Towards fast and certified multiple-precision libraries

Valentina Popescu

Numéro National de Thèse : 2017LYSEN036

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

École Doctorale : N°512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

Présentée et soutenue publiquement le 6 Juillet 2017 par

Valentina POPESCU

Towards fast and certified multiple-precision libraries

Vers des bibliothèques multi-précision certifiées et performantes

Devant le jury composé de :

Marius CORNEA	Ingénieur de recherche, Intel Corporation	Examinateur
Stef GRAILLAT	Professeur, Paris 6	Rapporteur
Paolo IENNE	Professeur, EPFL	Rapporteur
Mioara JOLDEȘ	Chargée de recherches, CNRS	Co-encadrante
Jean-Michel MULLER	Directeur de recherches, CNRS	Directeur de thèse
Sylvie PUTOT	Professeur, École Polytechnique	Examinatrice
Daniel WILCZAK	Professeur, Jagiellonian University	Examinateur

Contents

List of Figures

List of Tables

Résumé de la thèse

Actuellement la plupart des calculs en virgule flottante (VF) sont faits dans le format «double-précision» et sont conformes à la norme IEEE 754-2008. Cette norme exige l'arrondi correct des opérations arithmétiques de base avec plusieurs modes d'arrondi. Cette exigence améliore la portabilité des codes numériques et rend également possible et —relativement— aisée la construction d'une arithmétique d'intervalles correcte. La majorité des applications nécessitant des calculs haute performance (HPC), déployés sur les architectures parallèles usuelles, bénéficient directement des opérations en VF (en simple- ou double-précision) du fait de leur disponibilité en matériel.

Il existe cependant, de nombreux problèmes numériques qui demandent à faire appel, au moins dans des parties critiques, à une plus grande précision que celle offerte par les formats virgule flottante usuels. Parmi les exemples d'applications possibles, on vise le processus d'itération à long terme des systèmes dynamiques chaotiques. Cela apparaît à la fois dans des problèmes mathématiques (e.g., l'étude des attracteurs étranges comme celui de Hénon, l'analyse des bifurcations et l'étude de la stabilité des orbites périodiques) et dans des applications de mécanique spatiale (e.g., la stabilité à long terme du système solaire). Une autre application possible est le problème dit SDP (pour Semi-definite programming : programmation semi-définie positive) dans lequel si les conditions initiales sont mal posées on a besoin d'une précision plus grande que celle disponible en standard pour obtenir un résultat utilisable.

Une solution est de faire appel à des bibliothèques multi-précision telles que GNU-mpfr (http ://www.mpfr.org), mais elles peuvent parfois être une alternative assez lourde lorsqu'une précision de quelques centaines de bits suffit et que l'on a des exigences fortes de performance. Actuellement, le seul code disponible et facilement portable sur une architecture parallèle est celui de la bibliothèque QD de Bailey, qui propose une arithmétique en double-double (DD) et quad-double (QD), c.à.d., les nombres sont représentés comme la somme non évaluée de 2 ou 4 nombres virgule flottante en double-précision. Il est connu, cependant, que les opérations mises en œuvre dans cette bibliothèque ne sont pas conformes à la norme IEEE 754-2008, ne fournissant pas l'arrondi correct. En outre, les effets des erreurs d'arrondi peuvent être très élevés dans certaines applications, et aucune analyse systématique n'est actuellement disponible pour évaluer la précision effective des résultats obtenus avec de tels formats.

L'objectif majeur de cette thèse a été de développer une bibliothèque multi-précision ciblée pour les architectures hautement parallèles, comme les processeurs graphiques. Pour cela, nous nous sommes intéressés à des algorithmes arithmétiques suffisamment simples et efficaces pour obtenir, pour quelques précisions étendues fixées, des performances élevées, des présentations rigoureuses et des preuves solides. Nous avons proposé d'étendre la précision en représentant les nombres comme des sommes non évaluées de nombres en virgule flottante, aussi appelées «expansions en VF». Cette représentation permet d'utiliser directement les opérations «natives» et hautement optimisées disponibles en matériel (par exemple, dans les processeurs graphiques).

On a développé activement le logiciel CAMPARY—CudA Multiple Precision ARithmetic librarY—, présenté à http ://homepages.laas.fr/mmjoldes/campary/. Cette bibliothèque multi-précision est écrite en CUDA C, une version du langage C adaptée pour les GPUs. Notre implantation est très flexible et peut-être efficacement utilisée à la fois pour les programmes orientés CPU et GPU. Nous avons également évalué les performances de nos algorithmes, en termes de nombre d'opérations flottantes et de bornes d'erreur obtenues. Ils se comparent très favorablement par rapport aux autres bibliothèques existantes.

Chapter 1

Introduction

Even before the first electronic computer was developed in 1946 (ENIAC - *Electronic Numerical Integrator And Calculator*), scientists have tried different ways of approximating real numbers inside a computer, i.e., to find a mapping from the infinite continuous set of real numbers $\mathbb{R}$ to a discrete, finite one. Such representations include the well-known floating-point and fixed-point formats, logarithmic and semi-logarithmic number systems, rational numbers, etc. When choosing between these representations one has to take into account many constraints like speed, accuracy, dynamic range, ease of use and implementation, memory use or power consumption.

By far the most used in modern computers, the floating-point number system is considered to offer a good compromise among the above constraints. It appears that in itself, this idea is very old, dating back to as early as the Babylonians [51]. The first real implementation of such a system was in Zuse's Z1 mechanical computer, in 1938, followed by the modern implementation inside the Z3 electromechanical computer in 1941 (see [15]). For a more complete history see [68, Ch.1] and references therein.

In the early beginnings, each computer manufacturer chose a floating-point system of their liking to implement, but in 1985 the IEEE Standard for Binary Floating-Point Arithmetic [2] was released, that specified various formats, exceptions and basic operations behaviors. While its detailed description will be given in Section 1.1, let us first note that a binary floating-point number of precision p is a number of the form $M \cdot 2^{e-p+1}$, where M is an integer of absolute value less than or equal to $2^p - 1$ and e is an integer such that $e_{\min} \leq e \leq e_{\max}$, where the extremal exponents $e_{\min}$ and $e_{\max}$ are constants of the floating-point format being considered, and with the additional requirement that, unless $e = e_{\min}$, $2^{p-1} \leq |M|$. The standard also specified correct rounding for basic arithmetic operations, i.e., when the result cannot be exactly represented with precision p, the returned result should be as if computed with infinite precision and unlimited range, then rounded to the specified format. A new version of the standard was released in 2008 [38], which also recommended (but did not require) correctly rounded elementary functions.

From the four binary formats defined by the standard, the most commonly used is binary64, i.e., $p = 53$, which gives approximately 15 decimal digits, also known as *double*-precision. This is due not only to its accuracy and performance, but also to its wide availability. The more accurate format, binary128, is not usually implemented in hardware; the only exception that we know of is the IBM Z series of mainframes. The precision provided by the binary64 format is usually enough for numerical computing. For example, in [68], the authors even explain that this is enough for expressing "the distance from the Earth to the Moon with an error less than the thickness of a bacterium".

However this standard precision is not enough for some numerically sensitive problems. Examples include planetary orbit dynamics, such as the long-term stability of the solar system [54],

or supernovas simulations; also, when studying chaotic dynamical systems, like finding sinks in the Hénon Map [46], or iterating the Lorenz attractor [1]. Other examples can be found in experimental mathematics [7, 5], or in some numerically sensitive semidefinite optimization problems which have a very wide range of applications in control theory, quantum chemistry and physics. We will detail some of these applications in Chapter 5.

In such cases, when more than double-precision is required, the situation deteriorates brutally in terms of performance. Arbitrary precision, i.e., the ability of the user to choose the precision for each calculation, is available in software in most computer algebra systems like Maple or Sage. Also GNU MPFR [25] is a general-purpose high-precision arithmetic library that follows the general philosophy of IEEE-754. However, arbitrary precision, which is very useful in general, comes with drawbacks: *(i)* there is a slow-down factor of 10 to 50 compared to a native double computation; *(ii)* the algorithms are finely tuned and difficult to prove formally; *(iii)* the algorithms are not easily portable to highly parallel architectures, such as GPUs or Xeon Phi. All this is because these libraries offer the ability to manipulate numbers with tens of thousands – or even much more – of digits, which requires very complex and non regular arithmetic algorithms, heavily tuned programming, and nontrivial memory management.

In order to harness the availability and efficiency of the hardware implementations of the standard, our approach in this work consists in representing higher precision numbers as **floating-point expansions**. These are unevaluated sums of several floating-point numbers (of different magnitudes). Such a representation is possible thanks to the availability of *error-free transforms*, namely algorithms that allow one to compute the error of a floating-point addition or multiplication exactly, taking the rounding mode into account. For instance, the sum of two floating-point numbers, x and y can be represented *exactly* (in the sense of dyadic numbers) as a floating-point number s which is the correct rounding of the sum, plus another floating-point number e corresponding to the remainder. This will be detailed in Section 1.1.3, for the moment let us just note that under certain assumptions, this decomposition can be computed at a very low cost by a simple sequence like $s = x + y; z = s - x; e = y - z;$. This is actually Algorithm 1, known as Fast2Sum, and more general and sophisticated algorithms exist for a number of related questions and will be detailed throughout this work.

In what follows, we describe a typical example where such extended precision is used in practice. It appears when implementing transcendental functions in mathematical libraries (`libms`), like glibc, Sun libmcr, Intel©libm or CRlibm [19] [1].

Roughly speaking if the input y of a function, say $\sin(y)$, is given with 15 decimal digits of accuracy, then the result is expected to also have 15 digits of accuracy. More specifically, some developers of libms aim for correctly rounded $\sin$ in double-precision. To achieve this, usually, one firstly performs a so-called argument reduction, which allows for the input range to be sufficiently small, such that polynomial approximations are efficient. Such polynomials can be evaluated using only basic arithmetic operations like addition and multiplication. But if these operations are all performed in standard double-precision, it is very difficult to guarantee an intermediary extended accuracy that will allow for a final correctly rounded result in double-precision. Let us explain this in more detail in the following Example 1.0.1, taken from the actual $\sin$ function implementation in CRLibm [70].

Example 1.0.1. *For sine function evaluation, the reduced argument x is obtained by subtracting from the floating-point input y an integer multiple of $\pi/256$. As a consequence, $x \in [-\pi/512, \pi/512] \subseteq [-2^{-7}, 2^{-7}]$. Then, one needs to compute the value of the odd polynomial:*

$$p(x) = x + x^3 \cdot (s_3 + x^2 \cdot (s_5 + x^2 \cdot s_7)),$$

1. The CRlibm library is developed by my research group, AriC in Lyon, France.

which is a polynomial close to the Taylor approximation of the sine function. The coefficients s_1, s_3 *and* s_5 *are represented in binary64 precision arithmetic:* $s_3 = -6004799503160661/2^{55}$, $s_5 = 4803839602528529/2^{59}$, $s_7 = -3660068268593165/2^{64}$.

However, since x *is an irrational number, the implementation of the range reduction needs to return a number more accurate than a binary64, such that the intermediary output accuracy for* $p(x)$ *allows for subsequent correct rounding of* $\sin(x)$. *To overcome this problem, in the CRlibm library [19] the authors represent* x *as the unevaluated sum of two binary64 numbers* $x_h + x_l$, *representation also known as* double-double.

As a numerical example, let $y = 0.5$, *and the corresponding reduced argument* $x = 1/2 - 41\pi/256$. *This is approximated in double-double as the unevaluated sum* $x_h + x_l$, *with* $x_h = -7253486725817229/2^{61}$ *and* $x_l = -508039184604813/2^{112}$.

If one computes directly $p(x_h + x_l)$ *with the following Horner scheme and binary64 precision:*

$$p_{eval}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7)),$$

one obtains a poor accuracy. Note that with this order of operations, the floating-point addition $x_h + x_l$ *returns* x_h, *so the information held by* x_l *is lost. The other part of the Horner evaluation also has a much smaller magnitude than* x_h, *since* $|x| \leq 2^{-7}$, *which gives* $|x^3| \leq 2^{-21}$. *The following evaluation leads to a much more accurate algorithm, since the leftmost addition is performed with an extended precision, namely the above mentioned Fast2Sum algorithm:*

$$\begin{aligned} s &= x_l + (x_h \cdot x_h \cdot x_h \cdot (s_3 + (x_h \cdot x_h \cdot (s_5 + (x_h \cdot x_h \cdot s_7))))), \\ p'_{eval}(x_h + x_l) &= \mathit{Fast2Sum}(x_h, s). \end{aligned}$$

For our numerical example, one obtains $p'_{eval} = -7253474763108583/2^{61} + 82031/2^{79}$. *This allows for* 72 *bits of accuracy in the evaluation of* p *compared with* 54 *for the first evaluation scheme. Note that for both evaluation schemes only standard binary64 operations are used: the second one performs* 2 *more additions than the first one (by executing the Fast2Sum algorithm) and yet, it allows for an accuracy extension by* 33%.

This shows that it possible to compute very accurate values, even in the presence of roundings at the floating-point level, by using only standard precision floating-point arithmetic operations in a clever way. However, proofs of these algorithms get tricky very often. An important objective of this thesis is to generalize algorithms like Fast2Sum, for handling all arithmetic operations (addition, multiplication, division and square root) with numbers represented as unevaluated sums of floating-point values. We aim in the sequel to provide both efficient implementations and to prove tight error bounds.

Concerning efficiency, more than often, practical applications which require extended precision support, are also very computationally expensive and are executed in so-called *High Performance Computing* (HPC) architectures. In this work, we thus aim to tune our implementations of extended precision arithmetic algorithms for accelerators such as *Graphics Processing Units* (GPUs).

Contributions of this work.

In Chapter 2 we "double" the available precision by representing a real number as the unevaluated sum of two floating-point numbers. Though extensive work has been already done in this area, many algorithms have been published without a proof, or with error bounds that are sometimes loose. Thus we revisited the existing algorithms and proposed new ones, providing them with correctness and error bound proofs. This joint work with M. Joldes and J.-M. Muller, led to the article:

1. *Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic* [43], that is currently under revision for publication in the ACM Transactions on Mathematical Software journal.

Chapter 3 focuses on arbitrary precision by looking into algorithms for arithmetic operations using floating-point expansions, i.e., numbers are represented as the unevaluated sum of several (more than two) floating-point numbers. We propose several new algorithms designed to fit different needs a user might have, either very tight error bounds on the results, either "quick-and-dirty" results. This work was presented in:

3. *On the computation of the reciprocal of floating point expansions using an adapted Newton-Raphson iteration* [42], joint work with M. Joldes and J.-M. Muller, published in Proceedings of the 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2014);
2. *Arithmetic algorithms for extended precision using floating-point expansions* [41], joint work with M. Joldes, O. Marty (student intern during collaboration, now engineer at Google, France) and J.-M. Muller, published in the IEEE Transactions on Computers journal;
4. *A New Multiplication Algorithm for Extended Precision Using Floating-Point Expansions* [69], joint work with J.-M. Muller and P. Tang (senior engineer at Intel Corporation), published in Proceedings of the 23rd IEEE Symposium on Computer Arithmetic (ARITH 2016).

In Chapter 4 we explore the possibility of directly parallelizing the arithmetic algorithms by using what we called parallel floating-point expansions. This joint work with S. Collange (researcher at Inria Rennes), M. Joldes and J.-M. Muller, entitled:

5. *Parallel floating-point expansions for extended-precision GPU computations* [16] was published in Proceedings of the 27th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2016).

