A new multiplication algorithm for extended precision using floating-point expansions

Jean-Michel Muller, Valentina Popescu and Ping Tak Peter Tang

Abstract—Some important computational problems must use a floating-point (FP) precision several times higher than the hardware-implemented available one. These computations critically rely on software libraries for high-precision FP arithmetic. The representation of a high-precision data type crucially influences the corresponding arithmetic algorithms. Recent work showed that algorithms for FP expansions, that is, a representation based on unevaluated sum of standard FP types, benefit from various high-performance support for native FP, such as low latency, high throughput, vectorization, threading, etc. Bailey's QD library and its corresponding Graphics Processing Unit (GPU) version, GQD, are such examples. Despite using native FP arithmetic as the key operations, QD and GQD algorithms are focused on double-double or quad-double representations and do not generalize efficiently or naturally to a flexible number of components in the FP expansion.

In this paper, we introduce a new multiplication algorithm for FP expansion with flexible precision, up to the order of tens of FP elements in mind. The main feature consists in the partial products being accumulated in a special designed data structure that has the regularity of a fixed-point representation while allowing the computation to be naturally carried out using native FP types. This allows us to easily avoid unnecessary computation and to present rigorous accuracy analysis transparently. The algorithm, its correctness and accuracy proofs and some performance comparisons with existing libraries are all contributions of this paper.

Index Terms—floating-point arithmetic, floating-point expansions, high precision arithmetic, multiple-precision arithmetic, multiplication

1 Introduction

Many numerical problems in dynamical systems or planetary orbit dynamics require higher precisions than the standard *double*-precision (now called *binary64* [1]). Examples include, to list a few, long-term stability analysis of the solar system [2], finding sinks in the Henon Map [3] and iterating the Lorenz attractor [4]. Since quadruple or higher precision is rarely implemented in hardware, these calculations rely on software-emulated higher-precision libraries, also called arbitrary-precision libraries.

A crucial design point of these libraries is the way higher-precision numbers are represented as this will influence the choice of arithmetic algorithms and their subsequent performance on various computer architectures, including highly parallel accelerators such as Graphics Processing Units (GPUs). There are mainly two ways of representing numbers in higher precision. The first one is the *multiple-digit* representation: each number is represented by one exponent, followed by a sequence of possibly high-radix digits. This representation gives a fixed-point flavor as the radix positions of the digits follow a very regular pattern. This characteristic greatly facilitates rigorous accuracy analysis on the associated arithmetic algorithms. The GNU package MPFR [5] is an open-source C library that uses the multiple-digit representation. Besides arbitrary precision, it also provides correct rounding for each atomic operation. One drawback, however, is that full exploitation of highly-optimized native floating-point (FP) instructions can be hard to achieve. The more memory intensive nature of the algorithms render them not a natural fit for GPUs, to which MPFR has yet to be ported.

The second way is the *multiple-term* representation in which each number is expressed as the unevaluated sum of several standard FP numbers. This sum is usually called a *FP expansion*. Because each term of this expansion has its own exponent, the term's positions can be quite irregular. Bailey's library QD [6] uses this approach and supports double-double and quad-double computations, i.e. numbers are represented as the unevaluated sum of 2 or 4 standard *double*-precision FP numbers. The FP centric nature allows QD to take great advantage of optimized native FP infrastructure; and the GPU version, GQD, is already available. Nevertheless, there are some drawbacks of FP expansions as implemented in QD. These algorithms do not straightforwardly generalize to an arbitrary number of terms, for example, a multiplication of two n-term FP expansion can easily lead to $O(n^3)$ complexity. Moreover, their nature made rigorous error analysis unnatural, that is why the operations implemented in QD do not come with any form of guaranteed error bounds.

In this article we focus on multiplication of FP expansions. Our algorithm accumulates the partial products

- *J.-M. Muller is with LIP Laboratory, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France*
  E-mail: jean-michel.muller@ens-lyon.fr
- *V. Popescu is with LIP Laboratory, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France*
  E-mail: valentina.popescu@ens-lyon.fr
- *P. Tak P. Tang is with Intel Corporation*
  E-mail: peter.tang@intel.com

of a multiplication using a fixed-point structure that is nevertheless FP friendly. The overall algorithm can be viewed as a generalized version of the paper-and-pencil method for long multiplication. The accumulation is done using a method by Rump, first presented in [7]. The regularity of this algorithm allows us to provide a thorough error analysis and tight error bound, while its FP friendliness make it a natural fit for GPUs.

The outline of the paper is the following: after recalling some basic notions about FP expansions in Section 2, we detail the new algorithm for multiplying FP expansions in Section 3. Here we also include a correctness proof, the computation of the specific error bound (Section 3.1) and the complexity analysis (Section 3.2). Finally, in Section 4 we assess the performance of our algorithm in terms of performance comparing to other existing libraries and we draw the conclusions.

2 Floating-point expansions

A normal binary precision-p floating-point (FP) number is a number of the form

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

with $2^{p-1} \leq |M_x| \leq 2^p - 1$. The integer e_x is called the *exponent* of x, and $M_x \cdot 2^{-p+1}$ is called the *significand* of x. We denote accordingly to Goldberg's definition $\mathrm{ulp}(x) = 2^{e_x - p + 1}$ [8, Chap. 2].

A natural extension of the notion of double-double or quad-double is the notion of *floating-point expansion*.

