{3}{2}u^2 + 4u^3$	$1.5u^2$
	DWTimesFP2	$3u^2$	$3u^2$	$2.517u^2$
	DWTimesFP3	$2u^2$	$2u^2$	$1.984u^2$
DW × DW	DWTimesDW1	$7u^2$	$\mathbf{5u^2}$ (ties to even)	$4.985u^2$ (ties to even)
			$\mathbf{5.5u^2}$ (general)	$5.4907u^2$ (ties to 0)
	DWTimesDW2	$6u^2$	$\mathbf{5u^2}$	$4.9433u^2$
	DWTimesDW3	$5u^2$	$\mathbf{4u^2}$	$3.997u^2$
DW ÷ FP	DWDivFP1	$3.5u^2$	$3.5u^2$	$2.95u^2$
	DWDivFP2	$3.5u^2$	$3.5u^2$	$2.95u^2$
	DWDivFP3	$3u^2$	$3u^2$	$2.95u^2$
DW ÷ DW	DWDivDW1	$15u^2 + 56u^3$	$15u^2 + 56u^3$	$8.465u^2$
	DWDivDW2	$15u^2 + 56u^3$	$15u^2 + 56u^3$	$8.465u^2$
	DWDivDW3	$9.8u^2$	$9.8u^2$	$5.922u^2$

2.4 The major problems encountered

In the following, we focus on the two proofs for which we have encountered a major problem (one of them was in an early version of [16], and was corrected *before* final publication, so it does not appear in the published paper). The first of these two proofs is the proof of Algorithm **DWPlusFP** (Algorithm 5 in this paper, Algorithm 4 in [16]), which evaluates the sum of a DW number and a FP number.

Algorithm 5 – DWPlusFP(x_h, x_ℓ, y) (Algorithm 4 in [16]). Computes $(x_h, x_\ell) + y$. This algorithm is implemented in the QD library [12]. The number $x = (x_h, x_\ell)$ is a DW number.

1: $(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y)$
2: $v \leftarrow \text{RN}(x_\ell + s_\ell)$
3: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(s_h, v)$
4: **return** (z_h, z_ℓ)

The correctness and error bound of Algorithm **DWPlusFP** are claimed in [16] by the following Theorem.

Theorem 2.2 (Theorem 2.2 in [16])**.** *The relative error*

$$\left| \frac{(z_h + z_\ell) - (x + y)}{x + y} \right|$$

of Algorithm 5 (DWPlusFP) is bounded by $2 \cdot u^2$.

As said above, the error in the proof of Theorem 2.2 was in an early version of [16], so it does not appear in the published version. Correcting that error just required a slight modification of the existing proof.

The second proof is the common proof (in fact, the proofs slightly differ but they share the same major steps) of Algorithms **DWTimesDW1**, **DWTimesDW2** and **DWTimesDW3** (Algorithms 6, 7, and 8 in this paper, Algorithms 10, 11 and 12 in [16]) which evaluate the product of two DW numbers.

Algorithm 6 – DWTimesDW1$(x_h, x_\ell, y_h, y_\ell)$ (Algorithm 10 in [16]). Computes $(x_h, x_\ell) \times (y_h, y_\ell)$. This algorithm is implemented in the QD library [12].

1: $(c_h, c_{\ell 1}) \leftarrow \text{2Prod}(x_h, y_h)$
2: $t_{\ell 1} \leftarrow \text{RN}(x_h \cdot y_\ell)$
3: $t_{\ell 2} \leftarrow \text{RN}(x_\ell \cdot y_h)$
4: $c_{\ell 2} \leftarrow \text{RN}(t_{\ell 1} + t_{\ell 2})$
5: $c_{\ell 3} \leftarrow \text{RN}(c_{\ell 1} + c_{\ell 2})$
6: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(c_h, c_{\ell 3})$
7: **return** (z_h, z_ℓ)

Algorithm 7 – DWTimesDW2$(x_h, x_\ell, y_h, y_\ell)$ (Algorithm 11 in [16]). Algorithm for computing $(x_h, x_\ell) \times (y_h, y_\ell)$ in binary, precision-p, floating-point arithmetic, assuming an FMA instruction is available.

1: $(c_h, c_{\ell 1}) \leftarrow$ 2Prod(x_h, y_h)
2: $t_\ell \leftarrow$ RN$(x_h \cdot y_\ell)$
3: $c_{\ell 2} \leftarrow$ RN$(t_\ell + x_\ell y_h)$
4: $c_{\ell 3} \leftarrow$ RN$(c_{\ell 1} + c_{\ell 2})$
5: $(z_h, z_\ell) \leftarrow$ Fast2Sum$(c_h, c_{\ell 3})$
6: **return** (z_h, z_ℓ)

Algorithm 8 – DWTimesDW3$(x_h, x_\ell, y_h, y_\ell)$ (Algorithm 12 in [16]). Algorithm for computing $(x_h, x_\ell) \times (y_h, y_\ell)$ in binary, precision-p, floating-point arithmetic, assuming an FMA instruction is available.

1: $(c_h, c_{\ell 1}) \leftarrow$ 2Prod(x_h, y_h)
2: $t_{\ell 0} \leftarrow$ RN$(x_\ell \cdot y_\ell)$
3: $t_{\ell 1} \leftarrow$ RN$(x_h \cdot y_\ell + t_{\ell 0})$
4: $c_{\ell 2} \leftarrow$ RN$(t_{\ell 1} + x_\ell \cdot y_h)$
5: $c_{\ell 3} \leftarrow$ RN$(c_{\ell 1} + c_{\ell 2})$
6: $(z_h, z_\ell) \leftarrow$ Fast2Sum$(c_h, c_{\ell 3})$
7: **return** (z_h, z_ℓ)

The correctness and error bound of Algorithms **DWTimesDW1**, **DWTimesDW2** and **DWTimesDW3** are claimed in [16] by the following Theorems.

Theorem 2.3 (Theorem 5.1 in [16]). *If $p \geq 4$, the relative error of Algorithm 6 (DWTimesDW1) is less than or equal to $7u^2/(1+u)^2 < 7u^2$.*

Theorem 2.4 (Theorem 5.3 in [16]). *If $p \geq 5$, the relative error of Algorithm 7 (DWTimesDW2) is less than or equal to $(6u^2 + \frac{1}{2}u^3)/(1+u)^2 < 6u^2$.*

Theorem 2.5 (Theorem 5.4 in [16]). *If $p \geq 4$, the relative error of Algorithm 8 (DWTimesDW3) is less than or equal to $(5u^2 + \frac{1}{2}u^3)/(1+u)^2 < 5u^2$.*

It turns out that these three theorems are correct, despite a flaw in their original proof (as said above the proof is essentially the same, with small variants, for the three theorems). Our work on the formal proof of these theorems led us to find and to remove the flaw. It also led us to improvements of the bounds given by these Theorems and, interestingly enough, to see that the tie-breaking rule of the round-to-nearest function RN can have a significant influence on the bound. More precisely, we now have,

Theorem 2.6 (New error bound for Algorithm DWTimesDW1). *If $p \geq 6$, the relative error of Algorithm 6 (DWTimesDW1) is less than or equal to*

$$\frac{\frac{11}{2}u^2}{(1+u)^2} < 5.5u^2. \tag{2}$$

Furthermore, if the rounding mode RN *is round-to-nearest* ties-to-even *(which is the default in IEEE 754 arithmetic) then we have the better bound*

$$\frac{5u^2}{(1+u)^2} < 5u^2. \tag{3}$$

Note that the proof assistant does not warn its user that better bounds can be obtained. However, it helped us to improve the bounds by allowing us to focus on the modification of critical parts of the proofs only. The formal proof is then run again, with the modifications, to quickly check that the other parts of the proof remain valid with the modification.