Finally, in Chapter 5 we present two numerical applications in which CAMPARY proved to be useful. The first application comes from a classical problem in the field of chaotic dynamical systems. Specifically, we search for periodic orbits in the Hénon map, which is a very numerically sensitive problem. Our approach, based on extensive long-term numerical iterations of the map, also needs high-performance computing in order to be tackled. This joint work with M. Joldes and W. Tucker (professor at Department of Mathematics, Uppsala University), entitled:

6. *Searching for Sinks for the Hénon Map Using a Multiple-precision GPU Arithmetic Library* [46], was published in the ACM SIGARCH Computer Architecture News - HEART '14 journal.

Second, we took interest in another problem which needs both higher-precision and high-performance computing: semidefinite programming (SDP) solvers for numerically sensitive problems. We integrated CAMPARY with the already existing SDPA solver, and we provide arbitrary precision not only for the CPU routines, but also GPU support for matrix multiplication. This joint work with M. Joldes and J.-M. Muller, led to the article:

7. *Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming*, which was accepted for publication and is going to be presented at the 24th IEEE Symposium on Computer Arithmetic (ARITH 2017).

In what follows, some preliminary notions are given: we detail floating-point arithmetic in Section 1.1, followed by previously existing related software in Section 1.3. In Section 1.2 we detail the GPU architecture and programming model. In Section 1.4, we give a general overview of CAMPARY and we illustrate its features. At the same time, we aim to introduce in a concrete way the problems that will occupy us in the rest of the this work. CAMPARY was also presented in an extended abstract [45] published in Proceedings of the 5th International Congress on Mathematical Software, ICMS 2016.

1.1 Floating-point arithmetic

It is common knowledge that floating-point numbers are by far the most common used representation of real numbers. This section aims at recalling several basic definitions and concepts of floating-point arithmetic, as defined by the IEEE 754 standard for floating-point arithmetic, and at defining notations that are going to be used throughout this manuscript. Here we mainly focus on the specifications defined by the 2008 release of the IEEE 754-2008 standard [38]. All these concepts have been extensively studied and explained in the Handbook on Floating-Point Arithmetic [68]. Further information can be also found in [28, 36, 49, 52, 80] (this list is far from exhaustive).

A floating-point system is characterized by four integers:

- a radix (or base) $\beta \geq 2$;
- a precision $p \geq 2$ (roughly speaking, p is the number of "significant digits" of the representation);
- two *extremal* exponents $e_{\min}$ and $e_{\max}$ such that $e_{\min} < e_{\max}$ (in all practical cases, $e_{\min} < 0 < e_{\max}$).

Definition 1.1.1. *A finite precision-p floating-point number in such a format is a number x for which there exists at least one representation (M_x, e_x) such that*

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

where

- *the integer e_x is the exponent of x, such that $e_{\min} \leq e_x \leq e_{\max}$*
- *and $M_x \cdot \beta^{-p+1}$ is the significand (sometimes improperly called the mantissa) of x.*

1.1.1 IEEE 754 Standard

In 1985, the first release of the IEEE 754-1985 Standard [2] for floating-point arithmetic was introduced by the *Institute of Electrical and Electronics Engineers* and it was implemented by all the computers produced after that moment. It was a key factor in improving the quality of the computational environments and providing portable code. This initial version considered only binary representations, but two years later, the IEEE 854-1987 Standard [3] for "Radix-Independent" floating-point arithmetic was released, that would also consider decimal representations.

The latest version IEEE 754-2008 [38], encompasses both precision standards and adds some novelties, among them: portability, standardizing the FMA operation, *quadruple*-precision. The standard enforces maximum quality for the basic operations ($+, -, *, /, \sqrt{}$), unique exception handling and the implementation of four rounding modes. In the following we will recall some of its notions and requirements and since scientific computing always uses binary arithmetic, we consider the binary representation only. So from now on by a floating-point number we will understand a binary floating-point number.

Normal and subnormals. From the above definition one can notice that this representation is not unique. Just consider the "toy format" with $p = 5$ and two floating-point numbers $x = 1.1010 \cdot 2^{-1}$ and $y = 0.1101 \cdot 2^{0}$. They are both valid representations of the same number.

We can eliminate the redundancy, by *normalizing* the finite nonzero floating-point numbers. This is done by choosing the representation for which the exponent is minimum (yet larger than or equal to $e_{\min}$). The numbers that satisfy this are called *normalized* floating-point numbers. This type of representation allows for easier expression of error bounds, and it somewhat simplifies the implementation. Two cases may occur:

- when the number is greater than or equal to $2^{e_{\min}}$ and its representation satisfies $2^{p-1} \leq |M_x| \leq 2^p - 1$, we say that it is a *normal* number;
- otherwise, one necessarily has $e = e_{\min}$, and the significand adjusted according to that, with $|M_x| \leq 2^{p-1} - 1$; the corresponding floating-point number is called a *subnormal* number (the term *denormal* may be used also).

A consequence of the normalization in radix 2 is that the significand of a normal number always has the form $M_x = 1.m_1m_2m_3 \ldots m_{p-1}$ and $2^{p-1} \leq |M_x| \leq 2^p - 1$. Also, the significand of a subnormal number always has the form $M_x = 0.m_1m_2m_3 \ldots m_{p-1}$ and $|M_x| \leq 2^{p-1} - 1$. This allows one to save one bit of storage, by applying the "hidden bit" convention.

Even though subnormal numbers have been one of the most controversial parts of the IEEE 754-1985 standard, and they have not always been implemented in hardware (in modern computers they are), they allow for what Kahan calls *gradual underflow*: the loss of precision when numbers converge to zero is slow instead of being abrupt.

According to the above definitions, the smallest positive normal number is $2^{e_{\min}}$; the largest finite floating-point number is $\Omega = (2 - 2^{1-p}) \cdot 2^{e_{\max}}$; and the smallest positive subnormal number is $\alpha = 2^{e_{\min}-p+1}$.

Rounding modes. Sometimes, the result of an operation on floating-point numbers is not exactly representable in the floating-point system being used, so it has to be rounded. One of the most interesting ideas brought out by IEEE 754-1985 is the concept of rounding mode: how a numerical value is rounded to a finite floating-point number.

The four rounding modes defined by the standard (illustrated in Figure 1.1), when applied on a real number x, are:

- round toward $-\inf$: $\mathbf{RD}(\boldsymbol{x})$ is the largest floating-point number less than or equal to x;
- round toward $+\inf$: $\mathbf{RU}(\boldsymbol{x})$ is the smallest floating-point number greater than or equal to x;
- round toward zero: $\mathbf{RZ}(\boldsymbol{x})$ is the closest floating-point number to x that is no greater in magnitude than x (it is equal to $\mathrm{RD}(x)$ if $x \geq 0$, and to $\mathrm{RU}(x)$ if $x \leq 0$);
- round to nearest: $\mathbf{RN}(\boldsymbol{x})$ is the floating-point number that is the closest to x. A tie-breaking rule must be chosen when x falls exactly halfway between two consecutive floating-point numbers. A frequently chosen rule is "ties-to-even", i.e., x is rounded to the only one of these two consecutive floating-point numbers whose significand is even. This is the default mode.

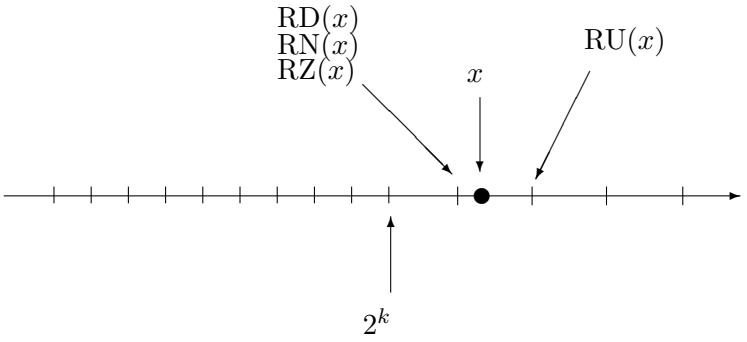


Figure 1.1 – Possible roundings of a real number x in a radix-2 floating-point system. In this example, $x > 0$.

When the exact result of a function is rounded according to a given rounding mode (as if the result was computed with infinite precision and unlimited range, then rounded), one says that the function is *correctly rounded*. The IEEE 754 Standard requires all basic arithmetic operations ($+/-, \times, \div, \sqrt{}$) to be correctly rounded.

The four rounding modes presented above ($\circ$) have the important property that they are *monotonic*, i.e., if $x \leq y$, then $\circ(x) \leq \circ(y)$. Moreover, if x is a floating-point number, then $\circ(x) = x$, which means that when the result of a correctly rounded operation is a floating-point number, we get that exact result.

A classic lemma that applies to any floating-point system with correct rounding and subnormal numbers is Sterbenz's [90] lemma, which implies that, under certain conditions, the subtraction is exact, i.e., the result is a representable floating-point number. This is also exemplified in Example 1.1.3. We do not present the entire proof here, but this lemma is very useful when designing and proving algorithms, and we will make extensive use of it throughout this work.

Lemma 1.1.2 (Sterbenz Lemma [90])**.** *Let x and y be two finite binary floating-point numbers. If*

$$\frac{y}{2} \leq x \leq 2y,$$

then $x - y$ is a floating-point number, so that $\circ(x - y) = x - y$ exactly, where $\circ$ is the rounding function.

Example 1.1.3. *Let $x = 0.713938$ and $y = 1.213694$ be two binary32 floating-point numbers. When computing $\mathrm{RN}(x - y)$ we obtain -0.499756, which is also representable as a binary32 number.*

A slightly more general result is that if the exponent of $x - y$ is less than or equal to the minimum between e_x and e_y, the exponents of x and y, respectively, then the subtraction is performed exactly. Another example of exact addition is given by Hauser [31], who showed that when a gradual underflow occurs it does not necessarily mean an inaccurate result [2]:

Lemma 1.1.4 (Hauser Lemma [31])**.** *Let x and y be two floating-point numbers. If $\mathrm{RN}(x + y)$ is subnormal, then $\mathrm{RN}(x + y) = x + y$ exactly.*

It is worth recalling that some multiplications and divisions are also performed exactly. A straightforward example is multiplication/division by a power of 2, which, assuming that no overflow or underflow occurs, can be reduced to an add/subtract to the exponent. Another example is multiplication of numbers with known number of trailing zero bits in the lower-order part of the significand. This property is at the heart of Dekker's multiplication algorithm (see Algorithm 5, Section 1.1.3). It is also very useful for reducing the range of inputs when evaluating elementary functions.

Special floating-point data and exceptions. Some data cannot be expressed as a normal or subnormal number. A straightforward example is the number zero, that requires a special encoding. In order to achieve a "closed" floating-point system in which any machine operation is well specified, the standard defines several not fully "numeric" data used to cope with values that are undefined or those of different magnitudes than the representable ones. These are:

- the **NaN** (Not a Number) value, which is used to deal with invalid operations (e.g., $\sqrt{-5}$ or $0/0$);
- two signed infinities, $+\mathbf{Inf}$ and $-\mathbf{Inf}$;
- two signed zeros, $+\mathbf{0}$ and $-\mathbf{0}$.

2. Notice that this is a consequence of the fact that the exponent of a subnormal number is the smallest possible exponent, $e_{\min}$.

Sometimes an exception can be signaled along with the result of an operation under the form of a status flag and/or some trap mechanism. The standard defines five such exceptions:

- **Overflow** is signaled when the result of an operation or function, rounded as if there was no exponent bound, is of absolute value strictly larger than Ω defined above;
- **Underflow** is opposite to the former; is signaled when the result is smaller than $2^{e_{\min}}$ and inexact;
- **Invalid** exception (an input is invalid for the function) is raised when the result is a NaN (e.g. $\infty - \infty, 0/0, \sqrt{-1}$);
- **DivideByZero** exception is self explanatory; signaled when an exact infinite result is defined for a function on finite inputs (e.g. $1/0, \log +0$);
- **Inexact** is signaled when the result is not exactly representable and it has to be rounded.

Standard formats. One of the main goals of the IEEE 754-2008 Standard for floating-point arithmetic was to reduce the implementation choices during hardware design. As we explained above, in radix 2, the leftmost bit of the significand of a finite normal floating-point number is always a 1, or 0 if the number is subnormal. The standard requires for specified formats the use of the "hidden bit" convention, i.e., the information on the type of floating-point number should be stored in the exponent field, allowing to store only the last $p-1$ significant bits of the significand. Also it is required that $e_{\min}$ should be $1 - e_{\max}$ for all formats.

Both binary and decimal formats are defines, but the standard only requires the availability of one of the following: binary32, binary64, binary128, decimal64, decimal 128. In this work we focus only on binary floating-point arithmetic. The four main binary formats defined are given in Table 1.1, though the most commonly available are the two presented in Figure 1.2.

Table 1.1 – Main parameters of the binary formats specified by the IEEE 754-2008 standard.

Name	*binary16* N/A	*binary32* (single)	*binary64* (double)	*binary128* (quad)
Storage width	16	32	64	128
Precision p	11	24	53	113
$e_{\max}$	$+15$	$+127$	$+1023$	$+16383$
$e_{\min}$	-14	-126	-1022	-16382

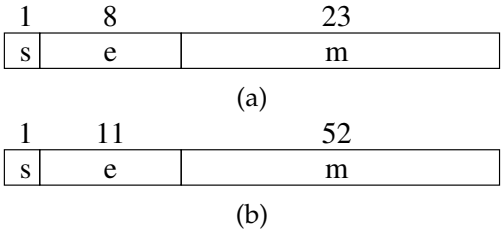


Figure 1.2 – Most commonly available floating-point formats: (a) binary32 (*single*-precision) and (b) binary64 (*double*-precision).

The Fused Multiply-Add Instruction. The Fused Multiply-Add (FMA) instruction was first introduced in 1990 on the IBM RS/6000 processor [37, 66], but it was included in the IEEE 754

standard only in 2008, on its latest release. Even though at first it was not available on widely spread processors, with the release of the Intel Haswell and AMD Bulldozer architectures, this is not the case anymore.

This instruction evaluates a floating-point multiplication and a summation (an expression of the form $a \cdot b + c$) using only one rounding. By retaining full precision in the intermediate stage it improves upon the classic Multiply-Add (MAD) instruction that executes the same operations performing two roundings. That is, if $\circ$ is the rounding function applied, the two instructions perform as follows:

$$\begin{aligned} \mathrm{fma}(a, b, c) &= \circ(a \cdot b + c); \\ \mathrm{mad}(a, b, c) &= \circ(\circ(a \cdot b) + c). \end{aligned}$$

The FMA is extremely helpful for the design of some arithmetic algorithms:

- it facilitates correctly rounded software division [14, 17, 48, 62];
- it makes some calculations (especially dot products and polynomial evaluations) faster and, in general, more accurate [18, 62];
- as explained in Section 1.1.3, it makes it possible to easily get the exact product of two floating-point numbers.

1.1.2 Error handling

Relative error. In a binary precision-p floating-point arithmetic that supports correct rounding, when approximating a real number $x \in \mathbb{R}$, with $x \neq 0$, by $\circ(x)$, with $\circ$ being the rounding mode, then the relative error that occurs satisfies:

$$\left| \frac{x - \mathrm{RN}(x)}{x} \right| \leq 2^{-p},$$

if the rounding function is round-to-nearest (the inequality is strict in round-to-nearest ties to even), and

$$\left| \frac{x - \circ(x)}{x} \right| < 2^{1-p}$$

with the other rounding functions, assuming that no underflow [3] /overflow occurs. When $\circ(x) = x = 0$, we consider that the relative error is 0.

If x is subnormal, the relative error can become very large (it can be close to 1). In that case, we have a bound on the *absolute* error due to rounding:

$$|x - \mathrm{RN}(x)| \leq \frac{1}{2} 2^{e_{\min} - p + 1}$$

in round-to nearest mode, and

$$|x - \circ(x)| < 2^{e_{\min} - p + 1}$$

if $\circ$ is one of the directed rounding modes.