Definition 2.1. A *FP expansion* u with n terms is the unevaluated sum of n FP numbers u_0, u_1, ..., u_{n-1}, in which all nonzero terms are ordered by magnitude (i.e., if v is the sequence obtained by removing all zeros in the sequence u, and if sequence v contains m terms, $|v_i| \geq |v_{i+1}|$, for all $0 \leq i < m - 1$).

Arithmetics on FP expansions have been introduced by Priest [9], and later on by Shewchuk [10].

To make sure that such an expansion carries significantly more information than only one FP number, it is required that the u_i's do not "overlap". The notion of *(non-)overlapping* varies depending on the authors. We give here different definitions, using the above introduced notation. The first two, by Priest and Bailey, have already been presented in the literature, but the third one introduces a new, weaker condition, that allows for a more relaxed handling of the FP expansions.

We specify first that even if a FP expansion may contain interleaving zeros, all the definitions that follow apply only to the non-zero terms of the expansion (i.e., the array v in Definition 2.1).

Definition 2.2. Assuming x and y are normal numbers with representations $M_x \cdot 2^{e_x - p + 1}$ and $M_y \cdot 2^{e_y - p + 1}$ (with $2^{p-1} \leq |M_x|, |M_y| \leq 2^p - 1$), they are *$\mathcal{P}$-nonoverlapping* (that is, nonoverlapping according to Priest's definition [11]) if $|e_y - e_x| \geq p$.

Definition 2.3. An expansion is *$\mathcal{P}$-nonoverlapping* (that is, nonoverlapping according to Priest's definition [11]) if all its components are mutually *$\mathcal{P}$-nonoverlapping*.

A slightly stronger sense of nonoverlapping was introduced by Hida, Li and Bailey [6]:

Definition 2.4. An expansion u_0, u_1, ..., u_{n-1} is *$\mathcal{B}$-nonoverlapping* (that is, nonoverlapping according to Bailey's definition [6]) if for all $0 < i < n$, we have $|u_i| \leq \frac{1}{2}\,\mathrm{ulp}(u_{i-1})$.

Remark 2.5. Note that for *$\mathcal{P}$-nonoverlapping* expansions we have $|u_i| \leq \frac{2^p - 1}{2^p}\,\mathrm{ulp}(u_{i-1})$.

Intuitively, the stronger the sense of the nonoverlapping definition, the more difficult it is to obtain, implying extra manipulation of the FP expansions. In order to save operations we chose to use a slightly weaker sense of nonoverlapping, referred to as ulp-*nonoverlapping* , that we define in what follows.

Definition 2.6. An expansion u_0, u_1, ..., u_{n-1} is ulp-*nonoverlapping* if for all $0 < i < n$, we have $|u_i| \leq \mathrm{ulp}(u_{i-1})$.

In other words, the components are either *$\mathcal{P}$-nonoverlapping* or they overlap by one bit, in which case the second component is a power of two.

Depending on the nonoverlapping type of an expansion, when using standard FP formats for representation, the exponent range forces a constraint on the number of terms. The largest expansion can be obtained when the largest term is close to overflow and the smallest is close to underflow. We remark that, when using any of the above nonoverlapping definitions, for the two most common FP formats, the constraints are:

- for *double*-precision (exponent range $[-1022, 1023]$) the maximum expansion size is 39;
- for *single*-precision (exponent range $[-126, 127]$) the maximum is 12.

The majority of algorithms performing arithmetic operations on FP expansions are based on the so-called *error-free transforms* (such as the algorithms 2Sum, Fast2Sum, Dekker's product and 2MultFMA presented for instance in [8]), that make it possible to compute both the result and the error of a FP addition or multiplication. This implies that each such *error-free transform* applied to two FP numbers, returns *still two* FP numbers. So, when adding two expansions x and y with n and m terms, respectively, the exact result is going to have at most $n+m$ terms. Similarly, for multiplication, the exact product is going to have at most $2nm$ terms [9]. A potential problem appears when subsequent computations are done using this results; the size of the exact result is going to increase more and more. To avoid this, some "truncation" methods (both *on-the-fly* or *a-posteriori*) may be used to compute only an approximation of the exact result. Also, so-called *(re-)normalization* algorithms are used to render the result nonoverlapping, which implies also a potential reduction in the number of components.

In what follows we will present a new multiplication algorithm that allows for computing the "truncated" product of two FP expansions. We also provide a full correctness proof and error analysis for it.

3 Multiplication algorithm for floating-point expansions

In Algorithm 1 we present the multiplication algorithm. Throughout this paper we assume that no underflow / overflow occurs during the calculations. We consider two ulp-*nonoverlapping* expansions x and y, with n and m terms, respectively, and we compute the r most significant components of the expansion $\pi = \pi_0, \pi_1, \pi_2, \ldots$ that represents the product $x \cdot y$.

Algorithm 1 Algorithm for multiplication of FP expansions. We denote by p, the precision of the used FP format and by b, the size of the bins. The algorithms called are going to be detailed later on in this paper.

Input: ulp-*nonoverlapping* FP expansions $x = x_0 + \ldots + x_{n-1}$; $y = y_0 + \ldots + y_{m-1}$.
Output: ulp-*nonoverlapping* FP expansion $\pi = \pi_0 + \ldots + \pi_{r-1}$.