Theorem 2.7 (New error bound for Algorithm DWTimesDW2)**.** *If $p \geq 5$, the relative error of Algorithm 7 (DWTimesDW2) is less than or equal to*

$$\frac{5u^2}{(1+u)^2} < 5u^2. \tag{4}$$

Theorem 2.8 (New error bound for Algorithm DWTimesDW3)**.** *If $p \geq 5$, the relative error of Algorithm 8 (DWTimesDW3) is less than or equal to*

$$\frac{4u^2 + \frac{1}{2}u^3}{(1+u)^2} < 4u^2.$$

The proof of Theorems 2.3, 2.4, and 2.5 in [16] use the following Lemma, which turned out to be wrong (it suffices to try $a = 1$ and $b = 2$).

Incorrect Lemma 2.9 (Lemma 5.2 in [16] (wrong))**.** *Let a and b be two positive real numbers. If $ab \leq 2$, $a \geq 1$ and $b \geq 1$, then $a + b \leq 2\sqrt{2}$.*

For Theorems 2.3, 2.4, and 2.5 a significant modification was necessary. Let us now detail the proof of Theorem 2.5, to understand where the problem was. We will also prove Theorem 2.8 at the same time. The proofs of Theorems 2.6, and 2.7, more involved, will be given afterwards. Note that, although we have no proof of asymptotic optimality, the bounds given by Theorems 2.6, 2.7, and 2.8 are very tight and therefore cannot be significantly improved: for each of the corresponding algorithms we have input examples for which the attained error is very close to the bound.

First, we will need another lemma, to replace Lemma 2.9.

Lemma 2.10. *Let a and b be two floating-point numbers satisfying $1 \leq a \leq 2 - 2u$ and $1 \leq b \leq 2 - 2u$. If $ab \leq 2$ then $a + b \leq 3 - 2u$.*

Proof. First, let us show that $a + b < 3$. We have $a \leq 2/b$, therefore $a + b \leq 2/b + b$. The number $2/b+b-3$ has the sign of b^2-3b+2: it is < 0 for $1 < b < 2$. Therefore, if $b \neq 1$, $a+b < 3$. The case $b = 1$ is easily dealt with: $a < 2$ implies $a+b = a+1 < 3$.

Since a and b are floating-point numbers larger than 1 they are multiple of $2u$. Therefore $a+b$ is a multiple of $2u$ strictly less than 3: this implies $a+b \leq 3-2u$. □

Now let us give a proof of Theorems 2.5 and 2.8. In all our paper proofs, we explicitly or implicitly use the following properties:

1. if $|x| \leq 2^k$ then $|x - \mathrm{RN}(x)| \leq u \cdot 2^{k-1}$;
2. $x \leq y \Rightarrow \mathrm{RN}(x) \leq \mathrm{RN}(y)$;
3. $\mathrm{RN}(x \cdot 2^k) = 2^k \cdot \mathrm{RN}(x)$;
4. $\mathrm{RN}(-x) = -\mathrm{RN}(x)$;
5. if $t \geq 0$ then $t(1-u) \leq \mathrm{RN}(t) \leq t(1+u)$.

2.5 Proof of Theorems 2.5 and 2.8

Without loss of generality, we assume that $1 \leq x_h \leq 2 - 2u$ and $1 \leq y_h \leq 2 - 2u$. This implies $|x_\ell| \leq u$ and $|y_\ell| \leq u$. We have $x_h y_h < 4$, and

$$c_h + c_{\ell 1} = x_h y_h,$$

with $|c_{\ell 1}| \leq 2u$. We also have $|x_\ell y_\ell| \leq u^2$, therefore, since RN is an increasing function, and since u^2 is a floating-point number (which implies $\mathrm{RN}(u^2) = u^2$), we have

$$|t_{\ell 0}| \leq u^2, \tag{5}$$

and

$$\begin{array}{l} t_{\ell 0} = x_\ell y_\ell + \epsilon_0, \\ \text{with } |\epsilon_0| \leq u^3/2. \end{array} \tag{6}$$

Similarly, from

$$|x_h y_\ell + t_{\ell 0}| \leq 2u - u^2,$$

we deduce

$$|t_{\ell 1}| \leq 2u,$$

and

$$\begin{array}{l} t_{\ell 1} = x_h y_\ell + t_{\ell 0} + \epsilon_1, \\ \text{with } |\epsilon_1| \leq u^2. \end{array} \tag{7}$$

Since $|x_\ell y_h| \leq 2u - 2u^2$, we have $|t_{\ell 1} + x_\ell y_h| \leq 4u - 2u^2$, so that

$$|c_{\ell 2}| \leq 4u$$

and

$$\begin{array}{l} c_{\ell 2} = t_{\ell 1} + x_\ell y_h + \epsilon_2, \\ \text{with } |\epsilon_2| \leq 2u^2. \end{array} \tag{8}$$

The number $|c_{\ell 1} + c_{\ell 2}|$ is less than or equal to $6u$, and $6u$ is a floating-point number (as soon as $p \geq 2$), therefore

$$|c_{\ell 3}| \leq 6u,$$

and

$$c_{\ell 3} = c_{\ell 1} + c_{\ell 2} + \epsilon_3,$$

with $|\epsilon_3| \leq 4u^2$. Since $|c_h| \geq 1$ and $|c_{\ell 3}| \leq 6u$, we can use Algorithm Fast2Sum at Line 6 of the algorithm, as soon as $6u \leq 1$ (i.e., as soon as $p \geq 3$). Therefore, $z_h + z_\ell = c_h + c_{\ell 3}$ and (z_h, z_ℓ) is a DW-number.