3. Let us say, as does the IEEE 754 standard, that an operation underflows when the result is subnormal and inexact.

ULPs. When expressing errors of "nearly atomic" functions (arithmetic operations, elementary functions, small polynomials, sums, dot products, etc.) it is advisable (frequently more accurate) to do it in terms of what we would intuitively define as the "weight of the last bit of the significand."

If x is a binary floating-point number and is not an integer power of 2, the function $\mathrm{ulp}(x)$ (for *unit in the last place*) denotes the magnitude of the least significant bit of M_x. I.e., if,

$$x = \pm\, m_0.m_1m_2\ldots m_{p-1} \cdot 2^{e_x},$$

then $\mathrm{ulp}(x) = 2^{e_x-p+1}$.

A definition of $\mathrm{ulp}(x)$ for all real x is desired. In the literature [28, 30, 50, 62, 80] there are several slightly different definitions, but they all coincide as soon as x is not extremely close to a power of 2 (for more details refer to [68]). In this work we consider only the definition given by Goldberg [28], that has been generalized by Cornea, Golliver, and Markstein [17, 62] and that follows.

Definition 1.1.5. *Let $x \in \mathbb{R}$. If $|x| \in [2^{e_x}, 2^{e_x+1})$, where e_x is exponent of x, then $\mathrm{ulp}(x) = 2^{\max(e_x,e_{\min})-p+1}$.*

It satisfies the following property:

Property 1.1.6. *Let real $x \in \mathbb{R}$ and X an exact floating-point number. Considering Definition 1.1.5 it holds:*

$$|X - x| < \tfrac{1}{2}\,\mathrm{ulp}(x) \Rightarrow X = \mathrm{RN}(x); \tag{1.1}$$
$$|X - x| < \tfrac{1}{2}\,\mathrm{ulp}(X) \Rightarrow X = \mathrm{RN}(x); \tag{1.2}$$
$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \tfrac{1}{2}\,\mathrm{ulp}(x); \tag{1.3}$$
$$X = \mathrm{RN}(x) \Rightarrow |X - x| \leq \tfrac{1}{2}\,\mathrm{ulp}(X). \tag{1.4}$$

The following lemma, which appears as an immediate consequence of Property (2.16) in [85], holds and the proof is actually very simple.

Lemma 1.1.7. *(see Property (2.16) in [85]) Let x and y be floating-point numbers, and let $s = \mathrm{RN}(x+y)$. If $s \neq 0$ then*

$$|s| \geq \max\left\{\frac{1}{2}\,\mathrm{ulp}(x), \frac{1}{2}\,\mathrm{ulp}(y)\right\}.$$

Proof. Without loss of generality, assume $|x| \geq |y|$, so that $\mathrm{ulp}(x) \geq \mathrm{ulp}(y)$. The number $|x+y|$ is the distance between x and $-y$. Hence, since $x \neq -y$ (otherwise s would be 0), $|x+y|$ is larger than or equal to the distance between x and the floating-point number nearest to x, which is larger than or equal to $\frac{1}{2}\,\mathrm{ulp}(x)$. Therefore $|\mathrm{RN}(x+y)| = \mathrm{RN}(|x+y|) \geq \mathrm{RN}(\frac{1}{2}\,\mathrm{ulp}(x)) = \frac{1}{2}\,\mathrm{ulp}(x)$. □

A similar concept is that of *unit in the last significant place*, denoted by uls, which, roughly speaking, gives the weight of the last non-zero bit of the significand. The formal definition follows.

Definition 1.1.8. *Let $x = M_x \cdot 2^{e_x-p+1}$ be a binary precision-p floating-point number. If $|x| \in [2^{e_x}, 2^{e_x+1})$, then $\mathrm{uls}(x)$, for $x \neq 0$, is the only power of 2 such that x is an odd integer times that power of 2.*

In Figure 1.3 we give a graphical representation of the above two concept.

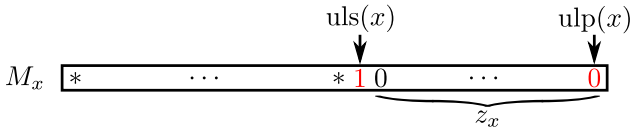


Figure 1.3 – Graphical representation of ulp vs. uls.

Unit roundoff. Another useful notion, closely related to the notion of ulp, is the notion of *unit roundoff*, also called *machine epsilon*.

Definition 1.1.9. *The* unit roundoff **u** *of a binary precision-p floating-point system is defined as*

$$u = \begin{cases} \frac{1}{2}\,\mathrm{ulp}(1) &= 2^{-p} \text{ in round-to-nearest mode,} \\ \mathrm{ulp}(1) &= 2^{1-p} \text{ in directed rounding modes.} \end{cases}$$

The two following lemmas referring to the unit roundoff are considered classical, this is why we do not include the proof here (for details see [68]).

Lemma 1.1.10. *Let $x \in \mathbb{R}$. If $|x| \leq 2^k$, where k is an integer, then*

$$|\mathrm{RN}(x) - x| \leq \frac{u}{2} \cdot 2^k.$$

Lemma 1.1.11. *Let $x \in \mathbb{R}$. There exist ϵ_1 and ϵ_2, both of absolute value less than or equal to u, such that*

$$\mathrm{RN}(x) = x \cdot (1 + \epsilon_1) = \frac{x}{1 + \epsilon_2}.$$

1.1.3 Error-free transforms: computing the error

Let x and y be two precision-p binary floating-point numbers, and $s = \mathrm{RN}(x + y)$. It can be shown that if the addition of x and y does not overflow, the rounding error that occurs, namely $(x + y) - s$, is exactly representable by a floating-point number of the same format. [4] The same thing holds for multiplication, with the condition that $e_x + e_y \geq e_{\min} + p - 1$, where e_x and e_y are the exponents of x and y, respectively. Interestingly enough, that error can be computed using the algorithms presented in what follows, that employ only basic operations ($+, \times$). The full correctness proofs of the algorithms can be consulted in the Handbook of Floating-Point Arithmetic [68].

Addition. For computing the error of a floating-point addition we have two possible algorithms. The first one is Fast2Sum (Algorithm 1 and Theorem 1.1.12) that computes the result using only 3 basic operations. It first appeared as part of a summation algorithm, called "Compensated sum method," due to Kahan [47], and was later published by Dekker [22].

Algorithm 1 – Fast2Sum (x, y).

$s \leftarrow \mathrm{RN}(x + y)$
$z \leftarrow \mathrm{RN}(s - x)$
$e \leftarrow \mathrm{RN}(y - z)$
return (s, e)

4. Beware: that property is not always true with rounding functions different from RN.

Theorem 1.1.12. *Let x and y be precision-p floating-point numbers that satisfy $e_x \geq e_y$, where e_x and e_y are the exponents of x and y, respectively. Algorithm 1 computes two floating-point numbers s and e such that:*

- $s + e = x + y$ *exactly;*
- $s = \mathrm{RN}(x + y)$.

The relationship between the exponents of x and y might be difficult to check, but it will always be satisfied if $|x| \geq |y|$.

If there is no preliminary knowledge on the order of magnitude of the input numbers, there exists another algorithm, due to Knuth [51] and Møller [64], called 2Sum (Algorithm 2 and Theorem 1.1.13) that computes the exact same result, but it uses 6 floating-point basic operations. It was actually showed by Kornerup et.al. [53] that this algorithm is optimal in terms of number of operations, if branch instructions cannot be used.

Algorithm 2 – 2Sum (x, y).

$s \leftarrow \mathrm{RN}(x + y)$
$x' \leftarrow \mathrm{RN}(s - y)$
$y' \leftarrow \mathrm{RN}(s - x')$
$\delta_x \leftarrow \mathrm{RN}(x - x')$
$\delta_y \leftarrow \mathrm{RN}(y - y')$
$e \leftarrow \mathrm{RN}(\delta_x + \delta_y)$
return (s, e)

Theorem 1.1.13. *Let x and y be precision-p floating-point numbers. Algorithm 2 computes two floating-point numbers s and e such that:*

- $s + e = x + y$ *exactly;*
- $s = \mathrm{RN}(x + y)$.

Notice that the two algorithms are mathematically equivalent in the sense that the 2Sum algorithm can always be replaced by a preliminary comparison followed by a possible swap of the operands and the Fast2Sum algorithm. Until recent years the penalty due to a wrong branch prediction when comparing two numbers was more costly than 3 additional basic operations. In modern CPU processors, e.g. the AMD Bulldozer or the Intel Haswell, due to highly optimized branch prediction, this is not the case anymore. Though, in the context of Graphics Processing Units (GPUs) branches are still costly and it is preferable to avoid them if possible (see Section 1.2).

Multiplication. Computing the rounding error of a floating-point multiplication is straightforward when an FMA instruction is available. In fact, unless underflow/overflow occurs, the algorithm 2ProdFMA (Algorithm 3 and Theorem 1.1.14) does this using only 2 floating-point basic operations.

Algorithm 3 – 2ProdFMA (x, y).

$\pi \leftarrow \mathrm{RN}(x \cdot y)$
$e \leftarrow \mathrm{fma}(x, y, -\pi)$
return (π, e)

Theorem 1.1.14. *Let x and y be precision-p floating-point numbers, such that $e_x + e_y \geq e_{\min} + p - 1$, where e_x and e_y are the exponents of x and y, respectively. Algorithm 3 and Algorithm 5 compute two floating-point numbers π and e such that:*

- $\pi + e = x \cdot y$ *exactly;*
- $\pi = \mathrm{RN}(x \cdot y)$.

For cases in which an FMA instruction is not available, the best known algorithm for computing the rounding error of a multiplication is Dekker's product [22] that is mathematically equivalent to Algorithm 3 (it also satisfies Theorem 1.1.14), but it is much more costly.

This algorithm is possible by means of an algorithm due to Veltkamp [94, 95] that can "split" a precision-p floating-point number x into two floating-point numbers x_h and x_ℓ such that, for a given integer $t < p$ and using a floating-point constant $C = 2^t + 1$, the significand of x_h fits in $p - t$ digits, the significand of x_ℓ fits in t digits, and $x = x_h + x_\ell$ exactly. The Split algorithm is presented in Algorithm 4.

Algorithm 4 – Split (x, t).

Constant: $C = 2^t + 1$
$\gamma \leftarrow \mathrm{RN}(C \cdot x)$
$\delta \leftarrow \mathrm{RN}(x - \gamma)$
$x_h \leftarrow \mathrm{RN}(\gamma + \delta)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$
return (x_h, x_ℓ)

When used in a binary floating-point system the algorithm has the following property ([22, 9]):

Property 1.1.15. *Algorithm 4 satisfies:*

- *if $C \cdot x$ does not overflow, no other operation will overflow;*
- *there is no underflow problem: if x_ℓ is subnormal, the result still holds;*
- *the significand of x_h fits in $p - t$ bits;*
- *the significand of x_ℓ actually fits in $t - 1$ bits.*

The full multiplication algorithm given by Dekker is presented in Algorithm 5. The first step is to split each of the operands x and y into two floating-point numbers, the significand of each of them being representable with $\lfloor p/2 \rfloor$ or $\lceil p/2 \rceil$ bits only. The underlying idea is that the pairwise products of these values should be exactly representable. Then these pairwise products are added.

Algorithm 5 – Dekker (x, y).

Constant: $t = \lceil p/2 \rceil$
$(x_h, x_\ell) \leftarrow \mathrm{Split}(x, t)$ //using Alg. 4
$(y_h, y_\ell) \leftarrow \mathrm{Split}(y, t)$
$\pi \leftarrow \mathrm{RN}(x \cdot y)$
$t_1 \leftarrow \mathrm{RN}(-\pi + \mathrm{RN}(x_h \cdot y_h))$
$t_2 \leftarrow \mathrm{RN}(t_1 + \mathrm{RN}(x_h \cdot y_\ell))$
$t_3 \leftarrow \mathrm{RN}(t_2 + \mathrm{RN}(x_\ell \cdot y_h))$
$e \leftarrow \mathrm{RN}(t_3 + \mathrm{RN}(x_\ell \cdot y_\ell))$
return (π, e)

The algorithm was analyzed by Boldo [9] who showed that in any case,

$$|xy - (\pi + e)| \leq \frac{7}{2} 2^{e_{\min} - p + 1}.$$

Dekker's multiplication algorithm requires 17 floating-point operations: 7 multiplications, and 10 additions/subtractions. This may seem a lot, compared to the 6 floating-point additions/subtractions required by the 2Sum algorithm (Algorithm 2). Yet an actual implementation of Algorithm 5 will not be $17/6$ times slower than an actual implementation of 2Sum, since many operations are independent and they can be performed in parallel or in pipeline if the underlying architecture supports it.

Since the two algorithms for computing the exact result of a multiplication are mathematically equivalent, throughout our work we will call algorithm 2Prod (Algorithm 6) that choses between 2ProdFMA and Dekker's product depending on the availability of an FMA instruction.[5] The choice is done using a *static if* instruction that is treated at compilation time, hence it does not slow down computations.

Algorithm 6 – 2Prod (x, y).

```
#if defined FP_FAST_FMA
  (π, e) ← 2ProdFMA(x, y)
#else
  (π, e) ← Dekker(x, y)
#endif
return (π, e)
```

The error-free transforms presented here are basic bricks for doing computations using the multiple-term extended precision format and we are going to use them extensively in the following chapters.

1.2 Graphics Processing Units - GPUs

Today, Graphics Processing Units (GPUs) represent an important hardware development platform for many problems where massive parallel computations are needed. Even though initially they were developed as highly specialized integer only processors for real time image rendering, they gradually evolved towards more programmability and increasingly powerful arithmetic capabilities.

In 2003, along with the appearance of some high-level shading languages and programming interfaces like Microsoft's DirectX, the development of General-Purpose GPU Computing (GPGPU) started. Despite all efforts, the GPUs were still difficult to program and the program complexity was much higher than for the CPU. In 2007 programmability was greatly improved by the appearance of Software Development Kits (SDK) like Nvidia's C-language based CUDA platform, followed by the Khronos Group OpenCL in 2009.

Soon after the two main vendors, Nvidia and ATI (now AMD), also started producing specialized computing cards, that would offer more arithmetic capabilities. This was one of the main turning points in the development of parallel computing. Our work targets Nvidia's CUDA based GPUs, so we continue by detailing them.

1.2.1 CUDA Architecture

Computer Unified Device Arichitecture – CUDA – is the hardware and software architecture that enables Nvidia GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages.

5. During operation count we will consider that an FMA is available.

The parallelization power that the GPU offers is closely related to its structure; it can be seen as a highly multi-threaded SIMD (Sigle Instruction Multiple Data) architecture [27]. One GPU is composed of more that one Streaming Multiprocessors (SM), each with several cuda cores. Each SM has its own cache memory and a shared memory that allows communication between the cuda cores. The communication between the SMs and with the CPU is done through the global memory. A visual description of this structure is given in Figure 1.4.

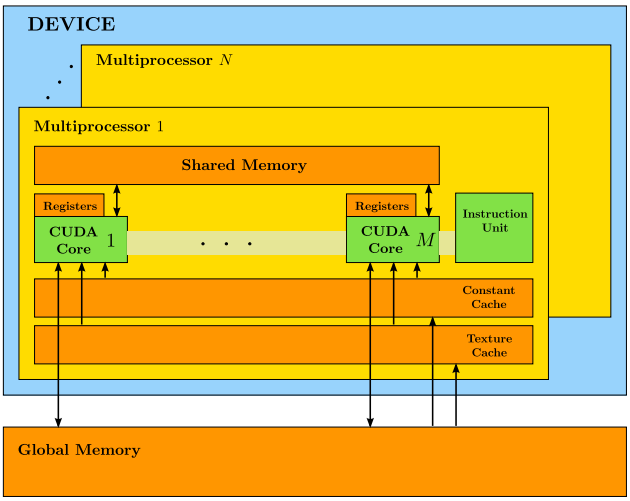


Figure 1.4 – Graphical representation of a general GPU's structure.