1: $e \leftarrow e_{x_0} + e_{y_0}$
2: **for** $i \leftarrow 0$ to $\lfloor r \cdot p/b \rfloor + 1$ **do**
3: $\quad B_i \leftarrow 1.5 \cdot 2^{e-(i+1)b+p-1}$
4: **end for**
5: **for** $i \leftarrow 0$ to $min(n-1, r)$ **do**
6: $\quad$ **for** $j \leftarrow 0$ to $min(m-1, r-1-i)$ **do**
7: $\quad\quad (P, E) \leftarrow 2\text{MultFMA}(x_i, y_j)$
8: $\quad\quad \ell \leftarrow e - e_{x_i} - e_{y_j}$
9: $\quad\quad sh \leftarrow \lfloor \ell / b \rfloor$
10: $\quad\quad \ell \leftarrow \ell - sh \cdot b$
11: $\quad\quad B \leftarrow \text{Accumulate}(P, E, B, sh, \ell)$
12: $\quad$ **end for**
13: $\quad$ **if** $j < m-1$ **then**
14: $\quad\quad P \leftarrow x_i \cdot y_j$
15: $\quad\quad \ell \leftarrow e - e_{x_i} - e_{y_j}$
16: $\quad\quad sh \leftarrow \lfloor \ell / b \rfloor$
17: $\quad\quad \ell \leftarrow \ell - sh \cdot b$
18: $\quad\quad B \leftarrow \text{Accumulate}(P, 0., B, sh, \ell)$
19: $\quad$ **end if**
20: **end for**
21: **for** $i \leftarrow 0$ to $\lfloor r \cdot p/b \rfloor + 1$ **do**
22: $\quad B_i \leftarrow B_i - 1.5 \cdot 2^{e-(i+1)b+p-1}$
23: **end for**
24: $\pi[0 : r-1] \leftarrow \text{Renormalize}(B[0 : \lfloor r \cdot p/b \rfloor + 1])$
25: **return** FP expansion $\pi = \pi_0 + \ldots + \pi_{r-1}$.

The way we compute the partial products is based on term-times-expansion products, $x_i \cdot y$, and it follows the paper-and-pencil technique. In order to gain performance we "truncate" the computations by discarding the partial products that have an order of magnitude less than π_r, meaning that we compute the approximate result based on the first $\sum_{k=1}^{r+1} k$ partial products. The products with the same order of magnitude as π_r are intended as an extra correction term, that is why we compute them using only standard FP multiplication.

The main idea of the algorithm is to accumulate numbers of size at most b in "containers", referred to as bins, that are FP variables whose least significant bit (LSB) has a fixed weight. This allows for errorless accumulation provided that we do not add more than 2^c numbers to one bin, where $b + c = p - 1$.

Even though in this article we use a general notation, our implementation is done for standard FP formats. When using *double*-precision ($p = 53$) we defined bins of size $b = 45$, that allows for $c = 7$ bits of carry to happen; this means that we can add 128 numbers that satisfy the above condition to each bin and the result is still going to be exact. For *single*-precision ($p = 24$) we chose bins with $b = 18$, which implies $c = 5$, allowing us to add up to 32 numbers to one bin. In both cases, these values also satisfy $3b > 2p$, a property that we are going to use later on. We suggest the use of the latter one only if is the only standard FP format available or if, by any reason, is much faster than the first one.

The number of allocated bins is computed as $\lfloor \frac{r \cdot p}{b} \rfloor + 2$ and the LSB of each bin is set according to the starting exponent, $e = e_{x_0} + e_{y_0}$, at a distance of b bits. We start the algorithm by initializing each bin B_i with the value $1.5 \cdot 2^{e-(i+1) \cdot b+p-1}$.

After the initialization step is done we start the actual computations. For each partial product computed using *2MultFMA* we get the pair (P, E) (line 7 of Algorithm 1) and, using the formula $\lfloor (e - e_{x_i} - e_{y_j})/b \rfloor$, we determine the corresponding bins in which we have to accumulate it.

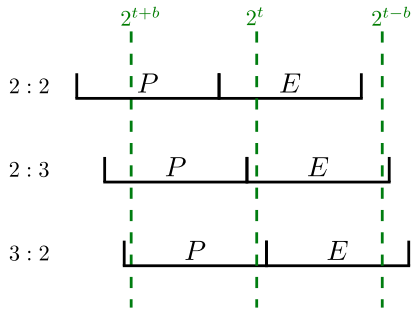

Fig. 1: Cases for accumulating the partial products into the bins.

We know that one pair of FP numbers can "fall" into at most four bins ($3b > 2p$), and we can deduce three different cases (see Fig. 1):

- 2:2 case, in which both P and E fall into two bins each;
- 2:3 case, in which P falls into two bins and E into three;

- 3:2 case, in which P falls into three bins and E into two.

These cases are dealt with in Algorithm 2 that accumulates the partial products. Apart from the (P, E) pair and the bins array, B, the algorithm also receives two integer parameters. The sh value represents the first corresponding bin for the pair and ℓ, computed as $e - e_{x_i} - e_{y_j} - sh \cdot b$, the number of leading bits. This value gives the difference between the LSB of B_{sh-1} and the sum of the exponents of x_i and y_j of the corresponding (P, E) pair.

We determine which one of the three cases apply depending on the ℓ value. So, if $2c+1 < b-\ell \leq b$ we are in the 2:2 case; if $c < b - \ell \leq 2c+1$ we need to consider the 2:3 case; and if $0 < b - \ell \leq c$ the 3:2 case applies.