We finally obtain

$$z_h + z_\ell = xy + \epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3,$$

and the sum of the error terms satisfies

$$|\epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3| \leq 7u^2 + \frac{u^3}{2}. \tag{9}$$

Now we need to consider three cases:

- if $x_h y_h > 2$ then $xy \geq (x_h(1-u)) \cdot (y_h(1-u)) > 2(1-u)^2$, therefore, from (9), the relative error
$$\frac{|z_h + z_\ell - xy|}{|xy|}$$
is bounded by
$$\frac{7u^2 + \frac{u^3}{2}}{2(1-u)^2}, \tag{10}$$
and we can check that the bound (10) is less than the bound of Theorem 2.5 as soon as $u \leq 1/16$ (i.e., as soon as $p \geq 4$), and less than the bound of Theorem 2.8 as soon as $u \leq 1/32$ (i.e., as soon as $p \geq 5$).

- if $x_h = 1$ or $y_h = 1$, we obtain $c_{\ell 1} = 0$, so that $c_{\ell 3} = c_{\ell 2}$ and $\epsilon_3 = 0$. Hence $|\epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3| \leq 3u^2 + u^3/2$, and the relative error is bounded by
$$\frac{3u^2 + \frac{u^3}{2}}{(1-u)^2},$$
which is less than the bound of Theorem 2.5 as soon as $u \leq 1/8$ (i.e., as soon as $p \geq 3$) and less than the bound of Theorem 2.8 as soon as $u \leq 1/16$ (i.e., as soon as $p \geq 4$).

- if $x_h y_h \leq 2$, with $x_h > 1$ and $y_h > 1$, then some bounds can be improved, as we are going to see.

  First, $x_h y_h \leq 2$ implies
  $$|c_{\ell 1}| \leq u. \tag{11}$$
  In [16], Lemma 2.9 was invoked to deduce a bound on $x_h + y_h$ from the bound on $x_h y_h$. The goal was to deduce from that bound on $x_h + y_h$ a new bound on $|c_{\ell 1} + c_{\ell 2}|$ small enough to guarantee that $|\epsilon_3|$ becomes less than or equal to $2u^2$. Unfortunately, Lemma 2.9 is wrong. As we are going to see, the bound given by Lemma 2.10 suffices.

  From $x_h y_h \leq 2$ and Lemma 2.10, we obtain $x_h + y_h \leq 3 - 2u$. Hence, using the bounds on $|x_\ell|$ and $|y_\ell|$,
  $$|x_h y_\ell + y_h x_\ell| \leq 3u - 2u^2,$$

therefore, using (5),

$$|x_h y_\ell + y_h x_\ell + t_{\ell 0}| \leq 3u - u^2.$$

Therefore, using (7),

$$|y_h x_\ell + t_{\ell 1}| \leq 3u,$$

from which we deduce,

$$|c_{\ell 2}| \leq \mathrm{RN}\,(3u) = 3u. \tag{12}$$

Combining (11) and (12), we obtain

$$|c_{\ell 1} + c_{\ell 2}| \leq 4u, \tag{13}$$

so that $|\epsilon_3| = |\mathrm{RN}(c_{\ell 1} + c_{\ell 2}) - (c_{\ell 1} + c_{\ell 2})|$ is now less than or equal to $2u^2$. An immediate consequence is that the sum of the error terms

$$|\epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3|$$

is now bounded by $5u^2 + u^3/2$. Since $x_h > 1$ and $y_h > 1$, we have $x_h \geq 1 + 2u$ and $x \geq 1 + u$, and, similarly, $y \geq 1 + u$. The product xy is therefore lower-bounded by $(1 + u)^2$, so that the relative error is bounded by

$$\frac{5u^2 + \frac{u^3}{2}}{(1+u)^2},$$

which is the bound given by Theorem 2.5. Now, we can improve it by raising the following remarks:

- Since x_h and y_h are both multiple of $2u$, their product is a multiple of $4u^2 = 2^{2-2p}$. Since c_h (which is a floating-point number larger than 1) is a multiple of $2u = 2^{1-p}$, the number $c_{\ell 1} = x_h y_h - c_h$ is a multiple of $4u^2$;
- if $|c_{\ell 2}| < 2u$ (which implies $|t_{\ell 1} + x_\ell y_h| \leq 2u - u^2$), then $|\epsilon_2|$ is bounded by u^2;
- if $|c_{\ell 2}| \geq 2u$ then $c_{\ell 2}$ is a multiple of $4u^2$. Therefore $c_{\ell 1} + c_{\ell 2}$ is a multiple of $4u^2$. This and (13) imply that it is a floating-point number. Therefore $\mathrm{RN}(c_{\ell 1} + c_{\ell 2}) = c_{\ell 1} + c_{\ell 2}$, so that $\epsilon_3 = 0$.

Therefore, either $|\epsilon_2| \leq u^2$ or $\epsilon_3 = 0$. An immediate consequence is that $|\epsilon_0 + \epsilon_1 + \epsilon_2 + \epsilon_3| \leq 4u^2 + u^3/2$. Using (as previously) the fact that $xy > (1+u)^2$, we obtain the relative error bound of Theorem 2.8.

□

Note that the new bound $4u^2$ given by Theorem 2.8 is very tight: if $p = 53$, for

$$\begin{aligned} x_h &= 2251799825991851/2^{51}, \\ x_\ell &= 9007199203085987/2^{106}, \\ y_h &= 4503599627471459/2^{52}, \\ y_\ell &= 4503599627284651/2^{105}, \end{aligned}$$

the relative error obtained with Algorithm 8 is $3.997 \cdots u^2$.

2.6 Proof of Theorem 2.6

Without loss of generality, we assume that $1 \leq x_h \leq 2-2u$ and $1 \leq y_h \leq 2-2u$. This implies $|x_\ell| \leq u$ and $|y_\ell| \leq u$. We have $1 \leq x_h y_h < 4$, and

$$c_h + c_{\ell 1} = x_h y_h,$$

with $|c_{\ell 1}| \leq 2u$.

Since x_h and y_h are both multiple of $2u$, their product is a multiple of $4u^2 = 2^{2-2p}$. Since c_h (which is a floating-point number larger than or equal to 1) is a multiple of $2u = 2^{1-p}$, the number $c_{\ell 1} = x_h y_h - c_h$ is a multiple of $4u^2$.