Execution model. Programmers describe compute kernels as a single cuda program run by many fine-grained threads. The compiler and hardware scheduler groups these into thread blocks and grids of thread blocks. In detail the three concepts are:

1. A **thread** executes an instance of the kernel, and has a thread ID within its thread block, a program counter, registers and per-thread private memory. The private memory is used for register spills, function calls, and C automatic array variables.
2. A **thread block** is a set of concurrently executing threads that can cooperate among themselves through shared memory and barrier synchronization. It has a block ID within its grid. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms.
3. A **block grid** is an array of thread blocks that execute the same kernel, communicate through global memory (see Figure 1.4) and synchronize between dependent kernel calls. Grids share results in global memory space after kernel-wide global synchronization.

A graphical representation of the cuda execution model is presented in Figure 1.5.

Hardware execution. The thread hierarchy presented above maps to the hierarchy of processors of the GPU as shown in Figure 1.6: a GPU executes one or more block grids; a SM executes one or more thread blocks; and cuda cores inside the SM execute threads.

A concept that we did not mention yet is that of a **warp**. A thread block can contain many threads, but they are executed by the SM in groups (usually 32 threads) called warps. While programmers can generally ignore warp execution for functional correctness, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses. Also, threads inside a warp run in lockstep and share a single control flow. This allows for *warp-synchronous* programming that we are going to detail later on.

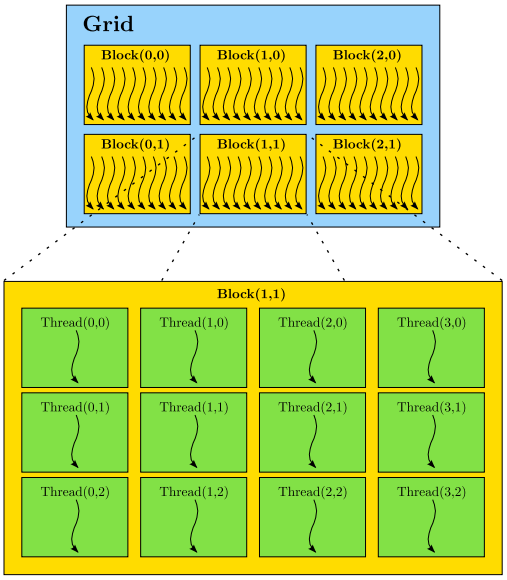


Figure 1.5 – Graphical representation of the GPU's thread execution layout.

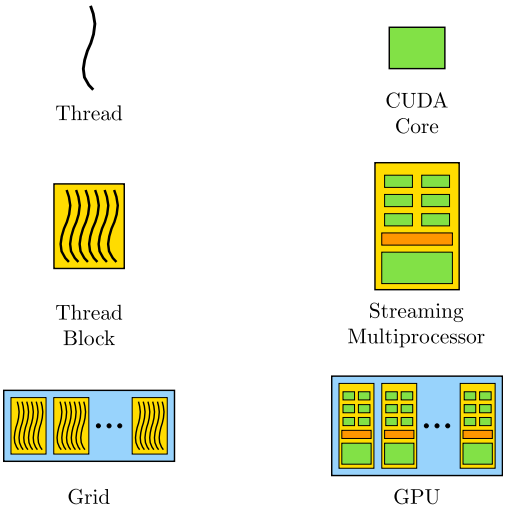


Figure 1.6 – Graphical representation of the GPU's execution model.

1.2.2 Floating-point units

Binary floating-point units appeared in 2002 in the GPUs of both main vendors, ATI and Nvidia. In the first implementation, addition and multiplication were incorrectly rounded: instead of rounding the exact sum or product, these implementations typically rounded a $p + 2$-bit intermediate result to the output precision of p bits.

Between 2007 and 2009 the floating-point support greatly improved. From there on GPUs support binary32 and binary64 precisions, with correctly rounded addition and multiplication. Also, comparing to prior generation GPUs that flushed subnormal operands and results to zero, incurring a loss of accuracy, todays GPUs support subnormal numbers by default in hardware.

They implement as well all four IEEE specified rounding modes and offer support for dynamic rounding mode changing, avoiding pipeline flushing and time penalties. The cuda architecture even offers correctly rounded FMAs (see Section 1.1). Each cuda core can perform one binary32

FMA operation in each clock period and one binary64 FMA in two clock periods. It is worth mentioning that GPUs also include hardware acceleration of some elementary functions.

1.2.3 Programming model

In what follows we are going to give a short introduction to CUDA C programming. For more details refer to the Cuda Programing Guide [78].

The CUDA C programing language is an extension of the C++ programing language. The difference between the CPU and GPU code is done through the use three specifiers:

- `__host__` specifies the code dedicated to the CPU;
- `__global__` specifies the kernel code;
- `__device__` specifies the GPU dedicated functions.

A function can be declared using both `__host__` and `__device__` specifiers.

A kernel is defined as a `void` function. The basic call syntax is `KernelName<<< Blocks, Threads >>>`, where:

- `Blocks` is the number of thread blocks per grid; it can be of type `int` or `dim3`;
- `Threads` is the number of threads per thread block; same types as above can be used.

The communication between the host and the device happens exclusively through global memory. Linear memory is managed by the host typically using the functions:

- `cudaMalloc(void** ptr, size_t size);`
- `cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind);`
- `cudaFree(void* ptr);`

Through `cudaMemcpyKind kind` one specifies the data transfer direction: `HostToHost`, `HostToDevice`, `DeviceToHost`, or `DeviceToDevice`.

The shared memory of the device is declared in device code using the `__shared__` specifier. The size of the shared memory block to use can also be given (in number of bytes) as a third parameter in the kernel call.

As mentioned before, a kernel is executed by many parallel threads. Each thread has access to four built-in variables that define it's position in the execution grid:

- `gridDim` gives the number of thread blocks launched;
- `blockIdx` returns the block ID inside the grid;
- `blockDim` gives the number of threads per block;
- `threadIdx` returns the thread ID inside its block;

Warp-synchronous programming. At execution time, the threads are grouped into so-called warps. Threads inside a warp run in lockstep and share a single control flow, and their instructions are executed on SIMD units (cuda cores), with one thread per lane. This type of programing allows for an implicit SIMD programming, that is particularly efficient starting with the Kepler architecture [77]. We are going to take advantage of it in Chapter 4.

Warp vote instructions perform boolean reductions across all threads within a warp. For instance, they can check whether a condition holds for all the threads, or for any of the threads of the warp. The `__any` function computes the logical OR of a warp-sized vector of predicates and broadcast it to all elements.

Warp shuffle instructions allow arbitrary communication between threads in a warp, without having to go through memory. They are analogous to shuffle or permute instruction in explicit SIMD instruction sets [24].

- `shfl_up(x, n, R)` and `shfl_down` shift components upward or downward, respectively, by n positions within each group of threads of size R, where R is less than or equal to the warp size;
- `shfl` reads a variable from a specific thread and broadcasts it to all other threads in the warp.

The code written in CUDA C is compiled with Nvidia's `nvcc` compiler, that separates the host code (compiled with the available `gcc` compiler) from the device one.

1.3 Multiple-precision libraries

There are mainly two ways of representing numbers in higher precision (see Figure 1.7):

(i) the *multiple-digit* representation, in which numbers are represented by a sequence of possibly high-radix digits coupled with a single exponent ($M \cdot 2^t$);

(ii) the *multiple-term* representation, in which a number is expressed as the unevaluated sum of several standard floating-point numbers, often called a floating-point *expansion* ($u_0 + u_1 + \cdots + u_{n-1}$) (a thorough definition will be given in Chapter 3).

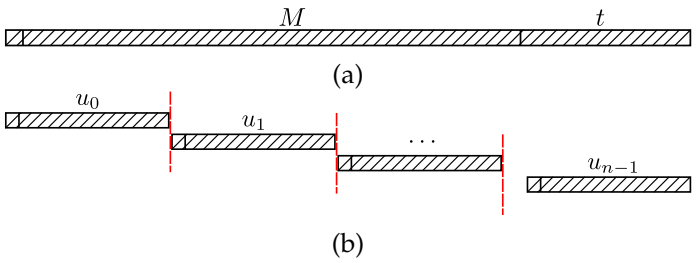


Figure 1.7 – Graphical representation of the (a) multiple-digit vs. the (b) multiple-term approach.

Note that the extended precision achieved when using an expansion with n terms of precision-p is not exactly the same as np bits of significant in the multiple-digit representation. In the multiple-term approach one can sometimes represent a wider precision by skipping some intermediary zero bits [68, Chap.14].

There exist many mathematical software that include multiple-precision facilities, like Maple, Mathematica, Sage, etc., but in what follows we are concerned only with stand alone high-precision libraries.

A well know arithmetic library that uses the first approach is the GNU MPFR [25]. This is an open-source C library based on the integer level of the GNU MP library (GMP) [29]. It provides not only arbitrary precision, but also correct rounding for each atomic operation. Using this library one can use extended precision for up to millions of bits (the only real constraint is the available memory). Its generality makes it a heavy alternative (in terms of speed and memory consumption) when only a few hundred bits are required. Another drawback of MPFR is that it is not ported on the GPU, and because it implements very complex arithmetic algorithms that employ non-trivial memory management, this would be a very difficult task, close to impossible (at least for now). Based on it, some of the same people developed GNU MPFI [83], which is a multiple-precision interval arithmetic library.

Another library that uses the multiple-digit representation for extending the available precision is ARPREC [6], which has been ported to GPUs under the name GARPREC [61]. In their

approach, the authors use the binary64 for representing the significant of an integer, real or complex number.

In [74] the CUMP library is presented, which targets exclusively GPUs. This library is based on the low-level integer arithmetic routines of GMP, and uses the 64-bit integer arithmetic internally on the GPU instead of the binary64 arithmetic used by GARPREC. On the NVIDIA Tesla C2050, CUMP is reported to be up to 2.6 times faster than GARPREC. For more details concerning integer multiple-precision and earlier work for GPUs without binary64 hardware support we refer to related works given in [74].

CUMP and GARPREC libraries were both tuned for big array operations where the data is generated on the CPU, and only the operations are performed on the GPU. They use an interval memory layout where the limbs of the multiple-precision numbers are interleaved. Specifically, for an array of n multiple-precision numbers, each with m limbs, the jth limb of the ith number is stored in the position $jn+i$ in the array. This format is best suited for operations with large arrays of multiple-precision numbers, since it favors coalesced accesses of off-chip memory.

When looking for libraries that employ the second approach, the multiple-term representation, the options are not as many. Even though the technique was often used in compensated algorithms or intermediate calculations, not many per-say libraries are available. One of the first software available was Brigg's *doubledouble* library [13], which extended the available precision to a double-double number, using two binary64 numbers (see Chapter 2). This library is no longer maintained and on his website Briggs states, and we quote: "I no longer support or recommend this software".

The most well known software that uses the second approach is Hida, Li, and Bailey's QD library [34, 35] that offers supports for double-double and quad-double computations, i.e., a number is represented as the unevaluated sum of two or four binary64 numbers, which is equivalent to roughly up to 212 bits of significand. This has been ported on the GPU under the name of GQD [61]. It is well known that the algorithms employed do not come with correctness proofs of error bound proofs.

After analyzing the available options, we concluded that there is a demand for algorithms for arithmetic operations using floating-point expansions with arbitrary precision, that are sufficiently simple yet efficient, and for which effective error bounds and thorough proofs are given. Our goal is to implement a library that can be used on both CPU and GPU.

1.4 CAMPARY - Features and implementation

An overview on the implementation features was published in the extended abstract [45].

As already mentioned, one of the main goals of this thesis was to develop a multiple-precision arithmetic software library. To this end we developed CAMPARY (CudA Multiple Precision ARithmetic LibrarY), an open source library distributed under the GNU General Public License as published by the Free Software Foundation, that is freely available at `http://homepages.laas.fr/mmjoldes/campary/`.

- a C++ version, compilable with `g++`, that can be used on the CPU;
- a CUDA C version, compilable with `nvcc`, that can be used both on GPU [6] and CPU, except for the parallel algorithms (see Chapter 4).

For each of these versions we offer support for extending both the binary64 and the binary32 formats, even though the latter one is mostly useful for GPU use, since some architectures offer

6. Compute capability 2.0 or greater for the sequential algorithms or at least 3.0 for the parallel ones.

optimizations only for this format. The constraint on the expansion size is given by the exponent range of the underlying floating-point format:

- for binary64 (exponent range $[-1022, 1023]$) the maximum expansion size is 39;
- for binary32 (exponent range $[-126, 127]$) the maximum expansion size is 12.

Since all these different versions are equivalent from a design and algorithmic point of view we will continue by exemplifying the binary64 GPU version.

One of the most important design decisions that we had to take was how to define the precision, i.e., the expansion size. We decided on using templates, because it offers flexibility to the implementation. The exact definition is:

```
template <int prec>
class multi_prec{
  private:
    double data[prec];
  public:
    ...
}
```

For allocating and/or initializing a *multi_prec* object we provide constructors along with random initialization functions and "visualization" functions like pretty print. We also provide low level routines for data encapsulation and overloaded operators (to list a few: arithmetic and assignment operators, relational operators, etc.). This makes the code robust, but in the same time flexible and easy to use. For example, for computing the sum of two *multi_prec* values one can use the code:

```
multi_prec<3> a; a.randomInit_ulp();
multi_prec<3> b(a + 4.25);
multi_prec<3> c = a + b;

a.prettyPrint();
b.prettyPrint();
c.prettyPrint();
```

That will output:

```
Prec = 3
   Data[0] = 6.803755e-01
   Data[1] = 5.960464e-08
   Data[2] = 2.924931e-15
Prec = 3
   Data[0] = 4.930376e+00
   Data[1] = -5.960464e-08
   Data[2] = -6.277831e-16
Prec = 3
   Data[0] = 5.610751e+00
   Data[1] = -1.192093e-07
   Data[2] = -1.255566e-15
```

The functions that implement the arithmetic operations are declared as templated friend functions[7], in order to keep the code of the class "clean". For each function that receives as input

7. Only the signature is included in the class, while the full body function is external.

multi_prec objects we also include the equivalent with a *multi_prec* object and a binary64 value as input.

Since fully certified algorithms usually come with a performance cost, we allow trade-off between proven output accuracy in the worst case versus highly efficient average case by offering two levels of algorithms. The library includes two separate files, that define the same class with the same functionalities, and the user has the choice of which one to use, depending on the time and accuracy constraints of one's problem.

1. The *multi_prec_certif.h* file implements the algorithms for all basic operations: addition/-subtraction, multiplication, reciprocal/division, and square root, according to the algorithms presented in Chapter 3, with specialized templates for double-double operations, using algorithms from Chapter 2. These algorithms come with rigorous correctness proofs and relatively tight error bounds. For example, for multiplying a double-double number with a floating-point one, we can prove that, using an algorithm that takes 10 Flops (floating-point operations), the relative error is less than $1.5 \cdot 2^{-106} + 4 \cdot 2^{-159}$ (see Table 2.2).

However, more than often these algorithms also come with a performance penalty. This is because when analyzing them we consider worst case scenarios, which for the average case is too pessimistic. If we loosen the accuracy constraints, for the same operation, one could use another algorithm, that only takes 7 Flops, but for which the relative error is less than $3 \cdot 2^{-106}$. This is detailed in what follows.