Remark 3.1. For simplicity, we consider that the extra error correction partial products, the ones that are computed using standard FP multiplication, are dealt with the same way, using the pair $(P, 0)$. This is not the case in our implementation, where we try to save operations by accumulating only the P term.

Algorithm 2 Accumulate(P, E, B, sh, ℓ).

Input: FP numbers P, E; FP array B;
Integers sh and ℓ.
Output: FP array B.

if $\ell < b - 2c - 1$ **then**
  $(B_{sh}, B_{sh+1}) \leftarrow \text{Deposit}(P)$
  // that is, $(B_{sh}, P) \leftarrow \text{Fast2Sum}(B_{sh}, P)$, and
  // $B_{sh+1} \leftarrow B_{sh+1} + P$
  $(B_{sh+1}, B_{sh+2}) \leftarrow \text{Deposit}(E)$
else if $\ell < b - c$ **then**
  $(B_{sh}, B_{sh+1}) \leftarrow \text{Deposit}(P)$
  $(B_{sh+1}, E) \leftarrow \text{Fast2Sum}(B_{sh+1}, E)$
  $(B_{sh+2}, B_{sh+3}) \leftarrow \text{Deposit}(E)$
else
  $(B_{sh}, P) \leftarrow \text{Fast2Sum}(B_{sh}, P)$
  $(B_{sh+1}, B_{sh+2}) \leftarrow \text{Deposit}(P)$
  $(B_{sh+2}, B_{sh+3}) \leftarrow \text{Deposit}(E)$
end if
return FP array B.

As we stated before, all the bins are initialized with a constant value depending on their LSB, that is going to be subtracted before the renormalization step. This type of addition was first used by Rump (in [7]) for adding the elements of an array of FP numbers. In his paper he proved that the result is correct. For the sake of completeness we also give here a short correctness proof.

Proof of correctness for Algorithm 2

In what follows we will prove that when accumulating the numbers as we do in Algorithm 2 no rounding errors can occur and the result is exact.

We consider all the values that fall into the same bins B_{sh}, B_{sh+1} as an array of FP numbers $x_1, \ldots, x_n$ that satisfy $|x_i| < 2^{e+b}$, where 2^e is the LSB of B_{sh} and b is the size of the bins. The lower part of each x_i is denoted by x_i^l and represents the part that will be accumulated into B_{sh+1}.

Proposition 3.2. *Let $x_1, \ldots, x_n$ an array of precision-p FP numbers that satisfy $|x_i| < 2^{e+b}$, for all $0 < i \leq n$, where $b < p-1$. We initialize a FP container with $s_0 = 1.5 \cdot 2^{e+p-1}$, we compute $s_1 = \text{RN}(s_0 + x_1); \ldots; s_i = \text{RN}(s_{i-1} + x_i); \ldots; s_n = \text{RN}(s_{n-1} + x_n)$; and we return the value $\text{RN}(s_n - s_0)$. For each x_i we also compute the lower part $x_i^\ell = \text{RN}(\text{RN}(s_{i-1} - s_i) + x_i)$. When using this method, no significant informations is lost in the process, though no rounding can occur, provided that $n \leq 2^{p-b-2} - 1$ and that no underflow / overflow occurs during the calculations.*

Proof: We first prove by induction that the following statement holds:

$$1.5 \cdot 2^{e+p-1} - i \cdot 2^{e+b} \leq s_i \leq 1.5 \cdot 2^{e+p-1} + i \cdot 2^{e+b}. \quad (1)$$

It is easy to see that it holds for $i = 0$. Now we assume that is true for i and we try to prove it for $i+1$. We deduce:

$$1.5 \cdot 2^{e+p-1} - (i+1)2^{e+b} \leq s_i + x_{i+1} \leq 1.5 \cdot 2^{e+p-1} + (i+1)2^{e+b}.$$

Hence, since rounding is a monotonic function:

$$\text{RN}(1.5 \cdot 2^{e+p-1} - (i+1)2^{e+b}) \leq s_{i+1} \leq \text{RN}(1.5 \cdot 2^{e+p-1} + (i+1)2^{e+b}).$$

The value $1.5 \cdot 2^{e+p-1} - (i+1)2^{e+b}$ is an exact FP number because it is a multiple of 2^e and it is less than 2^{e+p} in absolute value, provided that $i \leq 2^{p-b-1} \cdot 3.5 - 1$, which holds in all practical cases (with our parameters: $i \leq 447$ for *double*-precision –$p = 53$ and $b = 45$– and $i \leq 111$ for *single*-precision –$p = 24$ and $b = 18$–). The same holds for $1.5 \cdot 2^{e+p-1} + (i+1)2^{e+b}$ provided that $i \leq 2^{p-b-2} - 1$, which also holds in all practical cases (with our parameters: $i \leq 63$ for *double*-precision and $i \leq 15$ for *single*-precision).