Since $|x_h y_\ell| \leq 2u - 2u^2$, we have $|t_{\ell 1}| \leq \mathrm{RN}(2u - 2u^2) = 2u - 2u^2$, and

$$t_{\ell 1} = x_h y_\ell + \epsilon_1, \ with \ |\epsilon_1| \leq u^2.$$

Similarly, we have $|t_{\ell 2}| \leq 2u - 2u^2$, and

$$t_{\ell 2} = x_\ell y_h + \epsilon_2, \ with \ |\epsilon_2| \leq u^2.$$

This gives $|t_{\ell 1} + t_{\ell 2}| \leq 4u - 4u^2$, so that $|c_{\ell 2}| \leq \mathrm{RN}(4u - 4u^2) = 4u - 4u^2$, and

$$c_{\ell 2} = t_{\ell 1} + t_{\ell 2} + \epsilon_3, \ with \ |\epsilon_3| \leq 2u^2.$$

Therefore, $|c_{\ell 1} + c_{\ell 2}| \leq 6u - 4u^2$, which gives $|c_{\ell 3}| \leq 6u$, and

$$c_{\ell 3} = c_{\ell 1} + c_{\ell 2} + \epsilon_4, \ with \ |\epsilon_4| \leq 4u^2.$$

Since $c_h \geq 1$ and $|c_{\ell 3}| \leq 6u$, as soon as $6u \leq 1$ (which holds as soon as $p \geq 3$), we can use the Fast2Sum algorithm, and

$$z_h + z_\ell = c_h + c_{\ell 3}.$$

Defining

$$\eta = (z_h + z_\ell) - xy, \tag{14}$$

we obtain

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2 + u^2 + 2u^2 + 4u^2 = 9u^2. \end{array} \tag{15}$$

Let us now give a case-by-case analysis, for obtaining a bound on the relative error $|\eta|/(xy)$ and, in some cases, improving the bound (15).

2.6.1 If $x_h y_h \geq 2$

In that case $xy \geq x_h(1-u) \cdot y_h(1-u) > 2(1-u)^2$. So, that, using (15), the relative error is bounded by

$$\frac{9u^2}{2(1-u)^2},$$

which is less than the bound (3) as soon as $u \leq 1/64$ (i.e., as soon as $p \geq 6$).

2.6.2 If $x_h y_h < 2$

We can observe that $|c_{\ell 1}| \leq u$, and that Lemma 2.10 implies $|t_{\ell 1} + t_{\ell 2}| \leq x_h u + y_h u \leq 3u - 2u^2$, so that $|c_{\ell 2}| \leq 3u$. Therefore, $|c_{\ell 1} + c_{\ell 2}| \leq 4u$, so that $|\epsilon_4| \leq 2u^2$. This first improvement is at the origin of the "$7u^2$" in Theorem 2.3. However, we can improve the bound further by considering the following sub-cases:

- if $|c_{\ell 2}| \geq 2u$ then $c_{\ell 2}$ is a multiple of $4u^2$, therefore $c_{\ell 1} + c_{\ell 2}$ is a multiple of $4u^2$. Since it has an absolute value less than or equal to $4u$, this means that it is a floating-point number. An immediate consequence is $\mathrm{RN}(c_{\ell 1} + c_{\ell 2}) = c_{\ell 1} + c_{\ell 2}$ and $\epsilon_4 = 0$. This gives

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2 + u^2 + 2u^2 + 0 = 5u^2. \end{array}$$

- if $|c_{\ell 2}| < 2u$, which implies $|\epsilon_3| \leq u^2$ then
  - if $|t_{\ell 1}| < u/2$ then $|\epsilon_1| \leq u^2/4$ and $|y_\ell| < u/2$, so that $|x_\ell y_\ell| < u^2/2$, so that

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2/2 + u^2/4 + u^2 + u^2 + 2u^2 = 4.75u^2; \end{array}$$

  - if $|t_{\ell 2}| < u/2$ then $|\epsilon_2| \leq u^2/4$ and $|x_\ell| < u/2$, so that $|x_\ell y_\ell| < u^2/2$, which gives the same result;
  - if $|t_{\ell 1}| \geq u/2$ and $|t_{\ell 2}| \geq u/2$ then let us first note that $t_{\ell 1}$ and $t_{\ell 2}$ (and therefore $t_{\ell 1} + t_{\ell 2}$) are multiples of u^2.
    - if $|t_{\ell 1}| < u$ or $|t_{\ell 2}| < u$ then either $|\epsilon_1| \leq u^2/2$ or $|\epsilon_2| \leq u^2/2$, and the number $|t_{\ell 1} + t_{\ell 2}|$ is a multiple of u^2 less than $2u$ (otherwise $|c_{\ell 2}|$ would be $\geq 2u$). Therefore:
      - if $|t_{\ell 1} + t_{\ell 2}|$ is less than u or if it is an *even* multiple of u^2 then it is a floating-point number, which implies $\epsilon_3 = 0$, so that

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2/2 + u^2 + 0 + 2u^2 = 4.5u^2. \end{array}$$

      - if $|t_{\ell 1} + t_{\ell 2}|$ is larger than u and is an *odd* multiple of u^2, then it is halfway between two consecutive FP numbers, so that $|\epsilon_3| = u^2$ exactly and

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2/2 + u^2 + u^2 + 2u^2 = 5.5u^2. \end{array}$$

        Furthermore, if RN is round-to-nearest *ties to even* then $t_{\ell 1} + t_{\ell 2}$ is rounded to an *even* multiple of $\mathrm{ulp}(t_{\ell 1} + t_{\ell 2}) = 2u^2$. This implies that $c_{\ell 2}$ is a multiple of $4u^2$. Hence, $c_{\ell 1} + c_{\ell 2}$ is a multiple of $4u^2$ of absolute value less than $4u$. It is therefore a floating-point number, so that $\epsilon_4 = 0$, and

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2/2 + u^2 + u^2 + 0 = 3.5u^2. \end{array}$$

* finally, if $|t_{\ell 1}| \geq u$ and $|t_{\ell 2}| \geq u$, then $t_{\ell 1}$ and $t_{\ell 2}$ are multiple of $2u^2$, so that their sum is a multiple of $2u^2$ of absolute value less than $2u$. It is therefore a floating-point number, so that $\epsilon_3 = 0$, which implies

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_2| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2 + u^2 + 0 + 2u^2 = 5u^2. \end{array}$$

To sum up, we have shown that when $x_h y_h < 2$, the absolute error bound η is less than $\frac{11}{2}u^2$ in the general case, and less than $5u^2$ with the additional assumption that RN is "round-to-nearest *ties to even*". Let us now deduce relative error bounds from that. We can notice that if $x_h = 1$ or $y_h = 1$ then $c_{\ell 1} = 0$, so that $\epsilon_4 = 0$, and either $\epsilon_1 = 0$ or $\epsilon_2 = 0$. Thus $|\eta| \leq u^2 + 0 + u^2 + 2u^2 + 0 = 4u^2$ and the relative error bound is less than

$$\frac{4u^2}{\left(1 - \frac{u}{2}\right)^2},$$

which is less than the bound (3) as soon as $u \leq 1/16$ (i.e., as soon as $p \geq 4$). Now, if x_h and y_h are different from 1, they are larger than or equal to $1+2u$, so that x and y are larger than or equal to $1 + u$, and the relative error is bounded by

$$\frac{\frac{11}{2}u^2}{(1+u)^2}$$

in the general case, and by

$$\frac{5u^2}{(1+u)^2}$$

with the additional assumption that RN is "round-to-nearest *ties to even*".