2. The *multi_prec.h* file implements the "quick-and-dirty" level of the library. It uses algorithms which are faster, but do not consider accuracy issues for corner cases. In most cases the result is going to be the same as obtained when computing with the certified level. The uncertainty appears if cancellation happens during intermediate computations, since this can generate intermediate 0s or even non-monotonic expansions in the result. For double-double operations we also implement specialized templates using the least accurate algorithms from Chapter 2.

This level also comes with a code generation module, that allows for the user to code generate the algorithms function of the needed expansion size. This module provides increased performance by custom unrolling some complex loops (which are usually not optimized by `gcc` or `nvcc` compilers).

We recommend the use of this level if the performance requirements are strong, especially if there is the possibility of a-posteriori verification of the correctness of the numerical result.

More implementation details are given in Appendix A, alongside with the code of the class *multi_prec* in Appendix B.1.

Parallel expansions. In Chapter 4 we will present parallel arithmetic algorithms tuned for GPU implementation. These algorithms are dealt with separately. In a separate file, *gpu_mprec.h*, we declare the simple (not templated) class:

```
class gpu_mprec{
  private:
    double val;
  public:
    ...
}
```

which appears from the point of the view of a single execution thread, this is why it stores only one value.

For an easy "switch" between the two classes we implemented load and store functions that can distribute a *multi_prec* object across threads and reform it, respectively. In more detail:

```
template <int prec>
__device__ gpu_mprec loadExpans(multi_prec<prec> const &mp){
  if (threadIdx.x < prec)
    return gpu_mprec(mp.getData()[threadIdx.x]);
  else
    return gpu_mprec(0.);
}

template <int prec>
__device__ void storeExpans(gpu_mprec gmp, multi_prec<prec> &mp){
  mp.setElement(gmp.getVal(), threadIdx.x);
}
```

Alongside two constructors, a getter and a setter, the class implements the three parallel algorithms presented in Chapter 4. These are parallelized using the x dimension of the thread block, allowing the user to also parallelize at a higher level, depending on the problem, using dimensions y and z. At a first glance this may seam difficult, but users that are familiar with cuda programing can take advantage of these algorithms with minimum effort.

The entire code of this class can be consulted in Appendix B.2.

1.4.1 Our server configuration

During the development of CAMPARY we tested its accuracy and performance on different models of CPU and GPU. In what follows we detail the last server configuration that we used, on which we obtained the performance reported throughout this manuscript.

As the CPU we used an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz based on the Haswell architecture, which offers improved AVX[8] 2.0 instruction with floating-point FMA3 with up to twice the Flops per core (16 Flops/clock). A detailed overview is given in Appendix C: the main features in Table C.1 and an excerpt of the result obtained using the command *more /proc/cpuinfo* in Figure C.1.

On the GPU side we had two Nvidia Tesla K20Xm cards with Kepler GK110 architecture. Details are also included in Appendix C: important features in Tables C.2 and C.3 and an excerpt of the result obtained using the command *nvidia-smi -i 0 -q* in Figure C.2.

The software configuration was as follows:

- Debian 4.9.2-10 GNU/Linux 8.2 operating system with 3.16.0-4-amd64 kernel;
- compilers GCC and G++ 4.9.2;
- CUDA 7.5 toolkit with NVCC V7.5.17.

8. Advanced Vector Extensions.

Chapter 2

Double-Word Arithmetic

This chapter is dedicated to the analysis of several classical basic building blocks of double-word arithmetic (frequently called "double-double arithmetic" in the literature). We consider addition/subtraction, multiplication and division. Some of the algorithms are known, but some of them are new. For most of the algorithms we get better relative error bounds than the ones previously published, except for the addition of two double-word numbers, for which we show that the previously published error bound was wrong. We also give numerical examples that illustrate the tightness of our bounds.

This is a joint work with J.-M. Muller and M. Joldes, presented in *Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic* [43], which, as we are writing these lines, is under revision at ACM Transactions on Mathematical Software (ACM TOMS).

Given the fact that our goal is to offer extended precision for applications that could benefit from it, the first logical step towards achieving that is to extend it just "a little", say "double" it.

Double-word arithmetic, called "double-double" in most of the literature, consists in representing a real number as the unevaluated sum of two floating-point numbers. In all existing implementations, the underlying floating-point format is the binary64 defined by the IEEE 754-2008 Standard [38] (see Section 1.1), hence the name "double-double". Kahan qualifies double-double arithmetic as an "attractive nuisance except for the BLAS" and even compares it to an unfenced backyard swimming pool!

Double-word arithmetic is not similar to a conventional floating-point arithmetic with twice the precision. It lacks many nice properties such as Sterbenz Lemma (Lemma 1.1.2), clearly defined rounding modes, etc. The difference becomes more clear when comparing the double-double format (made up with binary64 floating-point numbers) and the binary128 format. Though one might have the impression that the two are equivalent, there are a few important differences (presented in Table 2.1) like the precision or the exponent range. Kahan even mentions in [48] that double-double "undermines the incentive to provide quadruple-precision correctly rounded".

This may well be true: although the binary128 format (frequently called "quad-precision") was specified by the IEEE 754-2008 Standard on Floating-Point Arithmetic, it is seldom implemented in hardware. To our knowledge, the only commercially significant platform that has supported binary128 in hardware for the last decade has been the IBM Z Systems [59]. Thus, one will be tempted to use "double-double" arithmetic at times. Furthermore, even if hardwired binary128 arithmetic becomes commonplace, there will be a need for "double-quad" operations for carefully implementing very accurate binary128 elementary functions. Hence, designing and analyzing algorithms for double-word arithmetic is of interest.

Table 2.1 – Main differences between the double-double format (made up with binary64 floating-point numbers) and quad-precision (binary128).

	double-double	quad-precision
Precision	≥ 107 bits "wobbling"	113 bits
Exponent range	-1022 to 1023 (11 bits)	-16382 to 16383 (15 bits)
Rounding modes	N/A	RN, RU, RD, RZ

Some of the first work in this area was done by Dekker [22], who presented algorithms for adding, multiplying, and dividing double-word numbers. His addition and multiplication algorithms are very similar (in fact, mathematically equivalent) to Algorithms 8 and 13, analyzed below, but his division algorithm was quite different (and less accurate) than the algorithms considered here. After that, Linnainmaa [60] suggested similar algorithms, assuming that an underlying extended precision format is available, but we will not consider that hypothesis here.

Libraries that offer double-double arithmetic have been written by Briggs [13], who no longer maintains his library, and by Hida, Li, and Bailey [34, 35]. The later one, the QD library, is the most often used nowadays, and it offers both a double-double format and a quad-double one (treated in Chapter 3).

Though extensive work has been done in this area, many algorithms have been published without a proof, or with error bounds that are sometimes loose, sometimes fuzzy (the error is "less than a small integer times u^2"), and sometimes unsure. Thus we felt the strong need to "clean up" the literature. In this chapter we provide a rigorous error analysis of some existing algorithms for double-word arithmetic, and introduce a few new ones. We cannot suppress all the drawbacks mentioned by Kahan: clearly, having in hardware a "real" floating-point arithmetic with twice the precision would be a better option. And yet, if rigorously proven and reasonably tight error bounds are provided, expert programmers can rely on double-word arithmetic for extending the precision of calculations in places where the available floating-point arithmetic does not suffice.

In Definition 2.0.1 we formally introduce the concept of double-word representation.

Definition 2.0.1. *A double-word number x is the unevaluated sum $x_h + x_\ell$ of two floating-point numbers x_h and x_ℓ such that*

$$x_h = \mathrm{RN}(x).$$

In our work we tried to obtain error bounds as tight as possible. To this end, roughly speaking, we limit the exponent range of the numbers and we subdivide the possible range of values into sections that allow us to deduce different properties between the numbers. Thus, when performing operations using two double-word numbers $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$, we can assume without loss of generality that:

- x_h is positive, otherwise we change the sign of all the operands;
- $1 \leq x_h < 2$ (which implies $1 \leq x_h < 2 - 2u$, since x_h is a floating-point number), otherwise we scale the operands by a power of 2.

The same principles can be applied if we have operations between a double-word number $x = (x_h, x_\ell)$ and a floating-point one y.

Throughout this entire chapter we will consider only double-word numbers that satisfy Definition 2.0.1 and use an underlying floating-point format with precision-p. This offers generality to the analysis and allows users to also extend the binary32 format or even the binary128 if one day

it becomes widely available. We also consider that underflow and overflow do not occur, even though in the case of addition underflow does not pose a problem. All algorithms return their results as a double-word number, insured by the Fast2Sum applied at the end of each algorithm, which has a "renormalization" purpose. Also, we stress here that all the constraints given in each theorem on the precision-p, in order for the error bound to hold, are satisfied in practice.

The sequel of the chapter is organized as follows: Section 2.1.1 deals with the sum of a double-word number and a floating-point number; Section 2.1.2 is devoted to the sum of two double-word numbers; in Section 2.2.1 we consider the product of a double-word number by a floating-point number; in Section 2.2.2 we consider the product of two double-word numbers; Section 2.3.1 deals with the division of a double-word number by a floating-point number, and Section 2.3.2 is devoted to the division of two double-word numbers.

2.1 Addition of double-word numbers

The basic idea behind the addition algorithms is to accumulate the numbers that are roughly of the same magnitude. For example, when adding two double-word numbers $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$, we consider that x_h and y_h have around the same magnitude (this is true in most cases), and we know for sure that, unless they cancel (but in this case Sterbenz lemma will help), they are the most significant terms in the addition (since all double-word numbers satisfy Definition 2.0.1). As a deduction, the error obtained from adding them should be, roughly speaking, around the same magnitude as x_ℓ and y_ℓ. On account of that, we try to minimize the number of basic floating-point operations that we do, while still accumulating the errors.

We present first the case of adding a double-word number with a floating-point one, followed by the addition of two double-word numbers.

2.1.1 Addition of a double-word number and a floating-point number

The algorithm implemented in the QD library [35] for adding a double-word number and a floating-point number is Algorithm 7 below that computes $(x_h, x_\ell) + y$. Figure 2.1 contains a graphical representation of the algorithm, using the same notations.

This algorithm, or variants of it, implicitly appears in many "compensated summation" algorithms that aim at accurately computing the sum of several floating-point numbers. At intermediate steps of the summation, most such algorithms represent the sum of all input numbers accumulated so far as a doubldotse-word number. For instance the first two lines of the algorithm constitute the internal loop of Rump, Ogita and Oishi's "cascaded summation" algorithm [79].

In what follows we analyze Algorithm 7 and we show its correctness by proving Theorem 2.1.1.

Theorem 2.1.1. *The relative error of Algorithm 7 (DWPlusFP) is bounded by*

$$\frac{2u^2}{1-2u} = 2u^2 + 4u^3 + 8u^4 + \cdots,$$

which is less than $2u^2 + 5u^3$ *as soon as* $p \geq 4$.

Algorithm 7 – DWPlusFP(x_h, x_ℓ, y).

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

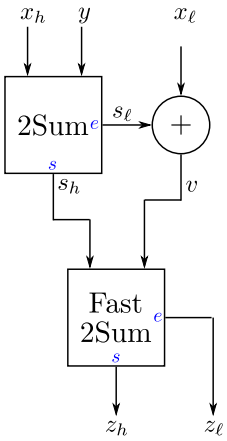


Figure 2.1 – Graphical representation of Algorithm 7. In the 2Sum and Fast2Sum calls the sum s is outputted downwards and the error e to the right.

Proof. First of all, we can quickly proceed with the case $x_h + y = 0$: in that case $s_h = s_\ell = 0$ and the computation is errorless. 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.[1] We also assume that x_h is positive (otherwise we change the sign of all the operands), and that $1 \leq x_h \leq 2 - 2u$ (otherwise we scale the operands by a power of 2).

1. If $-x_h \leq y \leq -\frac{x_h}{2}$, then Sterbenz Lemma (Lemma 1.1.2) implies $s_h = x_h + y$ and $s_\ell = 0$. It follows that $v = x_\ell$. Lemma 1.1.7 implies $|s_h| \geq \frac{1}{2}\,\mathrm{ulp}(x_h)$, which implies $|s_h| \geq |x_\ell|$. Hence Algorithm Fast2Sum introduces no error at line 3 of the algorithm, and so $z_h + z_\ell = s_h + v = x + y$ exactly.

2. If $-\frac{x_h}{2} < y \leq x_h$, then $\frac{1}{2} \leq \frac{x_h}{2} < x_h + y \leq 2x_h$, such that $s_h \geq \frac{1}{2}$. Since $|x_\ell + s_\ell| \leq 3u$ (see the two cases considered below), we have $|v| \leq 3u$. It follows that $s_h > |v|$, so Algorithm Fast2Sum introduces no error at line 3 of the algorithm.

- If $x_h + y \leq 2$ then $|s_\ell| \leq u$; so that $|x_\ell + s_\ell| \leq 2u$, hence,

$$v = x_\ell + s_\ell + \epsilon,$$

with $|\epsilon| \leq u^2$. In the end we get $z_h + z_\ell = s_h + v = x + y + \epsilon$ and the relative error $|\frac{\epsilon}{x+y}|$ of the calculation is bounded by

$$\frac{|\epsilon|}{\frac{1}{2} - u} \leq \frac{2u^2}{1 - 2u}.$$

- If $x_h + y > 2$ then $|s_\ell| \leq 2u$, such that $|x_\ell + s_\ell| \leq 3u$, hence,

$$v = x_\ell + s_\ell + \epsilon,$$

1. (y, x_ℓ) may not be a double-word number, according to Definition 2.0.1, in the case $x_\ell = \frac{1}{2}\,\mathrm{ulp}(y) = \frac{1}{2}\,\mathrm{ulp}(x_h)$. However, one easily checks that in that case the algorithm returns an exact result.

with $|\epsilon| \leq 2u^2$. Therefore the relative error $|\frac{\epsilon}{x+y}|$ of the calculation is bounded by

$$\frac{|\epsilon|}{2-u} \leq \frac{2u^2}{2-u}.$$

□

Notice that the bound given by Theorem 2.1.1 is very sharp. In fact, it is *asymptotically optimal*, showed by the generic example given in Example 2.1.2.

Example 2.1.2. *Let the following values as input for Algorithm 7:*

$$\begin{aligned} x_h &= 1, \\ x_\ell &= (2^p - 1)2^{-2p}, \text{ and} \\ y &= -\tfrac{1}{2}(1 - 2^{-p}). \end{aligned}$$

The algorithm computes the sum as

$$z_h + z_\ell = \frac{1}{2} + 3 \cdot 2^{-p-1},$$

while the exact sum is

$$x_h + x_\ell + y = \frac{1}{2} + 3 \cdot 2^{-p-1} - 2^{-2p},$$

resulting in a relative error equal to

$$\frac{2u^2}{1 + 3u - 2u^2} \approx 2u^2 - 6u^3.$$

For an implementation that uses the binary64 format, the input from the above example gives an error equal to $1.99999999999999933\ldots \times 2^{-106}$.

2.1.2 Addition of two double-word numbers

Algorithm 8 (Figure 2.2) below was first given by Dekker [22], under the name of *add2*, but in a slightly different presentation: he did not use Algorithm 2Sum in line 1 of the algorithm, instead there was a comparison of $|x_h|$ and $|y_h|$ followed by a possible swap of x and y and a call to Fast2Sum. However, from a mathematical point of view, Dekker's algorithm and Algorithm 8 are equivalent: they always return the same result. This algorithm was then implemented in the QD library [35] under the name "sloppy addition".

Dekker proved an error bound on the order of $(|x| + |y|) \cdot 4u^2$. Because of the absolute values, when x and y do not have the same sign, there is no proof that the relative error is bounded. Indeed, the relative error can be so large that the obtained result has no significance at all. A generic example of such values is given in Example 2.1.3 below.