Furthermore, we have $(s_i, x_i^\ell) = \text{Fast2Sum}(s_{i-1}, x_i)$, therefore

$$\forall i, s_i + x_i^\ell = s_{i-1} + x_i,$$

such that, by induction,

$$s_i + x_i^\ell + x_{i-1}^\ell + \ldots + x_1^\ell = s_0 + x_1 + x_2 + \ldots + x_i,$$

which implies

$$(s_n - s_0) + x_n^\ell + x_{n-1}^\ell + \ldots + x_1^\ell = x_1 + x_2 + \ldots + x_n.$$

From (1), we easily find that s_n and s_0 are within a factor 2 (in practice, much less), such that (from Sterbenz' lemma), their difference is exactly computed: $\text{RN}(s_n - s_0) = s_n - s_0$. We therefore conclude that

$$\text{RN}(s_n - s_0) + x_n^\ell + x_{n-1}^\ell + \ldots + x_1^\ell = x_1 + x_2 + \ldots + x_n.$$

□

The last part of Algorithm 1 consists in applying a renormalization algorithm (see Algorithm 3) on the bins array, in order to render the result ulp-*nonoverlapping* . This is a chain of *Fast2Sum* starting from the two most significant components and propagating the errors. If however, the error after a *Fast2Sum* block is zero, then we propagate the sum (this is shown in Figure 2).

Algorithm 3 Renormalize(x).

Input: FP array $x = (x_0, \ldots, x_{n-1})$;
Output: ulp-*nonoverlapping* FP expansion $r = r_0 + \ldots + r_{k-1}$.

1: $\varepsilon \leftarrow x_0$
2: $j \leftarrow 0$
3: $i \leftarrow 1$
4: **while** $i < n$ and $j < k$ **do**
5: $(r_j, \varepsilon) \leftarrow \text{Fast2Sum}(\varepsilon, x_i)$
6: **if** $\varepsilon = 0$ **then**
7: $\varepsilon \leftarrow r_j$
8: **else**
9: $j \leftarrow j + 1$
10: **end if**
11: $i \leftarrow i + 1$
12: **end while**
13: **if** $\varepsilon \neq 0$ and $j < k$ **then**
14: $r_j \leftarrow \varepsilon$
15: **end if**
16: **return** FP expansion $r = r_0 + \ldots + r_{k-1}$.

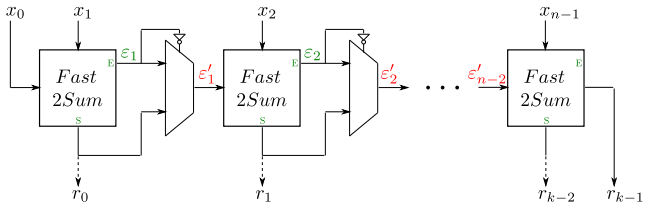

Fig. 2: *Renormalize* with n terms. Each *Fast2Sum* box performs an error-free transform; the sum is outputted downwards and the error to the right. If the error is zero, the sum is propagated to the right, otherwise the error is propagated and the sum is outputted.

Proof of correctness for Algorithm 3

In what follows we will prove the correctness of Algorithm 3 by using the notations seen in Figure 2.

Proposition 3.3. *Let an input array $x = (x_0, \ldots, x_{n-1})$ that satisfies: x_i is multiple of $2^{e_0 - ib} = 2^{e_i}$ and $|x_{i+1}| < 2^{e_i + c + 1}$, for all $0 \leq i < n-1$, where $b + c = p - 1$ ($c << b$). Also $|x_0| > |x_1|$ and $|x_0| < 2^{e_0 + p + 1}$. After applying Algorithm 3, the output array $r = (r_0, \ldots, r_{k-1})$, with $0 \leq k \leq n-1$ is an* ulp-nonoverlapping *expansion, provided that no underflow / overflow occurs during the calculations.*

Proof: The case when x contains 1 or 2 elements is trivial. Consider now at least 3 elements. By the input type we know that:

$$\begin{aligned} x_0 &= X_0 \cdot 2^{e_0}, \\ x_1 &= X_1 \cdot 2^{e_1} \text{ with } e_1 = e_0 - b. \end{aligned}$$

Hence, r_0 and ε_1 are both multiples of $2^{e_0 - b}$. Two possible cases may occur:

(i) $\varepsilon_1 = 0$. If we choose to propagate directly ε_1, then $r_1 = x_2$ and $\varepsilon_2 = 0$. This implies, by induction, that $r_i = x_{i+1}, \forall i \geq 1$. So, directly propagating the error poses a problem, since the whole remaining chain of *Fast2Sum* is executed without any change between the input and the output. So, as shown in Algorithm 3, line 7, when $\varepsilon_i = 0$ we propagate the sum r_j, so $\varepsilon'_i \leftarrow r_j$.

(ii) $\varepsilon_1 \neq 0$. By definition of *Fast2Sum*, we have $|\varepsilon_1| \leq \frac{1}{2}\,\text{ulp}(r_0)$. We also have $|x_0| > |x_1|$, so $|r_0| = |\text{RN}(x_0 + x_1)| \leq 2\,|x_0|$. Hence: $|\varepsilon_1| < 2^{e_0 + 2}$.

We now prove by induction the following statement: at each step $i \geq 1$ of the loop in Algorithm 3, both r_{i-1} and ε_i are multiples of 2^{e_i} and $\varepsilon_i = 0$ or $|\varepsilon_i| < 2^{e_i + b + 2}$, meaning that ε_i fits in at most $b+1$ bits. We proved above that for $i = 1$ this holds. Suppose now it holds for i and prove it for $i + 1$.

At this step we have $\varepsilon'_i + x_{i+1} = r_i + \varepsilon_{i+1}$. Since ε'_i is a multiple of 2^{e_i} and x_{i+1} is a multiple of $2^{e_{i+1}}$, with $e_{i+1} = e_i - b$, then both r_i and ε_{i+1} are multiples of $2^{e_{i+1}}$. Two cases may occur:

– if $|\varepsilon'_i| < 2^{e_i + c + 1}$ then $\varepsilon'_i + x_{i+1}$ is a FP number, which implies $r_i = \varepsilon'_i + x_{i+1}$ exactly and $\varepsilon_{i+1} = 0$, in which case we propagate $r_i < 2^{e_i + c + 2}$.