□

It is worth mentioning that the bounds given by Theorem 2.6 are tight:

- With RN being " round-to-nearest *ties to even*" and $p = 24$, relative error $4.98575990u^2$ is reached with $x_h = 2097221/2^{21}$, $x_\ell = 16777007/2^{48}$, $y_h = 131077/2^{17}$, and $y_\ell = 16777037/2^{48}$;
- With RN being " round-to-nearest *ties to zero*" and $p = 53$, relative error $5.490790833u^2$ is reached if we choose $x_h = 4503599652744837/2^{52}$, $x_\ell = 9007199254569309/2^{106}$, $y_h = 2251799817602973/2^{51}$, and $y_\ell = 4503599582208165/2^{105}$.

2.7 Proof of Theorem 2.7

Without loss of generality, we assume that $1 \leq x_h \leq 2 - 2u$ and $1 \leq y_h \leq 2 - 2u$. This implies $|x_\ell| \leq u$ and $|y_\ell| \leq u$. We have $1 \leq x_h y_h < 4$, and

$$c_h + c_{\ell 1} = x_h y_h,$$

with $|c_{\ell 1}| \leq 2u$.

Since x_h and y_h are both multiple of $2u$, their product is a multiple of $4u^2 = 2^{2-2p}$. Since c_h (which is a floating-point number larger than or equal to 1) is a multiple of $2u = 2^{1-p}$, the number $c_{\ell 1} = x_h y_h - c_h$ is a multiple of $4u^2$.

Since $|x_h y_\ell| \leq 2u - 2u^2$, we have $|t_\ell| \leq \mathrm{RN}(2u - 2u^2) = 2u - 2u^2$, and

$$t_\ell = x_h y_\ell + \epsilon_1, \ with \ |\epsilon_1| \leq u^2.$$

This gives $|t_\ell + x_\ell y_h| \leq 4u - 4u^2$, so that $|c_{\ell 2}| \leq \mathrm{RN}(4u - 4u^2) = 4u - 4u^2$, and

$$c_{\ell 2} = t_\ell + x_\ell y_h + \epsilon_3, \ with \ |\epsilon_3| \leq 2u^2.$$

Therefore, $|c_{\ell 1} + c_{\ell 2}| \leq 6u - 4u^2$, which gives $|c_{\ell 3}| \leq 6u$, and

$$c_{\ell 3} = c_{\ell 1} + c_{\ell 2} + \epsilon_4, \ with \ |\epsilon_4| \leq 4u^2.$$

Since $c_h \geq 1$ and $|c_{\ell 3}| \leq 6u$, as soon as $6u \leq 1$ (which holds as soon as $p \geq 3$), we can use the Fast2Sum algorithm, and

$$z_h + z_\ell = c_h + c_{\ell 3}.$$

Defining

$$\eta = (z_h + z_\ell) - xy, \tag{16}$$

we obtain

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2 + 2u^2 + 4u^2 = 8u^2. \end{array} \tag{17}$$

Let us now give a case-by-case analysis, for obtaining a bound on the relative error $|\eta|/(xy)$ and, in some cases, improving the bound (17).

2.7.1 If $x_h y_h \geq 2$

In that case $xy \geq x_h(1-u) \cdot y_h(1-u) \geq 2(1-u)^2$. So, that, using (17), the relative error is bounded by

$$\frac{4u^2}{(1-u)^2},$$

which is less than the bound (4) as soon as $u \leq 1/32$ (i.e., as soon as $p \geq 5$).

2.7.2 If $x_h y_h < 2$

We can observe that $|c_{\ell 1}| \leq u$, and that Lemma 2.10 implies $|t_\ell + x_\ell y_h| \leq x_h u + y_h u \leq 3u - 2u^2$, so that $|c_{\ell 2}| \leq 3u$. Therefore, $|c_{\ell 1} + c_{\ell 2}| \leq 4u$, so that $|\epsilon_4| \leq 2u^2$. This first improvement is at the origin of the "$6u^2$" in Theorem 2.4. However, we can improve the bound further by considering the following sub-cases:

- if $|c_{\ell 2}| \geq 2u$ then $c_{\ell 2}$ is a multiple of $4u^2$, therefore $c_{\ell 1} + c_{\ell 2}$ is a multiple of $4u^2$. Since it has an absolute value less than or equal to $4u$, this means that it is a floating-point number. An immediate consequence is $\mathrm{RN}(c_{\ell 1} + c_{\ell 2}) = c_{\ell 1} + c_{\ell 2}$ and $\epsilon_4 = 0$. This gives

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2 + 2u^2 + 0 = 4u^2. \end{array}$$

- if $|c_{\ell 2}| < 2u$, then $|\epsilon_3| \leq u^2$, so that

$$\begin{array}{rcl} |\eta| & \leq & |x_\ell y_\ell| + |\epsilon_1| + |\epsilon_3| + |\epsilon_4| \\ & \leq & u^2 + u^2 + u^2 + 2u^2 = 5u^2. \end{array}$$

To sum up, we have shown that when $x_h y_h < 2$, the absolute error bound η is less than $5u^2$. We can notice that if $x_h = 1$ or $y_h = 1$ then $c_{\ell 1} = 0$, so that $\epsilon_4 = 0$, so that $|\eta| \leq 3u^2$ and the relative error bound is less than

$$\frac{3u^2}{\left(1 - \frac{u}{2}\right)^2},$$

which is less than the bound (4) as soon as $u \leq 1/8$ (i.e., as soon as $p \geq 3$). Now, if x_h and y_h are different from 1, they are larger than or equal to $1 + 2u$, so that x and y are larger than or equal to $1 + u$, and the relative error is bounded by

$$\frac{5u^2}{(1+u)^2}$$

□

3 Formalization

We describe here our formalization in Coq of the major results published in the original paper [16] and the additional results presented in Section 2 (i.e., Theorems 2.6, 2.7, and 2.8), trying to keep the notations and theorem statements as close as possible to those of [16]. All our formal proofs, along with explanations for running them, are freely available at

`http://www-sop.inria.fr/members/Laurence.Rideau/DW_arithmetic-submitted_paper_release.zip`.

We will not give all details of the mathematical proofs that the reader will find in the original article and in Section 2 of this one, but we will focus on the problems we have encountered when doing the formalization.