Example 2.1.3. *Let the following values as input for Algorithm 8:*

$$\begin{aligned} x_h &= 1 + 2^{-p+3}, \\ x_\ell &= -2^{-p}, \\ y_h &= -1 - 6 \cdot 2^{-p}, \text{ and} \\ y_\ell &= -2^{-p} + 2^{-2p}. \end{aligned}$$

The algorithm computes the sum as $z_h + z_\ell = 0$, while the exact sum is $x + y = 2^{-2p}$, resulting in a relative error equal to 1.

Algorithm 8 – SloppyDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$.

1: $(s_h, s_\ell) \leftarrow \mathsf{2Sum}(x_h, y_h)$
2: $v \leftarrow \mathrm{RN}(x_\ell + y_\ell)$
3: $w \leftarrow \mathrm{RN}(s_\ell + v)$
4: $(z_h, z_\ell) \leftarrow \mathsf{Fast2Sum}(s_h, w)$
5: **return** (z_h, z_ℓ)

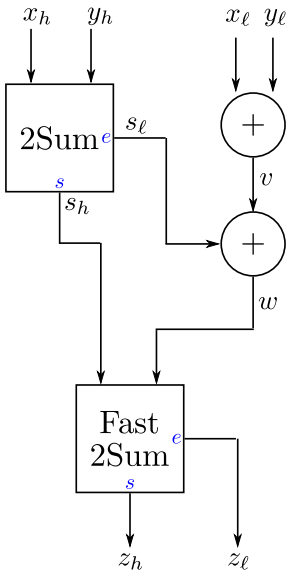


Figure 2.2 – Graphical representation of Algorithm 8. In the 2Sum and Fast2Sum calls the sum s is outputted downwards and the error e to the right.

Algorithm 9 – AccurateDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$.

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: $c \leftarrow \mathrm{RN}(s_\ell + t_h)$
4: $(v_h, v_\ell) \leftarrow \mathsf{Fast2Sum}(s_h, c)$
5: $w \leftarrow \mathrm{RN}(t_\ell + v_\ell)$
6: $(z_h, z_\ell) \leftarrow \mathsf{Fast2Sum}(v_h, w)$
7: **return** (z_h, z_ℓ)

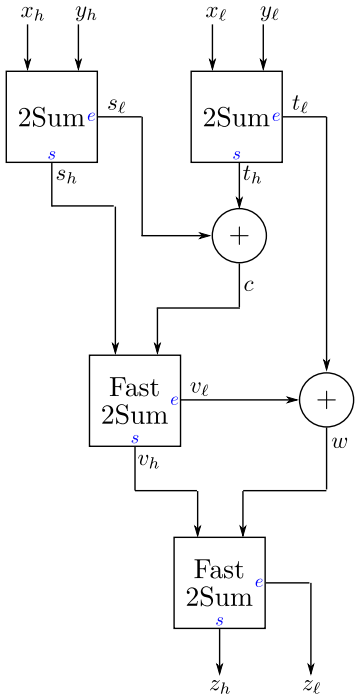


Figure 2.3 – Graphical representation of Algorithm 9. In the 2Sum and Fast2Sum calls the sum s is outputted downwards and the error e to the right.

As showed above, the result of Algorithm 8 are not to be trusted, this is why, the use of this algorithm should be restricted to special cases such as, for instance, when we know that the operands will have the same sign. When accurate computations are required, it is much more advisable to use Algorithm 9 (Figure 2.3), presented by Li et al. [57, 58] and implemented in the QD library under the name of "IEEE addition".

In [57, 58], the authors claim that in binary64 arithmetic the relative error of Algorithm 9 is upper bounded by $2 \cdot 2^{-106}$. This statement is not correct, as shown by Example 2.1.4.

Example 2.1.4. *If we have as input for Algorithm 9, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 9007199254740991, \\ x_\ell &= -9007199254740991/2^{54}, \\ y_h &= -9007199254740987/2, \text{ and} \\ y_\ell &= -9007199254740991/2^{56}, \end{aligned}$$

then the relative error of the algorithm is

$$2.24999999999999956\ldots \times 2^{-106}.$$

In Example 2.1.5 we give a generic example, to show that the bound can be exceeded for any precision-p floating-point system.

Example 2.1.5. *Let the following values as input for Algorithm 9:*

$$\begin{aligned} x_h &= 2^p - 1, \\ x_\ell &= -(2^p - 1)2^{-p-1}, \\ y_h &= -(2^p - 5)/2, \text{ and} \\ y_\ell &= -(2^p - 1)2^{-p-3}. \end{aligned}$$

It leads to a relative error that is asymptotically equivalent (as p goes to infinity) to $2.25u^2$.

A new bound on the relative error of Algorithm 9 is given in Theorem 2.1.6, for which we provide a rigorous proof.

Theorem 2.1.6. *If $p \geq 3$, the relative error of Algorithm 9 (AccurateDWPlusDW) is bounded by*

$$\frac{3u^2}{1-4u} = 3u^2 + 12u^3 + 48u^4 + \cdots,$$

which is less than $3u^2 + 13u^3$ as soon as $p \geq 6$.

Proof. First of all, we exclude the straightforward case in which one of the operands is zero. We can also quickly proceed with the case $x_h + y_h = 0$: in that case one easily sees that the returned result is $\mathrm{2Sum}(x_\ell, y_\ell)$, which is equal to $x + y$, i.e., the computation is errorless. Now, without loss of generality, we assume $1 \leq x_h \leq 2 - 2u$, $x \geq |y|$ (which implies $x_h \geq |y_h|$), and $x_h + y_h$ nonzero. Notice that $1 \leq x_h < 2$ implies $1 \leq x_h \leq 2 - 2u$, since x_h is a floating-point number.

1. If $-2 + 2u \leq y_h \leq -1$. Notice that $|x_\ell|$ and $|y_\ell|$ are bounded by u, and that x and $|y|$ are bounded by $2 - u$. We know that $-1 + 2u \leq x_h + y_h \leq 1 - 2u$ and the sum $x_h + y_h$ is a multiple of $2^{-p+1} = 2u$, hence it is a floating-point number. This implies $s_h = x_h + y_h$ exactly and $s_\ell = 0$, therefore $c = \mathrm{RN}(t_h) = t_h$. We also have

$$t_h + t_\ell = x_\ell + y_\ell,$$

and, since $|x_\ell + y_\ell| \leq 2u$, $|t_h| \leq 2u$ and $|t_\ell| \leq u^2$. Because s_h is a nonzero multiple of $2u$ and $|c| = |t_h| \leq 2u$, we can legitimately use the Fast2Sum Algorithm in line 4 of the algorithm (it does not introduce any error). Therefore,

$$v_h + v_\ell = s_h + t_h = x + y - t_\ell.$$

Also, $|s_h + t_h| \leq (1 - 2u) + 2u \leq 1$, so that $|v_h| \leq 1$ and $|v_\ell| \leq \frac{u}{2}$. We finally have

$$w = t_\ell + v_\ell + \epsilon_2,$$

with

$$|\epsilon_2| \leq \frac{1}{2}\operatorname{ulp}(t_\ell + v_\ell) \leq \frac{1}{2}\operatorname{ulp}(u^2 + \frac{u}{2}), \tag{2.1}$$

and

$$|\epsilon_2| \leq \frac{1}{2}\operatorname{ulp}\left[\frac{1}{2}\operatorname{ulp}(x_\ell + y_\ell) + \frac{1}{2}\operatorname{ulp}\left((x+y) + \frac{1}{2}\operatorname{ulp}(x_\ell + y_\ell)\right)\right]. \tag{2.2}$$

From (2.1), we get $|\epsilon_2| \leq \frac{u^2}{2}$.

Now, since s_h is a nonzero multiple of $2u$, $|s_h| \geq 2u$. Lemma 1.1.7 implies that either $s_h + t_h = 0$, or $|v_h| = |\operatorname{RN}(s_h + c)| = |\operatorname{RN}(s_h + t_h)| \geq 2u^2$. If $s_h + t_h = 0$ then $v_h = v_l = 0$ and the sequel of the proof is straightforward. Therefore, in the following, we assume $|v_h| \geq 2u^2$. This implies $|v_h| \geq \frac{u^2}{1-u}$, so that $|v_h| \geq u|v_h| + u^2 \geq |v_\ell| + |t_\ell|$. Hence $|w| = |\operatorname{RN}(v_\ell + t_\ell)| \leq |v_h|$, so the Fast2Sum Algorithm in line 6 introduces no error. We finally have,

$$z_h + z_\ell = v_h + w = x + y + \epsilon_2. \tag{2.3}$$

Directly using (2.3) and the bound $\frac{u^2}{2}$ on $|\epsilon_2|$ to get a relative error bound might result in a rather large bound, because $x + y$ may be small. However, as we are going to see, when $x + y$ is very small, some simplification occurs thanks to Sterbenz Lemma. First, $x_h + y_h$ is a nonzero multiple of $2u$. Hence, since $|x_\ell + y_\ell| \leq 2u$, we have $|x_\ell + y_\ell| \leq x_h + y_h$.

- If $-(x_h + y_h) \leq x_\ell + y_\ell \leq -\frac{1}{2}(x_h + y_h)$, which implies $-s_h \leq t_h \leq -\frac{1}{2}s_h$, then Sterbenz Lemma applies to the floating-point addition of s_h and $c = t_h$. Therefore line 4 of the algorithm produces $v_h = s_h$ and $v_\ell = 0$. An immediate consequence is $\epsilon_2 = 0$, so that $z_h + z_\ell = v_h + w = x + y$: the computation of $x + y$ is errorless.

- If $-\frac{1}{2}(x_h + y_h) \leq x_\ell + y_\ell \leq x_h + y_h$, then $2(x_\ell + y_\ell) \leq (x_h + y_h) + (x_\ell + y_\ell)$, which implies $x_\ell + y_\ell \leq \frac{1}{2}(x + y)$. Also, $-\frac{1}{2}(x + y) \leq \frac{1}{2}(x_\ell + y_\ell)$, which implies $-(x + y) \leq x_\ell + y_\ell$. Hence $|x_\ell + y_\ell| \leq x + y$, so that

  $$\operatorname{ulp}(x_\ell + y_\ell) \leq \operatorname{ulp}(x + y).$$

  Combined with (2.2), this gives

  $$|\epsilon_2| \leq \frac{1}{2}\operatorname{ulp}\left[\frac{1}{2}\operatorname{ulp}(x+y) + \frac{1}{2}\operatorname{ulp}\left((x+y) + \frac{1}{2}\operatorname{ulp}(x+y)\right)\right].$$

  Hence,

  $$|\epsilon_2| \leq \frac{1}{2}\operatorname{ulp}\left(\frac{3}{2}\operatorname{ulp}(x+y)\right) \leq 2^{-p}\operatorname{ulp}(x+y) \leq 2 \cdot 2^{-2p} \cdot (x+y).$$

2. If $-1 + u \leq y_h \leq -\frac{x_h}{2}$. It implies $u \leq x_h + y_h \leq \frac{x_h}{2}$, so Sterbenz Lemma can be applied to the first line of the algorithm. Therefore $s_h = x_h + y_h$ and $s_\ell = 0$, so that $c = \mathrm{RN}(t_h) = t_h$. Hence, this case is very similar to the previous one. Since s_h is a nonzero multiple of u and $|c| = |t_h| \leq \frac{3u}{2}$, the floating-point exponent of s_h is at least 2^{-p} and the floating-point exponent of c is at most 2^{-p}. Hence Algorithm Fast2Sum at line 4 of the algorithm introduces no error, and $v_h + v_l = s_h + c$.

Now, since s_h is a nonzero multiple of u, $|s_h| \geq u$. Lemma 1.1.7 implies that either $s_h + t_h = 0$, or $|v_h| = |\mathrm{RN}(s_h + c)| = |\mathrm{RN}(s_h + t_h)| \geq u^2$.

- If $s_h + t_h = 0$ then $v_h = v_l = 0$ and the sequel of the proof is straightforward.
- If $|v_h| = u^2$, then $|v_\ell + t_\ell| \leq u|v_h| + u^2 = u^3 + u^2$, therefore $|w| = |\mathrm{RN}(v_\ell + t_\ell)| \leq u^2 = |v_h|$.
- If $|v_h| > u^2$, then (since v_h is a floating-point number) $|v_h| \geq u^2 + 2u^3$ hence $|v_h| \geq \frac{u^2}{1-u}$, so that $|v_h| \geq u|v_h| + u^2 \geq |v_\ell| + |t_\ell|$. Hence $|w| = |\mathrm{RN}(v_\ell + t_\ell)| \leq |v_h|$.

Therefore, in all cases, Algorithm Fast2Sum introduces no error in line 6 of the algorithm. Again, we have

$$z_h + z_\ell = v_h + w = x + y + \epsilon_2,$$

with

$$|\epsilon_2| \leq \frac{1}{2}\,\mathrm{ulp}\left[\frac{1}{2}\,\mathrm{ulp}(x_\ell + y_\ell) + \frac{1}{2}\,\mathrm{ulp}\left((x+y) + \frac{1}{2}\,\mathrm{ulp}(x_\ell + y_\ell)\right)\right].$$

From $x_h + y_h \geq u$ and $|x_\ell + y_\ell| \leq 2u$, we deduce $-2(x_h + y_h) \leq x_\ell + y_\ell \leq 2(x_h + y_h)$.

- If $-2(x_h + y_h) \leq x_\ell + y_\ell \leq -\frac{1}{2}(x_h + y_h)$ then $-2s_h \leq t_h = c \leq -\frac{1}{2}s_h$, hence Sterbenz Lemma can be applied to line 4 of the algorithm, so that $v_\ell = 0$, hence $w = \mathrm{RN}(t_\ell) = t_\ell$ and $\epsilon_2 = 0$ so that the computation is errorless: $z_h + z_\ell = x + y$;
- If $-\frac{1}{2}(x_h + y_h) < x_\ell + y_\ell \leq 2(x_h + y_h)$, then $3(x_\ell + y_\ell) \leq 2(x_h + y_h + x_\ell + y_\ell)$ so that $x_\ell + y_\ell \leq \frac{2}{3}(x+y)$, and $-\frac{1}{2}(x+y) \leq \frac{1}{2}(x_\ell + y_\ell)$, so that $-(x+y) < x_\ell + y_\ell$. Hence, in any case, $|x_\ell + y_\ell| < x + y$, so that $\mathrm{ulp}\,(x_\ell + y_\ell) \leq \mathrm{ulp}(x + y)$, and we end up with the same bound as in the previous case:
$$|\epsilon_2| \leq 2 \cdot 2^{-2p}|x + y|.$$

3. If $-\frac{x_h}{2} < y_h \leq x_h$. It implies $\frac{x_h}{2} < x_h + y_h$, and consequently $\frac{1}{2} < x_h + y_h$.