– if $|\varepsilon'_i| > 2^{e_i + c}$ we have $|r_i| \leq 2\,|\varepsilon'_i|$ and we get (by definition of *Fast2Sum*):

$$\begin{aligned} |\varepsilon_{i+1}| &\leq \frac{1}{2}\,\text{ulp}(r_i) \\ &< 2^{-p} \cdot 2 \cdot 2^{e_i + b + 2} \\ &< 2^{e_i - c + 2} \end{aligned} \tag{2}$$

This condition is even stronger than what we were trying to prove, so the induction holds.

Finally, we prove the relation between r_{i-1} and r_i. If $\varepsilon_i = 0$, we propagate r_{i-1}, i.e. $\varepsilon'_i \leftarrow r_{i-1}$. Otherwise $|r_i| = |\text{RN}(\varepsilon'_i + x_{i+1})| \leq 2\,|\varepsilon'_i|$ and since $\varepsilon'_i \leq \frac{1}{2}\,\text{ulp}(r_{i-1})$, then $|r_i| \leq \text{ulp}(r_{i-1})$ and the proposition is proven. □

Remark 3.4. After using Algorithm 3, the result $\pi = \pi_0 + \ldots + \pi_{r-1}$ (of Algorithm 1) cannot have interleaving zeros; zeros may appear only at the end of the expansion.

3.1 Error analysis for multiplication

The multiplication algorithm explained in the above section returns only an approximation of the exact result, but the maximum error that can occur is bounded. We give here the error bound that we computed and we prove its correctness.

Proposition 3.5. *Let x and y be two* ulp-nonoverlapping *FP expansions, with n, and m terms, respectively. If, for computing the product xy we use Algorithm 1, the result π,*

with r terms, satisfies: $|xy-\pi| \leq |x_0y_0|\,2^{-(p-1)r}[1+(r+1)2^{-p}+2^{-(p-1)}\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}+\frac{m+n-r-2}{1-2^{-(p-1)}}\right)]$, provided that no underflow / overflow occurs during the calculations.

For simplicity, we will first prove an intermediate proposition, and only after that we proceed to the proof on the final bound.

Proposition 3.6. *Let x and y be two* ulp-nonoverlapping *FP expansions, with n, and m terms, respectively. If, when computing the product xy we "truncate" the operations by adding only the first $\sum_{k=1}^{r+1} k$ partial productsproducts, where r is the required size of the final result, then the generated error satisfies: $\left|xy-\sum_{k=0}^{r}\sum_{i+j=k}x_iy_j\right| \leq |x_0y_0|\,2^{-(p-1)(r+1)}\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}+\frac{m+n-r-2}{1-2^{-(p-1)}}\right)$, provided that no underflow / overflow occurs during the calculations.*

Proof: From the definition of the ulp-*nonoverlapping* expansion we have $x_1 \leq \mathrm{ulp}(x_0) \leq 2^{-p+1}|x_0|$ and, by induction, we get $x_i \leq 2^{-pi+i}|x_0|$, for all $0<i<n$. The same goes for y.

The discarded partial products satisfy:

$$\begin{aligned}\sum_{k=r+1}^{m+n-2}\sum_{i+j=k}a_ib_j &\leq \sum_{k=r+1}^{m+n-2}\sum_{i+j=k}2^{-p(i+j)+i+j}|x_0y_0| \\ &\leq |x_0y_0|\sum_{k=r+1}^{m+n-2}\sum_{i+j=k}2^{-(p-1)k} \\ &\leq |x_0y_0|\sum_{k=r+1}^{m+n-2}(m+n-1-k)2^{-(p-1)k} \\ &\leq |x_0y_0|\sum_{k'=0}^{m+n-r-3}(m+n-k'-r-2)2^{-(p-1)(k'+r+1)} \\ &\leq |x_0y_0|\,2^{-(p-1)(r+1)} \\ &\times \sum_{k'=0}^{m+n-r-3}(m+n-r-2-k')2^{-(p-1)k'}.\end{aligned} \tag{3}$$

We define the function $\phi(e)=\sum_{k=0}^{\infty}(m+n-r-2-k)e^k$ that satisfies:

$$\begin{aligned}\phi(e) &= \sum_{k=0}^{\infty}-ke^k+\sum_{k=0}^{\infty}(m+n-r-2)e^k \\ &= -e\sum_{k=1}^{\infty}ke^{k-1}+(m+n-r-2)\sum_{k=0}^{\infty}e^k \\ &= -e\frac{d}{de}\left(\sum_{k=1}^{\infty}e^k\right)+(m+n-r-2)\frac{1}{1-e} \\ &= -e\frac{d}{de}\left(\frac{1}{1-e}\right)+\frac{m+n-r-2}{1-e} \\ &= \frac{-e}{(1-e)^2}+\frac{m+n-r-2}{1-e}.\end{aligned}$$

When applying function $\phi(2^{-(p-1)})$ in equation (3) we get:

$$\sum_{k=r+1}^{m+n-2}\sum_{i+j=k}x_iy_j \leq |x_0y_0|\,2^{-(p-1)(r+1)}\cdot\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}+\frac{m+n-r-2}{1-2^{-(p-1)}}\right),$$

which concludes the proof. □

We are now able to prove Prop. 3.5.