3.1 Context of the formalized proofs

We first define the various notions used in the proofs published in [16]. For this purpose, we rely on the Flocq library [3, 4], which provides several definitions and theories (properties and proofs) on floating-point arithmetic.

```
Local Notation two := radix2. (* 2-radix from Flocq *)

(* bpow is the Flocq function taking a radix and an
   exponent as arguments
returning radix^e : 2^e here *)
Notation pow e := (bpow two e).
```

```
Variables p : Z. (* precision *)
Hypothesis Hp : (1 < p).

(* radix-2, precision-p floating-point with unlimited
   exponent range *)
Notation fexp := (FLX_exp p).
Notation format := (generic_format two fexp).
```

Let us now define in Coq the roundoff error unit u, and let us state and prove some properties.

```
(* we use the generic ulp function from Flocq associated to
    a given FP format defined by a radix and an exponent
   function *)
Notation ulp := (ulp two fexp).
(* The u constant *)
Let u := pow (-p).
Lemma u_ulp1: u = /2 * (ulp 1).
(* Property of the relative error *)
Lemma rel_error_u t (tn0 : t <> 0) : Rabs ((rnd_p t - t) /t
   ) <= u/(1+u).
```

Here, `rnd_p` stands for the round to nearest function RN in the FP set. It is defined as:

```
Variable choice : Z -> bool.
Local Notation rnd_p := (round two fexp (Znearest choice)).
```

Note that here the `choice` function is just declared as a boolean function on integers: no specific tie-breaking rule (such as the classical *ties-to-even* rule, which is, as said above, the default in IEEE 754 arithmetic) is specified. We will specify it, if needed, during the proof process. This makes it possible to formalize more general results and to clearly locate places when the ties-to-even tie breaking rule is necessary. The concept of double-word representation of a real number is defined in Coq as a property on a pair (x_h, x_ℓ) of real numbers, as follows (one easily checks that this is a direct transcription of Definition 1):

```
Definition double_word xh xl := ((format xh) /\ (format xl)
   )
                              /\ (xh = rnd_p (xh + xl)).
```

where `format x`, as defined above, means that the real number `x` belongs to the set of radix-2, precision-p FP numbers.

The Fast2Sum and TwoSum algorithms are provided by the Flocq library, along with the proofs that the pair of FP numbers they return satisfies the property that their sum is exactly the sum of the two FP numbers given in input:

```
(* first projection: rounded result *)
Definition TwoSum_sum a b := fst (TwoSum a b).

(* second projection: error *)
Definition TwoSum_err a b := snd (TwoSum a b).
```

```
Lemma TwoSum_correct a b (Fa : format a) (Fb : format b):
                    TwoSum_err a b = a + b - TwoSum_sum a b.
```

Note that, in the Flocq library, the proof of correctness of Fast2Sum(a, b) was based on the rather strong condition that $|a| \geq |b|$. This was not sufficient in some cases, for example for the proof of the **AccurateDWPlusDW** algorithm (Algorithm 4), and we had to formalize again the proof of correctness of Fast2Sum with the somehow weaker condition on the exponents given at the beginning of Section 2.1. This new, more general, proof should be soon included in the Flocq library.

Moreover, as some algorithms of [16] are used in other algorithms (for example **DWDivDW1** uses **DWTimesFP1**, assuming of course that the pair returned by **DWTimesFP1** is a double-word number), we have also needed to formalize the proof that their result is a double-word number. In all algorithms, the result is a pair returned by a Fast2Sum. So we have to prove that the pair (s, t) computed by a correct call to Fast2Sum (or by a call to TwoSum) is a double-word number.
Incidentally, that proof is straightforward : s, t, and the outputs of the rounding function rnd_p are FP numbers, and the correctness of the algorithms implies that $s + t = a + b$ with $s = rnd\_p\,(a + b)$ so that $rnd\_p\,(s + t) = rnd\_p\,(a + b) = s$.

3.2 Formalization of the DWPlusFP algorithm

DWPlusFP (Algorithm 5) is defined in Coq as follows:

```
Variables xh xl y : R.

Notation sh := (TwoSum_sum xh y).
Notation sl := (TwoSum_err xh y).
Notation v := (rnd_p (xl + sl)).

Definition DWPlusFP := (Fast2Sum sh v).
```

With these definitions, the correctness and error bound are stated in the following theorem (which is the Coq equivalent of Theorem 2.2):

```
Notation x := (xh + xl).
Notation zh := (Fast2Sum_sum sh v).
Notation zl := (Fast2Sum_err sh v).


Definition relative_errorDWFP := (Rabs (((zh + zl) - (x + y
   ))/ (x + y))).
Definition DWFP_add_correct xh xl y :=
 relative_errorDWFP xh xl y)<= 2 * u^2 /\ (double_word zh
    zl).

Theorem DWPlusFP_correct ( xh xl y : R)
      (Fy : format y)
      (DWx: double_word xh xl): DWFP_add_correct xh xl y.
```

As explained above, the fact that the result is a double-word number is a direct consequence of the fact that the last instruction of the algorithm is a Fast2Sum, so **DWPlusFP** returns a double-word number (if the Fast2Sum is correct, i.e., if it is called with operands that satisfy the required conditions on the exponents). Incidentally, let us note that the proof of correctness of the use of Fast2Sum (condition on the operands) is closely linked to the proof of correctness of the error bound, so these proofs cannot be done independently: the formalization, like the paper proof, must combine the two, i.e., formalizing that the result is a DW number requires to unroll all the proof of the error bound until the call to Fast2Sum.

The proofs of [16] all follow the same pattern: first of all they deal with particular cases: null operands, null intermediate sums, etc. Then, by using properties of the operation being implemented by the algorithm ($+$, $\times$, $\div$), the domain where the operands lie (or the domain where at least one of the operands lies) is restricted by using an argument of the form "without loss of generality, we can assume P", where P is a arithmetic property on the operands.

In the case of the **DWPlusFP** algorithm, in the first (unpublished) version submitted for review, the authors write: *"Now, without loss of generality, we can assume* $|x_h| \geq |y|$*. If this is not the case, since* x_h *and* y *play a symmetrical role in the algorithm, we can exchange them in our proof: We add the double-word number* (y, x_ℓ) *and the floating-point number* x_h*".*

We encountered our first problem of formalization while trying to prove that assertion in Coq. Let us first focus on the wlog (without loss of generality) notion in mathematics, then we will try to understand its translation in a proof assistant.

3.2.1 Wlog in mathematics and in Coq

When, in a mathematical proof, we want to use an argument of symmetry, we express that "without loss of generality" we can assume an adequate hypothesis, which does not change the desired result. For example, to prove a property `P(x,y)` in which `x` and `y` play symmetrical roles, we can assume that `x` $\leq$ `y`, which sometimes simplifies the proof (this is precisely that kind of assumption that was made in the case of **DWPlusFP**). This implicitly requires the reader to convince himself or herself of the equivalence of the result with or without this additional assumption. However, in the case of formal proofs, it is necessary to:

i) prove first the result assuming the added hypothesis, and

ii) show that from the proof with the added hypothesis one can deduce the proof in the general case.