- If $\frac{1}{2} < x_h + y_h \leq 1 - 2u$ then $\frac{1}{2} \leq s_h \leq 1 - 2u$ and $|s_\ell| \leq \frac{u}{2}$. Notice that having $x_h + y_h \leq 1 - 2u$ requires y_h to be negative, so that $-1 < y \leq 0$, which implies $|y_\ell| \leq \frac{u}{2}$. We have
$$t_h + t_\ell = x_\ell + y_\ell,$$
with $|x_\ell + y_\ell| \leq \frac{3u}{2}$, hence $|t_h| \leq \frac{3u}{2}$, and $|t_\ell| \leq u^2$. Now,
$$c = s_\ell + t_h + \epsilon_1,$$

with $|s_\ell + t_h| \leq 2u$, so that $|c| \leq 2u$, and $|\epsilon_1| \leq u^2$. Since $s_h \geq \frac{1}{2}$ and $|c| \leq 2u$, we can legitimately use Algorithm Fast2Sum in line 4 of the algorithm because it introduces no error, meaning that

$$v_h + v_\ell = s_h + c \leq 1 - 2u + 2u = 1.$$

Therefore $v_h \leq 1$ and $|v_\ell| \leq \frac{u}{2}$. Thus,

$$w = t_\ell + v_\ell + \epsilon_2,$$

where $|t_\ell + v_\ell| \leq \frac{u}{2} + u^2$, so that $|\epsilon_2| \leq \frac{u^2}{2}$. Also, $s_h \geq \frac{1}{2}$ and $|c| \leq 2u$ imply $v_h \geq \frac{1}{2} - 2u$. And $|t_\ell + v_\ell| \leq \frac{u}{2} + u^2$ implies $|w| \leq \frac{u}{2} + u^2$. Hence, if $p \geq 3$ (i.e., $u \leq \frac{1}{8}$) Algorithm Fast2Sum introduces no error at line 6 of the algorithm, i.e., $z_h + z_\ell = v_h + w$. Therefore,

$$z_h + z_\ell = x + y + \eta,$$

with $|\eta| = |\epsilon_1 + \epsilon_2| \leq \frac{3u^2}{2}$.
Since $x + y \geq (x_h - u) + (y_h - \frac{u}{2}) > \frac{1}{2} - \frac{3u}{2}$, the relative error $\frac{|\eta|}{x+y}$ is upper-bounded by

$$\frac{\frac{3u^2}{2}}{\frac{1}{2} - \frac{3u}{2}} = \frac{3u^2}{1 - 3u}.$$

- If $1 - 2u < x_h + y_h \leq 2 - 4u$ then $1 - 2u \leq s_h \leq 2 - 4u$ and $|s_\ell| \leq u$. We have,

$$t_h + t_\ell = x_\ell + y_\ell,$$

with $|x_\ell + y_\ell| \leq 2u$, hence $|t_h| \leq 2u$, and $|t_\ell| \leq u^2$. Now,

$$c = s_\ell + t_h + \epsilon_1,$$

with $|s_\ell + t_h| \leq 3u$, so that $|c| \leq 3u$, and $|\epsilon_1| \leq 2u^2$. Since $s_h \geq 1 - 2u$ and $|c| \leq 3u$, if $p \geq 3$ then Algorithm Fast2Sum introduces no error in line 4 of the algorithm, i.e., $v_h + v_\ell = s_h + c$. Therefore,

$$v_h + v_\ell \leq 2 - 4u + 3u = 2 - u,$$

so that $v_h \leq 2$ and $|v_\ell| \leq u$. Thus,

$$w = t_\ell + v_\ell + \epsilon_2,$$

where $|t_\ell + v_\ell| \leq u + u^2$, so that $|\epsilon_2| \leq u^2$. Also, $s_h \geq 1 - 2u$ and $|c| \leq 3u$ imply $v_h \geq 1 - 5u$, and $|t_\ell + v_\ell| \leq u + u^2$ implies $|w| \leq u$. Hence, if $p \geq 3$, Algorithm Fast2Sum introduces no error in line 6 of the algorithm, i.e., $z_h + z_\ell = v_h + w$. Therefore,

$$z_h + z_\ell = x + y + \eta,$$

with $|\eta| = |\epsilon_1 + \epsilon_2| \leq 3u^2$.
Since $x + y \geq (x_h - u) + (y_h - u) > 1 - 4u$, the relative error $\frac{|\eta|}{x+y}$ is upper-bounded by

$$\frac{3u^2}{1 - 4u}.$$

- If $2-4u < x_h + y_h \leq 2x_h$ then $2-4u \leq s_h \leq \mathrm{RN}(2x_h) = 2x_h \leq 4-4u$ and $|s_\ell| \leq 2u$. We have,
$$t_h + t_\ell = x_\ell + y_\ell,$$
with $|x_\ell + y_\ell| \leq 2u$, hence $|t_h| \leq 2u$, and $|t_\ell| \leq u^2$. Now,
$$c = s_\ell + t_h + \epsilon_1,$$
with $|s_\ell + t_h| \leq 4u$, so that $|c| \leq 4u$, and $|\epsilon_1| \leq 2u^2$. Since $s_h \geq 2-4u$ and $|c| \leq 4u$, if $p \geq 3$, then Algorithm Fast2Sum introduces no error in line 4 of the algorithm. Therefore,
$$v_h + v_\ell = s_h + c \leq 4 - 4u + 4u = 4,$$
so that $v_h \leq 4$ and $|v_\ell| \leq 2u$. Thus,
$$w = t_\ell + v_\ell + \epsilon_2,$$
where $|t_\ell + v_\ell| \leq 2u + u^2$. Hence, either $|t_\ell + v_\ell| < 2u$ and $|\epsilon_2| \leq \frac{1}{2}\,\mathrm{ulp}(t_\ell + v_\ell) \leq u^2$, or $2u \leq t_\ell + v_\ell \leq 2u + u^2$, in which case $w = \mathrm{RN}(t_\ell + v_\ell) = 2u$ and $|\epsilon_2| \leq u^2$. In all cases $|\epsilon_2| \leq u^2$. Also, $s_h \geq 2-4u$ and $|c| \leq 4u$ imply $v_h \geq 2-8u$. And $|t_\ell + v_\ell| \leq 2u + u^2$ implies $|w| \leq 2u$. Hence, if $p \geq 3$ then Algorithm Fast2Sum introduces no error in line 6 of the algorithm. All this gives
$$z_h + z_\ell = v_h + w = x + y + \eta,$$
with $|\eta| = |\epsilon_1 + \epsilon_2| \leq 3u^2$.
Since $x + y \geq (x_h - u) + (y_h - u) > 2 - 6u$, the relative error $\frac{|\eta|}{x+y}$ is upper-bounded by
$$\frac{3u^2}{2-6u},$$
The largest bound obtained in the various cases we have analyzed is
$$\frac{3u^2}{1-4u}.$$

Elementary calculus shows that for $u \in [0, \frac{1}{64}]$ (i.e., $p \geq 6$) this is always less than $3u^2 + 13u^3$. □

The bound given in Theorem 2.1.6 is probably not optimal. The largest relative error we have obtain through many random tests is around $2.25u^2$, as showed by Examples 2.1.4 and 2.1.5.

2.2 Multiplication of double-word numbers

For performing a double-word multiplication the algorithms follow the paper-and-pencil technique by accumulating partial products. A big advantage when it comes to multiplication, as opposed to addition, is that we know for sure cancellation cannot happen at the level of the most significant term of the result. For example, when multiplying two double-word numbers $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ we know that the product $z = (z_h, z_\ell)$ will be roughly around $2^{e_{x_h}+e_{y_h}}$ (more precisely, $2^{e_{x_h}+e_{y_h}} \leq |xy| < 2^{e_{x_h}+e_{y_h}+2}$ holds), where e_{x_h} and e_{y_h} are the exponents of x_h and y_h, respectively.

The above property allows us to explore different algorithms that offer compromises between speed and accuracy. For example, in some cases we can even chose not to accumulate at all the partial product $x_\ell y_\ell$ (we know that its magnitude is very small), and to simply account for it when computing the error bound.

2.2.1 Multiplication of a double-word number by a floating-point number

For multiplying a double-word number with a floating-point one we first consider Algorithm 10 (Figure 2.4) that was suggested by Li et al. [57].

Algorithm 10 – DWTimesFP1(x_h, x_ℓ, y).

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

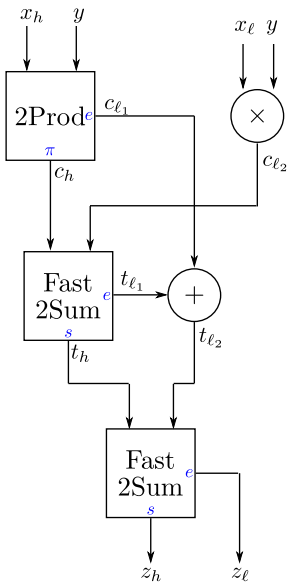


Figure 2.4 – Graphical representation of Algorithm 10. In the 2Prod and Fast2Sum calls the sum s or the product π are outputted downwards and the error e to the right.

In [57, 58] (with more detail in the technical report [57], which is a preliminary version of the journal paper [58]), Li et al. give a relative error bound $4 \cdot 2^{-106}$ for Algorithm 10 when the underlying floating-point arithmetic is binary64. Below, in Theorem 2.2.1, we present a new result, in the more general context of precision-p arithmetic, that significantly improves their relative error bound.

Theorem 2.2.1. *If $p \geq 4$, the relative error*

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

of Algorithm 10 (DWTimesFP1) is bounded by $\frac{3}{2}u^2 + 4u^3$.

Proof. One easily notices that if $x = 0$, or $y = 0$, or y is a power of 2, the obtained result is exact. Therefore, without loss of generality, we can assume $1 \leq x_h \leq 2 - 2u$ and $1 + 2u \leq y \leq 2 - 2u$. This gives $1 + 2u \leq x_h y \leq 4 - 8u + 4u^2$, so that

$$1 + 2u \leq c_h \leq 4 - 8u, \tag{2.4}$$

and

$$|c_{\ell 1}| \leq \frac{1}{2} \operatorname{ulp}(4 - 8u) = 2u. \tag{2.5}$$

From $|x_\ell| \leq u$ and $y \leq 2 - 2u$ we deduce

$$|c_{\ell 2}| \leq 2u - 2u^2, \tag{2.6}$$

so that $\epsilon_1 = x_\ell y - c_{\ell 2}$ satisfies $|\epsilon_1| \leq u^2$. From (2.4) and (2.6) we deduce that Algorithm Fast2Sum introduces no error at line 3 of the algorithm, i.e., $t_h + t_{\ell 1} = c_h + c_{\ell 2}$. Also, we deduce that

$$1 = \mathrm{RN}(1 + 2u^2) \leq t_h \leq \mathrm{RN}(4 - 6u - 2u^2) = 4 - 8u, \tag{2.7}$$

and

$$|t_{\ell 1}| \leq \frac{1}{2}\,\mathrm{ulp}(4 - 8u) = 2u. \tag{2.8}$$

From (2.5) and (2.8), we obtain

$$|t_{\ell 2}| \leq \mathrm{RN}(4u) = 4u, \tag{2.9}$$

and we find that $\epsilon_2 = t_{\ell 2} - (t_{\ell 1} + c_{\ell 1})$ satisfies

$$|\epsilon_2| \leq 2u^2. \tag{2.10}$$

Define $\epsilon = \epsilon_2 - \epsilon_1$. Using (2.7) and (2.9), we deduce that Algorithm Fast2Sum introduces no error at line 5 of the algorithm. Therefore,

$$\begin{aligned} z_h + z_\ell &= t_h + t_{\ell 2} \\ &= t_h + t_{\ell 1} + c_{\ell 1} + \epsilon_2 \\ &= c_h + c_{\ell 2} + c_{\ell 1} + \epsilon_2 \\ &= x_h y + x_\ell y - \epsilon_1 + \epsilon_2 \\ &= xy + \epsilon. \end{aligned} \tag{2.11}$$

Hence the absolute error of Algorithm 10 is $|\epsilon| \leq |\epsilon_1| + |\epsilon_2| \leq 3u^2$. Let us now consider two possible cases:

1. If $x_h y \geq 2$, then $xy \geq x_h(1-u)y \geq 2 - 2u$. This leads to a relative error $|\frac{\epsilon}{xy}|$ bounded by

$$\frac{3u^2}{2 - 2u} = \frac{3}{2}u^2 + \frac{3}{2}u^3 + \frac{3}{2}u^4 + \cdots. \tag{2.12}$$

2. If $x_h y < 2$, which implies $|c_h| \leq 2$, we easily improve on some of the previously obtained bounds. We have, $|c_{\ell 1}| \leq u$, and $t_h \leq \mathrm{RN}(2 + 2u - 2u^2) = 2$.

The case $t_h = 2$ is easily handled: (2.11) implies $xy = t_h + t_{\ell 2} - \epsilon \geq 2 - 4u - 3u^2$, and the relative error $|\frac{\epsilon}{xy}|$ is bounded by

$$\frac{3u^2}{2 - 4u - 3u^2} = \frac{3}{2}u^2 + 3u^3 + \frac{33}{4}u^4 + \cdots. \tag{2.13}$$

If $t_h < 2$ then $|t_{\ell 1}| \leq u$, $|t_{\ell 2}| \leq 2u$, and $|\epsilon_2| \leq u^2$. Hence, a first upper bound on $|\epsilon|$ is $2u^2$. However, some refinement is possible.

- first, if $|c_{\ell 2}| < u$ then $|\epsilon_1| \leq \frac{u^2}{2}$, which implies $|\epsilon| \leq \frac{3u^2}{2}$;
- second, if $|c_{\ell 2}| \geq u$, then $c_{\ell 2}$ is a multiple of $\mathrm{ulp}(u) = 2u^2$, so that $t_{\ell 1}$ is a multiple of $2u^2$. Also, since x_h and y are multiple of $2u$, $x_h y$ is a multiple of $4u^2$, so that $c_{\ell 1}$ is a multiple of $4u^2$. Hence, $t_{\ell 1} + c_{\ell 1}$ is a multiple of $2u^2$ of absolute value less than or equal to $2u$. This implies that $t_{\ell 1} + c_{\ell 1}$ is a floating-point number, hence $\mathrm{RN}(t_{\ell 1} + c_{\ell 1}) = t_{\ell 1} + c_{\ell 1}$ and $\epsilon_2 = 0$.

Therefore, when $t_h < 2$, $|\epsilon|$ is upper-bounded by $\frac{3u^2}{2}$, so the relative error $|\frac{\epsilon}{xy}|$ is bounded by

$$\frac{\frac{3}{2}u^2}{(1-u)(1+2u)} \leq \frac{3}{2}u^2. \tag{2.14}$$

The largest of the three bounds (2.12), (2.13), and (2.14) is the second one. It is less than $\frac{3}{2}u^2 + 4u^3$ as soon as $u \leq \frac{1}{16}$. This proves the theorem. □

The bound given by Theorem 2.2.1 is very sharp. The largest error we have found so far when performing many random tests is about $1.5u^2$. An example of input values that would lead to an error of that magnitude is given in Example 2.2.2.

Example 2.2.2. *If we have as input for Algorithm 10, with binary32 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 8388609, \\ x_\ell &= 4095/2^{13}, \text{ and} \\ y &= 8389633, \end{aligned}$$

then the relative error of the algorithm is

$$1.4993282\ldots \times 2^{-48}.$$

In the QD library [35] as well as in Briggs' library [13], Algorithm 11 (Figure 2.5) below is suggested for multiplying a double-word number by a floating-point one.

Algorithm 11 – DWTimesFP2(x_h, x_ℓ, y).

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

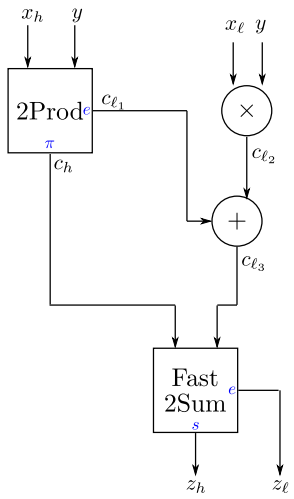


Figure 2.5 – Graphical representation of Algorithm 11. In the 2Prod and Fast2Sum calls the sum s or the product π are outputted downwards and the error e to the right.

Indeed, Algorithm 11 is faster than Algorithm 10 (we save one call to Fast2Sum), but it is less accurate. One can easily find values for x and y for which the error attained using Algorithm 11 is larger than the bound given by Theorem 2.2.1 (see Example 2.2.3).