Proof: (of Prop. 3.5) When using Algorithm 1 we "truncate" the result by discarding the partial products with an order of magnitude less than π_r. From Prop. 3.6 we know that this causes a maximum error that is less or equal to

$$|x_0y_0|\,2^{-(p-1)(r+1)}\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}+\frac{m+n-r-2}{1-2^{-(p-1)}}\right). \tag{4}$$

Also, in the algorithm we do not use error-free transforms for computing the last $r+1$ partial products, the ones with the same order of magnitude as π_r, for which $i+j=r$. We know that $|x_iy_j| \leq 2^{-(p-1)(i+j)}|x_0y_0|$, from where $|x_iy_j-\mathrm{RN}(x_iy_j)| \leq 2^{-(p-1)r}|x_0y_0|\,2^{-p}$. This implies that the maximum error caused doing this is less or equal to:

$$(r+1)|x_0y_0|\,2^{-(p-1)r}\cdot 2^{-p}. \tag{5}$$

Apart from these two possible errors we also need to account for the error caused by the renormalization step. We already showed that Algorithm 3 returns an ulp-*nonoverlapping* expansion, in which case the maximum error is less or equal to $\mathrm{ulp}(\pi_{r-1})$. This implies that is less or equal to:

$$|x_0y_0|\,2^{-(p-1)r}. \tag{6}$$

To get the final error bound we have to add the bounds on all the possible errors that can occur, Eq. (4), (5) and (6), and we get:

$$\begin{aligned}&|x_0y_0|2^{-(p-1)r}[1+(r+1)2^{-p}+ \\ &\quad+2^{-(p-1)}\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}+\frac{m+n-r-2}{1-2^{-(p-1)}}\right)]\end{aligned}$$

This concludes our proof. □

3.2 Complexity analysis for multiplication

As presented before our algorithm has the advantage of computing "truncated" expansions on the fly, by taking into account only the significant terms of the input expansions.

We present here the worst case FP operation count of our algorithm, by taking ([8]) 3 FP operations for *Fast2Sum* and 2 for *2MultFMA*. For the sake of simplicity, we will consider that the input expansions and the result have the same number of terms, say k.

Proposition 3.7. *When using Algorithm 1 for computing the product of two FP expansions with k terms, we perform $\frac{13}{2}k^2+\frac{33}{2}k+6(\lfloor\frac{k\cdot p}{b}\rfloor+2)+55$ FP operations.*

Proof: During a preprocessing step we need to get the exponents of each term of the input expansions. We do this using the *math.h* library function, *frexp*, that uses only one FP operation, which we call $2k$ times.

The first step of the algorithm consists in initializing the bins; as mentioned in Sec. 3, we allocate $binNr = \lfloor \frac{k \cdot p}{b} \rfloor + 2$ bins. For this we use twice the *math.h* library function, *ldexp* (that can take up to 34 FLOPS, depending on the exponent's size), and after that we perform $binNr - 1$ FP multiplications.

During the main loop of the algorithm we compute $\sum_{i=1}^{k} i$ partial products using *2MultFMA*, which we accumulate into the bins using 3 *Fast2Sum* calls and 2 FP additions. During this same loop we also compute $k-1$ correction terms using simple FP multiplication, which we can accumulate using only 2 *Fast2Sum* calls and one FP addition.

In the last part of the algorithm we first unbias the bins, by subtracting their initial values from each one of them, followed by the renormalization of the result. This accounts for a total of $binNr$ FP subtractions followed by $binNr - 1$ calls to *Fast2Sum* and comparisons.

When adding all the operations accounted for above, we conclude that Algorithm 1 requires a total of $\frac{13}{2}k^2 + \frac{33}{2}k + 6(\lfloor \frac{k \cdot p}{b} \rfloor + 2) + 55$ FP operations.

□

4 Comparison and Discussion

In Table 1 we give effective values of the worst case FP operation count for our algorithm (see Sec. 3.2) vs. Priest's multiplication algorithm ([9]), which, when multiplying two $\mathcal{P}$*-nonoverlapping* expansions with n and respectively m terms, it requires $81\,mn^2 + 747\,nm + 2\,m - 233\,n$ FP operations.

We chose to use it for comparison because it is more general than the QD/GQD one, which is limited to double-double and quad-double precision, plus, from our knowledge, is the only algorithm in the literature of FP expansions provided with a complete correctness proof and error analysis. This algorithm is not tuned for obtaining "truncated" results on the fly, meaning that it computes the result fully and only then truncates, which allows for a straightforward error analysis, but makes it more costly.

TABLE 1: Effective values of the worst case FP operation count for Algorithm 1 and Priest's multiplication algorithm [9] when the input and output expansions are of size r.

r	2	4	8	16
Algorithm 1	138	261	669	2103
Priest's mul. ([9])	3174	16212	87432	519312

In practice however, Baileys QD/GQD and MPFR libraries are very often used, so we present the performance results comparing to these. We do this for completeness reasons, even though the comparison is not completely fair since we target different things. To be exact:
(i) MPFR uses a different representation for extended precision and is limited to CPU use;
(ii) QD and GQD libraries are limited to double-double and quad-double precisions, plus the algorithm used in the implementation of quad-double are not provided with any correctness proof and there is no specific error bound given.