To do this we use a simple but effective Coq tool: the "`wlog`" command. For example, to prove the proposal (P x y), the command:

```
wlog H: x y / x <= y
```

will generate two sub-goals:

```
H : \forall x y, x <= y -> P x y,
```

and[3]

```
H -> (P x y).
```

In addition, as we can see in the statement of hypothesis H, the variables x and y involved by the `wlog` extra assumption $x \leq y$, are generalized by the use of a "$\forall$". The Coq system also detects the assumptions implying x or/and y in the context, and signals an error to the user, requiring that these hypotheses should also be generalized for the logical consistency of the proof context. In the `wlog` command, the generalized objects (variables or hypotheses) are provided by the user between the ":" and the "/" that precede the `wlog` hypothesis (`x <= y` in our example). For instance, if the context contains the hypotheses "*x has a red nose*", "*y has blue eyes*" and "*x and y were born in the same year*", Coq asks for the generalization of these three hypotheses. It is up to the user to decide for each hypothesis if it is useful for the proof of `wlog` (and in this case he or she must generalize it), or if it is not useful (and in this case he or she must remove it from the context before calling "`wlog` "). So, if the context contains assumptions about x or y (let us call them `H1x`, `H2x`, `Hxy`, and `Hy`), the `wlog` command will require the generalization of these four assumptions when generalizing the associated x and y variables. If `H1x`, `Hxy` and `Hy` only are useful for the following of the `wlog` proof, the user will first remove `H2x` from the context (by calling the `clear H2x` command), then the call

```
wlog H: x y H1x Hxy Hy / x <= y
```

will generate as first sub-goal:

```
H : \forall x y H1x Hxy Hy, x<= y -> P x y,
```

meaning that $\forall x\ y$ such that `H1x`, `Hxy` and `Hy` hold, if $x \leq y$ then `P(x,y)` holds.

It is that generalization mechanism that allowed us to find an error.

Let us resume our work on the proof of the **DWPlusFP** algorithm. At this point of the proof, we know that (x_h, x_ℓ) is a DW number, y is a FP number, $x \neq 0$ and $x_h + y \neq 0$. The current context in Coq (the hypotheses are above the double line, and the current goal to be proven is below the double line) is the following:

```
  y, xh, xl : R
  Fy : format y
  DWx : double_word xh xl
  xh0 : xh <> 0
  xhy0 : xh + y <> 0
...
  ============================
  DWFP_add_correct xh xl y
```

Then calling the command:

```
wlog xhy : y xh Fy xhy0 DWx xh0 / Rabs y <= Rabs xh}
```

generates the two sub-goals:

[3] Note that in Coq, one writes a simple arrow "->" for an implication ($\Rightarrow$).

```
xh, xl, y : R
Fy : format y
DWx : double_word xh xl
xh0 : xh <> 0
xhy0 : xh + y <> 0
============================
(forall y xh : R,
 format y -> xh + y <> 0 -> double_word xh xl -> xh <> 0 ->
 Rabs y <= Rabs xh -> DWFP_add_correct xh xl y) ->
 DWFP_add_correct xh xl y
```

and

```
xl : R
y, xh : R
Fy : format y
xhy0 : xh + y <> 0
DWx : double_word xh xl
hxh0 : xh <> 0
xhy : Rabs y <= Rabs xh
============================
  DWFP_add_correct xh xl y
```

To prove the first sub-goal, we have to consider two cases. If $|y| \leq |x_h|$, then we obtain the result directly. If $|x_h| \leq |y|$, we call the hypothesis by exchanging x_h and y, so we must first prove that the `DWFP_add_correct` formula is symmetrical in x_h and y, and then that the other hypotheses (`format xh, y + xh <> 0, double_word y xl, y <> 0`) hold.

The symmetry of the formula is deduced from the commutativity of the sum (the variables s_h s_ℓ and v are symmetrical in x_h and y, as are z_h and z_ℓ, and the relative error). The same applies to the assumption $y + x_h \neq 0$. Moreover x_h being a double-word number, we get by definition that x_h is a FP number, and as $x_h \neq 0$ and $|x_h| \leq |y|$, we get that $y \neq 0$).

The last thing we need to prove is that the pair (y, x_ℓ) is a double-word number. We know that y and x_ℓ are floating point numbers (by hypothesis for y and by definition of (x_h, x_ℓ) being a DW number for x_ℓ). So we have just to prove that $y = \text{RN}(y + x_\ell)$. The pair (x_h, x_ℓ) being a double-word number, x_ℓ is negligible with respect to x_h, therefore it is negligible with respect to y, which is greater in absolute value than x_h. So it seems reasonable to think that (y, x_ℓ) is a DW number, which is what the authors assumed in a first draft of the article. But the problem is that this is not always true: in the particular case where both $(x_h + x_\ell)$ and $(y + x_\ell)$ are exactly halfway between two consecutive FP numbers, the rounding of (y, x_ℓ) could be the successor or the predecessor of y depending of the tie-breaking rule, i.e., depending on the `choice` function. So it is not always true that (y, x_ℓ) is a DW number and it was impossible to prove that hypothesis with Coq.

To solve the problem, the authors of [16] were able to add this little sentence as a footnote in the final version:

> (y, x_ℓ) may not be a double-word number, according to definition (...), in the case $x_\ell = 1/2\ ulp(y) = 1/2.ulp(x_h)$. However, one easily checks that in that case the algorithm returns an exact result.

This assertion was correct, and the algorithm does return an exact result (and thus a null error), but this result was not so easy to formalize. Indeed, in the case where the sum $(x_h + y)$ is not a floating number, there are several cases to consider to verify that the calculation of variable v at Line 2 of Algorithm 5 is exact (it is that calculation that can produce the error of the **DWPlusFP** algorithm) and that the conditions for the Fast2Sum algorithm to return a correct result are satisfied.

This example illustrates that a simple step of a paper proof may require a more laborious formalization work than expected. This example shows also that the use of wlogs in proofs can be dangerous.

However, wlogs are very useful in proofs of algorithms of FP arithmetic. They make it possible to reduce, by successive refinements, the interval of study for the input variables. For example for the correctness of **DWPlusFP**, the "wlog" is used three times, with the hypotheses $|y| \leq |x_h|$; then $0 < x_h$; and finally $1 \leq x_h \leq 2-2u$. Note that, in reference [16], this form of proof was used several dozen times, and only the case presented here was incorrect.

3.3 Methodology for proofs with "`wlog`"

During this formalization work, we developed a methodology to facilitate proofs using "`wlog`". The idea is to treat separately the core of the proof, in the particular case where all the hypotheses resulting from all the "wlog"(s) are satisfied. This makes it possible, thanks to notations and fixed operands, to follow as closely as possible the paper proof. Then we can treat the general case (with arbitrary operands: we generalize all the variables) which consists in treating first the particular cases, then the successive wlog(s), in order to apply the previous theorem at the very end.

3.4 Formalisation of the DWTimesDW1, DWTimesDW2, and DWTimesDW3 algorithms

Let us now consider the second error, in the proof of the double-word multiplication algorithms **DWTimesDW1**, **DWTimesDW2** and **DWTimesDW3** (Algorithms 6, 7, and 8 in this paper, Algorithms 10, 11 and 12 in [16]). The original proof used the wrong lemma 2.9, and remained undetected before publication. When formalizing, we have to recognise that we first tried to prove this wrong lemma! Then we thought it was just a typo in the statement and we continued the formalization to understand under which conditions it was used, this led us to be convinced that there really was an error. We first proved a close (but right!) lemma which allowed us to formalize the proofs of correctness of the first two theorems but which was not sufficient for the third one.

Lemma 3.1 (First replacement of the (wrong) Lemma 5.2 in [16], now replaced by the more accurate Lemma 2.10 of this paper)**.** *Let a and b be two positive real numbers. If $ab \leq 2$, $a \geq 1$ and $b \geq 1$, then $a + b \leq 3$.*

For the third theorem, we have needed the slightly stronger Lemma 2.10, given in Section 2.4.

The correct proofs (validated by Coq just by following the rationale of the paper proofs) are given in Section 2.4 of this paper (and with an improved error bound for each of the three algorithms).

Note that the paper [16] gave detailed proof, only of the first theorem, and suggested to the reader that the other proofs were similar. The error in the proof of the first theorem imposed a fortiori the formalization of the other theorems as well. In any case, claiming that the proofs are similar is not possible in a formalization unless the scripts are exactly the same.

Conclusion and general discussion

The first conclusion of this paper is that the algorithms and error bounds given in [16] are all correct, despite errors (one marginal and detected before publication, and one more serious) in two proofs. We have also improved some of the error bounds and shown the asymptotic optimality of the bound of Algorithm 4. Our formal proofs are available at

`http://www-sop.inria.fr/members/Laurence.Rideau/DW_arithmetic-submitted_paper_release.zip`.

This work provides the users with a set of formally proven algorithms for manipulating double-word numbers, with sharp error bounds: this helps making double-word arithmetic (once qualified by Kahan as "an attractive nuisance, like an unfenced backyard swimming pool" [21]) a trustable tool. A byproduct of our work is an improvement in the Flocq library, where the proof of correctness of Fast2Sum(a, b) will now use a weaker condition (see Section 3.1).

Beyond the particular case of these algorithms for manipulation of double-word numbers, this study leads to some observations concerning the use of formal proof techniques in computer arithmetic:

- beyond just *giving confidence* in the results, formal proof helps to generalize the results (for example, we could rather easily check which bounds do not depend on the tie-breaking rule of the RN rounding function: the initial proofs were built with the "ties to even" default tie-breaking rule in mind);
- the proofs of floating-point algorithms such as the ones presented in this paper and in [16] do not use, in general, very complex and abstract mathematics, but they are frequently very long, with often many particular cases that need to be considered. This makes them error-prone and, at times, quite boring to read (with the consequence that few people will actually fully check them, which is dangerous): formalization is very useful in that context;
- many classical results of the FP literature are still not formalized. Continuing efforts in this domain is necessary, because all of numerical computing is built upon the underlying arithmetic;

- tight cooperation between specialists of formal proof and specialists of FP arithmetic is necessary, otherwise too much time is lost trying to find again proofs of implicit sub-properties that are well-known by the specialists, or trying to understand strange intermediate results that are just straightforward typos in the paper (typically, the paper proof is OK but an error such as a wrong variable name was produced when typing it: the computer arithmetic specialist will not mind the typo, but the formal proof specialist will get lost). Training people competent in both computer arithmetic and formal proof would greatly help. Also, the formal proof tools remain complex to use by nonspecialists: making them more easily usable by researchers and engineers from other domains must remain a priority;
- "WLOGs" (*Without loss of generality...*) are necessary: without them the proofs would be much longer, much heavier, and therefore more likely to contain undetected errors (for instance, in the proof of Algorithm 6, one would have to carry everywhere the exponent of x_h instead of assuming *wlog*, that is is between 1 and 2). But they must be handled with much care, because they are the major source of nontrivial errors;
- maybe, when publishing paper proofs, we should distinguish between the "rough sketch of the proof", which is essential for the reader to understand the underlying idea, and the fully detailed proof, that could be a downloadable appendix. Ideally, that fully detailed proof would be a formal proof.

References

[1] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[2] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, 26(7):1196–1233, 2016.

[3] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *20th IEEE Symposium on Computer Arithmetic (ARITH-20)*, pages 243–252, Tübingen, Germany, 2011.

[4] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.

[5] Tim Coe and Ping Tak Peter Tang. It takes six ones to reach a flaw. In *12th IEEE Symposium on Computer Arithmetic (ARITH-12)*, pages 140–146, July 1995.

[6] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 169–184, Edinburgh, Scotland, 2001.

[7] Florent de Dinechin, Alexey V. Ershov, and Nicolas Gast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17)*, pages 288–295, 2005.

[8] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[9] John Harrison. Floating-point verification in HOL light: The exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[10] John Harrison. A machine-checked theory of floating point arithmetic. In *12th International Conference in Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, September 1999. Springer-Verlag, Berlin.

[11] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (ARITH-15)*, pages 155–162, June 2001.

[12] Yozo Hida, Xiaoye S. Li, and David H. Bailey. C++/Fortran-90 double-double and quad-double package, release 2.3.17, March 2012. Accessible electronically at `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`.

[13] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[14] Konstantin Isupov. Performance data of multiple-precision scalar and vector blas operations on cpu and gpu. *Data in Brief*, 30:105506, 2020.

[15] Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: optimal bounds and applications. *Mathematics of Computation*, 87:803–819, 2018.

[16] Mioara Joldeş, Jean-Michel Muller, and Valentina Popescu. Tight and rigourous error bounds for basic building blocks of double-word arithmetic. *ACM Transactions on Mathematical Software*, 44(2), 2017.

[17] Mioara Joldeş, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. CAMPARY: Cuda multiple precision arithmetic library and applications. In *5th International Congress on Mathematical Software (ICMS)*, volume 9725 of *Lecture Notes in Computer Science*, pages 232–240, Berlin, Germany, July 2016.

[18] William M. Kahan. Lecture notes on the status of IEEE-754. Available at `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`, 1997.

[19] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[20] Xiaoye S. Li, James Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, 2002.

[21] Xiaoye S. Li, James Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Anil Kapur, Michael C. Martin, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. Technical Report 45991, Lawrence Berkeley National Laboratory, 2000. `https://www.davidhbailey.com/dhbpapers/xblas-report.pdf`.

[22] Ole Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[23] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[24] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[25] Yves Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software*, 29(1):27–48, 2003.

[26] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[27] Jason Riedy and James Demmel. Augmented arithmetic operations proposed for IEEE-754 2018. In *25th IEEE Symposium on Computer Arithmetic*, pages 45–52, Amherst, MA, USA, June 2018.

[28] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry*, 18:305–363, 1997.