This algorithm is part of the CAMPARY (CudA Multiple Precision ARithmetic librarY) software that is available at: http://homepages.laas.fr/mmjoldes/campary/. Our implementation was done using CUDA C – an extension of the C language developed by NVIDIA [12] for their GPUs. Algorithm 1 is suitable for GPU use, since all basic operations $(+, -, *, /, \sqrt{})$ conform to the IEEE 754-2008 standard for FP arithmetic for *single*- and *double*-precision, support for the four rounding modes is provided and dynamic rounding mode change is supported without any penalties. The `fma` instruction is supported for all devices with *Compute Capability* at least 2.0, which allows us to use the *2MultFMA* algorithm.

In the implementation we use templates for the number of terms in the expansions, meaning that we allow static generation of any input-output precision combinations (e.g. add a double-double with a quad-double and store the result on triple-double). All the functions are defined using __host__ __device__ specifiers, which allows for the library to be used on both CPU and GPU. We would like to stress here the fact that the algorithm is not itself parallelized; our intention is to provide a multiple precision alternative for problems that are suitable for GPU use.

In Table 2 we give some CPU performance measurements obtained on an Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz computer. The values are given in MFlops/s and by one Flop we understand one operation using extended precision.

TABLE 2: Performance in MFlops/s for multiplying two FP expansions on the CPU; d_x and d_y represent the number of terms in the input expansions and d_r is the size of the computed result. $*$ precision not supported

d_x, d_y, d_r	Algorithm 1	QD	MPFR
2, 2, 2	11.69	99.16	18.64
1, 2, 2	14.96	104.17	19.85
3, 3, 3	6.97	$*$	12.1
2, 3, 3	8.62	$*$	13.69
4, 4, 4	4.5	5.87	10.64
1, 4, 4	8.88	15.11	14.1
2, 4, 4	6.38	9.49	13.44
8, 8, 8	1.5	$*$	6.8
4, 8, 8	2.04	$*$	9.15
16, 16, 16	0.42	$*$	2.55

Table 3 contains the performance values obtained on a GPU, in comparison to the GQD library. The tests were performed on a Tesla K40c GPU, using CUDA 7.5 software architecture, running on a single thread of execution.

TABLE 3: Performance in MFlops/s for multiplying two FP exansions on the GPU; d_x and d_y represent the number of terms in the input expansions and d_r is the size of the computed result. $*$ precision not suported

d_x, d_y, d_r	Algorithm 1	QD
$2, 2, 2$	0.027	0.1043
$1, 2, 2$	0.365	0.1071
$3, 3, 3$	0.0149	$*$
$2, 3, 3$	0.0186	$*$
$4, 4, 4$	0.0103	0.0174
$1, 4, 4$	0.0215	0.0281
$2, 4, 4$	0.0142	$*$
$8, 8, 8$	0.0034	$*$
$4, 8, 8$	0.0048	$*$
$16, 16, 16$	0.001	$*$

From the above mentioned tables we deduce that for small size expansions (2 up to 4 terms) the algorithm is too complex to be efficient. In these cases we suggest the use of a different algorithm, presented shortly in [13] (Algorithm 5 and Figure 4), that takes advantage of the processor's pipeline by avoiding branches. That algorithm is a generalized version of the QD's multiplication combined with a proven renormalization algorithm. It allows for a tight error bound when small expansion results are needed (just slightly larger than the one obtained when using Algorithm 1), but it becomes exponentially worst as the size of the result increases, which limits its practical interest to expansions with at most 4 terms.

ACKNOWLEDGMENTS

The Authors would like to thank Région Rhône-Alpes and ANR FastRelax Project for the grants that supports this activity.

REFERENCES

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008, available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[2] J. Laskar and M. Gastineau, "Existence of collisional trajectories of Mercury, Mars and Venus with the Earth," *Nature*, vol. 459, no. 7248, pp. 817–819, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1038/nature08096

[3] M. Joldes, V. Popescu, and W. Tucker, "Searching for sinks for the hénon map using a multipleprecision gpu arithmetic library," *SIGARCH Comput. Archit. News*, vol. 42, no. 4, pp. 63–68, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2693714.2693726

[4] A. Abad, R. Barrio, and A. Dena, "Computing periodic orbits with arbitrary precision," *Phys. Rev. E*, vol. 84, p. 016701, Jul 2011. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.84.016701

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, 2007, available at http://www.mpfr.org/.

[6] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating-point arithmetic," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, N. Burgess and L. Ciminiera, Eds., Vail, CO, Jun. 2001, pp. 155–162.

[7] S. M. Rump, "Ultimately Fast Accurate Summation," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3466–3502, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1137/080738490

[8] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. 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.

[9] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, P. Kornerup and D. W. Matula, Eds. IEEE Computer Society Press, Los Alamitos, CA, Jun. 1991, pp. 132–144.

[10] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete Computational Geometry*, vol. 18, pp. 305–363, 1997. [Online]. Available: http://link.springer.de/link/service/journals/00454/papers97/18n3p305.pdf

[11] D. M. Priest, "On properties of floating-point arithmetics: Numerical stability and the cost of accurate computations," Ph.D. dissertation, University of California at Berkeley, 1992.

[12] NVIDIA, *NVIDIA CUDA Programming Guide 5.5*, 2013.

[13] M. Joldes, O. Marty, J.-M. Muller, and V. Popescu, "Arithmetic algorithms for extended precision using floating-point expansions," *IEEE Transactions on Computers*, vol. PP, no. 99, p. 1, 2015.