Example 2.2.3. *If we have as input for Algorithm 11, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4525788557405064, \\ x_\ell &= 8595672275350437/2^{54}, \text{ and} \\ y &= 5085664955107621, \end{aligned}$$

then the relative error of the algorithm is

$$2.517\ldots \times 2^{-106}.$$

The relative error bound we are going to prove for Algorithm 11 is the one stated in Theorem 2.2.4 that follows, for which the proof is very similar to (in fact, simpler than) the proof of Theorem 2.2.1.

Theorem 2.2.4. *If $p \geq 3$, the relative error of Algorithm 11 (DWTimesFP2) is less than or equal to $3u^2$.*

Proof. Without loss of generality, we can assume $1 \leq x_h \leq 2-2u$ and $1 \leq y \leq 2-2u$. Since the analysis of the case $y = 1$ is straightforward, we even assume $1+2u \leq y \leq 2-2u$. This implies $1+2u \leq x_h y \leq 4-8u+4u^2$, thus $1+2u \leq c_h \leq 4-8u$ and $|c_{\ell 1}| \leq 2u$. From $|x_\ell| \leq u$ and $y \leq 2-2u$ we deduce $|c_{\ell 2}| \leq 2u-2u^2$, so that $\epsilon_1 = x_\ell y - c_{\ell 2}$ satisfies $|\epsilon_1| \leq u^2$.

Now, $|c_{\ell 1} + c_{\ell 2}| \leq 4u - 2u^2$, hence $|c_{\ell 3}| \leq 4u$, and $c_{\ell 3} = c_{\ell 1} + c_{\ell 2} + \epsilon_2$, with $|\epsilon_2| \leq 2u^2$. From $|c_{\ell 3}| \leq 4u$ and $c_h \geq 1+2u$ we deduce that Algorithm Fast2Sum introduces no error at line 4 of the algorithm.

Further more,

$$z_h + z_\ell = c_h + c_{\ell 3} = xy - \epsilon_1 + \epsilon_2,$$

and $|-\epsilon_1 + \epsilon_2| \leq 3u^2$. Since $xy \geq (x_h - u)y \geq (1-u)(1+2u) \geq 1$, we deduce that the relative error of Algorithm 11 is less than $3u^2$. □

Algorithm 11 can be improved both in speed and accuracy if an FMA instruction is available. This is done by merging the multiplication in line 2 with the addition in line 3. This results in Algorithm 12 bellow, with the corresponding Theorem 2.2.5.

Algorithm 12 – DWTimesFP3(x_h, x_ℓ, y).

1: $(c_h, c_{\ell 1}) \leftarrow$ 2ProdFMA(x_h, y)
2: $c_{\ell 3} \leftarrow$ fma$(x_\ell, y, c_{\ell 1})$
3: $(z_h, z_\ell) \leftarrow$ Fast2Sum$(c_h, c_{\ell 3})$
4: **return** (z_h, z_ℓ)

Theorem 2.2.5. *If $p \geq 3$, the relative error of Algorithm 12 (DWTimesFP3) is less than or equal to $2u^2$.*

The proof of Theorem 2.2.5 is very similar to the proof of Theorem 2.2.4, so we omit it. We do believe that the bound is very sharp, since the largest error that we obtained through many random tests is the one presented in Example 2.2.6.

Example 2.2.6. *If we have as input for Algorithm 12, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4505619370757448, \\ x_\ell &= -9003265529542491/2^{54}, \text{ and} \\ y &= 4511413997183120, \end{aligned}$$

then the relative error of the algorithm is

$$1.984\ldots \times 2^{-106}.$$

2.2.2 Multiplication of two double-word numbers

Algorithm 13 (Figure 2.6) below was first suggested by Dekker (under the name *mul2* in [22]), with the only difference that he always used Algorithm 5 (Dekker's product) for getting the result and error of a floating-point multiplication. This algorithm is also the one that has been implemented in the QD library [35] and in Briggs' library [13].

Dekker proved a relative error bound of $11 \cdot 2^{-2p}$, but we were able to significantly improve that bound. The result that we obtained is the one stated by Theorem 2.2.8. In the proof we will make use of Lemma 2.2.7, for which the proof is straightforward calculus, this is why we do not include it.

Algorithm 13 – DWTimesDW1$(x_h, x_\ell, y_h, y_\ell)$.

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)

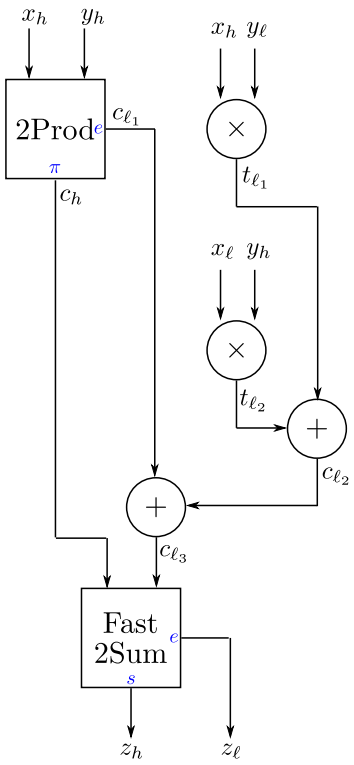


Figure 2.6 – Graphical representation of Algorithm 13. In the 2Prod and Fast2Sum calls the sum s or the product π are outputted downwards and the error e to the right.

Lemma 2.2.7. *Let x and y be two positive real numbers. If $xy \leq 2$, $x \geq 1$ and $y \geq 1$, then $x + y \leq 2\sqrt{2}$.*

Theorem 2.2.8. *If $p \geq 4$, the relative error of Algorithm 13 (DWTimesDW1) is less than or equal to*

$$\frac{7u^2}{(1+u)^2} < 7u^2.$$

Proof. Without loss of generality, we assume that $1 \leq x_h \leq 2 - 2u$ and $1 \leq y_h \leq 2 - 2u$. As a deduction $x_h y_h < 4$, and

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

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

$$t_{\ell 1} = x_h y_\ell + \epsilon_1,$$

with $|x_h y_\ell| \leq 2u - 2u^2$, so that $|t_{\ell 1}| \leq 2u - 2u^2$ and $|\epsilon_1| \leq u^2$; and

$$t_{\ell 2} = x_\ell y_h + \epsilon_2,$$

with $|x_\ell y_h| \leq 2u - 2u^2$, so that $|t_{\ell 2}| \leq 2u - 2u^2$ and $|\epsilon_2| \leq u^2$. Now, we have

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

with $|t_{\ell 1} + t_{\ell 2}| \leq 4u - 4u^2$, which implies $|c_{\ell 2}| \leq 4u - 4u^2$ and $|\epsilon_3| \leq 2u^2$. We finally obtain

$$c_{\ell 3} = c_{\ell 1} + c_{\ell 2} + \epsilon_4,$$

and from $|c_{\ell 1} + c_{\ell 2}| \leq 6u - 4u^2$, we deduce $|c_{\ell 3}| \leq 6u$. Hence $|\epsilon_4| \leq 4u^2$ and, since $c_h \geq 1$, Algorithm Fast2Sum introduces no error at line 6 of the algorithm.Therefore,

$$\begin{aligned} z_h + z_\ell &= c_h + c_{\ell 3} \\ &= (x_h y_h - c_{\ell 1}) + c_{\ell 1} + c_{\ell 2} + \epsilon_4 \\ &= x_h y_h + t_{\ell 1} + t_{\ell 2} + \epsilon_3 + \epsilon_4 \\ &= x_h y_h + x_h y_\ell + x_\ell y_h + \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4 \\ &= xy - x_\ell y_\ell + \epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4 \\ &= xy + \eta, \end{aligned} \tag{2.15}$$

with $|\eta| \leq u^2 + |\epsilon_1 + \epsilon_2 + \epsilon_3 + \epsilon_4| \leq 9u^2$. On top of that, $x_h \geq 1$ and $y_h \geq 1$ imply $xy \geq (1 - \frac{u}{2})^2$, so that we can first deduce a relative error bound $9u^2/(1 - \frac{u}{2})^2$. That bound can be improved by looking at two different cases.

1. If $x_h y_h > 2$. Since $x \geq x_h - u$ and $y \geq y_h - u$, we have $xy \geq x_h y_h - u(x_h + y_h) + u^2 > 2 - u(4 - 4u) + u^2 = 2 - 4u + 5u^2$. Hence the relative error is bounded by

$$\frac{9u^2}{2 - 4u + 5u^2} = \frac{9}{2}u^2 + 9u^3 + \frac{27}{4}u^4 + \cdots. \tag{2.16}$$

2. If $x_h y_h \leq 2$. We now obtain $|c_{\ell 1}| \leq u$. Furthermore, Lemma 2.2.7 implies

$$x_h + y_h \leq 2\sqrt{2}. \tag{2.17}$$

We have,

$$|t_{\ell 1}| = |\operatorname{RN}(x_h y_\ell)| \leq \operatorname{RN}(x_h u) = x_h u,$$

and, similarly, $|t_{\ell 2}| \leq y_h u$, so that, using (2.17),

$$|t_{\ell 1} + t_{\ell 2}| \leq x_h u + y_h u \leq 2\sqrt{2}u. \tag{2.18}$$

Since $2\sqrt{2}u$ is between $2u$ and $4u$, (2.18) gives $|\epsilon_3| \leq 2u^2$ (i.e., the same bound on $|\epsilon_3|$ as previously). Hence $c_{\ell 2}$ satisfies

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

Consequently

$$|c_{\ell 1} + c_{\ell 2}| \leq u \cdot (2\sqrt{2} + 1) + 2u^2.$$

If $p \geq 4$ (i.e., $u \leq \frac{1}{16}$), this new bound is always less than $4u$. Therefore, we now have $|\epsilon_4| \leq 2u^2$. In (2.15), this will result in $|\eta| \leq 7u^2$ instead of $9u^2$. Again, using $xy \geq (1 - \frac{u}{2})^2$, we can deduce a

relative error bound $\frac{7u^2}{(1-\frac{u}{2})^2}$. However, that error bound can be made slightly smaller by noticing that if $x_h = 1$ or $y_h = 1$ then, either $\epsilon_1 = 0$ or $\epsilon_2 = 0$, which results in a significantly smaller bound for $|\eta|$. So we can assume that $x_h \geq 1 + 2u$ (hence, $x > 1 + u$) and $y_h \geq 1 + 2u$ (hence, $y > 1 + u$). Therefore the relative error is bounded by

$$\frac{7u^2}{(1+u)^2} < 7u^2. \tag{2.19}$$

One easily notices that if $p \geq 4$ the bound (2.16) is less than the bound (2.19). □

The bound provided by Theorem 2.2.8 is probably too pessimistic. The largest relative error we have encountered in our tests using binary32 and binary64 formats are the ones given in Example 2.2.9 and Example 2.2.10, respectively.

Example 2.2.9. *If we have as input for Algorithm 13, with binary32 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 8399376, \\ x_\ell &= 16763823/2^{25}, \\ y_h &= 8414932, \text{ and} \\ y_\ell &= 16756961/2^{25}, \end{aligned}$$

then the relative error of the algorithm is

$$4.947 \times 2^{-48}.$$

Example 2.2.10. *If we have as input for Algorithm 13, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4508231565242345, \\ x_\ell &= -9007199254524053/2^{54}, \\ y_h &= 4504969740576150, \text{ and} \\ y_\ell &= -4503599627273753/2^{53}, \end{aligned}$$

then the relative error of the algorithm is

$$4.9916 \times 2^{-106}.$$

As in the case of multiplication of a double-word number by a floating-point number, Algorithm 13 can also be improved if an FMA instruction is available, by merging lines 3 and 4, as showed in Algorithm 14 below, with the corresponding error bound given in Theorem 2.2.11. This way we save one floating-point operation and we are able to slightly improve our error bound.

Algorithm 14 – DWTimesDW2$(x_h, x_\ell, y_h, y_\ell)$.

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

Theorem 2.2.11. *If $p \geq 5$, the relative error of Algorithm 14 (DWTimesDW2) is less than or equal to*

$$\frac{6u^2 + \frac{1}{2}u^3}{(1+u)^2} < 6u^2.$$

Proof. The proof is very similar to the proof of Theorem 2.2.8, and follows the same structure, so we do not detail it. The major changes are:

- the term ϵ_2 of the proof of Theorem 2.2.8 no longer exists;
- $c_{\ell 2} = x_\ell y_h + t_\ell + \epsilon_3$, where $\epsilon_3 \leq 2u^2$,
- instead of (2.15), we now have

$$z_h + z_\ell = xy - x_\ell y_\ell + \epsilon_1 + \epsilon_3 + \epsilon_4 = xy + \eta,$$

with $|\eta| \leq u^2 + |\epsilon_1 + \epsilon_3 + \epsilon_4| \leq 8u^2$.

□

We do not know if the bound given by Theorem 2.2.11 is optimal, because the largest errors we have obtained during intensive random testing are the ones presented in Example 2.2.12 and Example 2.2.13 below.

Example 2.2.12. *If we have as input for Algorithm 14, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 8404039, \\ x_\ell &= -8284843/2^{24}, \\ y_h &= 8409182, \text{ and} \\ y_\ell &= -4193899/2^{23}, \end{aligned}$$

then the relative error of the algorithm is

$$4.936 \times 2^{-48}.$$

Example 2.2.13. *If we have as input for Algorithm 14, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4515802244422058, \\ x_\ell &= -2189678420952711/2^{52}, \\ y_h &= 4503988428047019, \text{ and} \\ y_\ell &= -2248477851812015/2^{52}, \end{aligned}$$

then the relative error of the algorithm is

$$4.9433 \times 2^{-106}.$$

The multiplication algorithm for two double-word numbers can be improved even further, by taking into account also the partial product $x_\ell y_\ell$. The algorithm that does this is shown in Algorithm 15, with the corresponding error bound in Theorem 2.2.14.

Algorithm 15 – DWTimesDW3$(x_h, x_\ell, y_h, y_\ell)$.

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

Theorem 2.2.14. *If $p \geq 4$, the relative error of Algorithm 15 (DWTimesDW3) is less than or equal to*

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

Proof. The proof is very similar to the proof of Theorem 2.2.8, and follows the same structure, so we do not detail it. The major changes are:

- $t_{\ell 1} = x_h y_\ell + x_\ell y_\ell + \epsilon_1$, where ϵ_1, now, is bounded by $u^2 + \frac{u^3}{2}$,
- the term ϵ_2 of the proof of Theorem 2.2.8 no longer exists;
- instead of (2.15), we now have

$$z_h + z_\ell = xy + \epsilon_1 + \epsilon_3 + \epsilon_4 = xy + \eta,$$

with $|\eta| \leq |\epsilon_1 + \epsilon_3 + \epsilon_4| \leq 7u^2 + \frac{u^3}{2}$.

□

We do not know if the bound given by Theorem 2.2.11 is optimal. The largest relative error we have encountered so far in intensive tests is the one presented in Example 2.2.15.

Example 2.2.15. *If we have as input for Algorithm 14, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4510026974538724, \\ x_\ell &= 4232862152422029/2^{53}, \\ y_h &= 4511576932111935, \text{ and} \\ y_\ell &= 2250098448199619/2^{52}, \end{aligned}$$

then the relative error of the algorithm is

$$3.936 \times 2^{-106}.$$

2.3 Division of double-word numbers

The same as for the double-word multiplication, the division algorithms follow the paper-and-pencil technique, this is why they are called long division algorithms. More specifically, when computing the division of two real numbers, $x \div y$, we first aproximate the quotient $q_0 = \frac{x}{y}$, then compute the reminder $r = x - q_0 y$, and we finish by computing a second correction term $q_1 = \frac{r}{y}$. In what follows we show how we apply this techique to double-word division.

2.3.1 Division of a double-word number by a floating-point number

The algorithm suggested by Li et al. in [57] for dividing a double-word number by a floating-point number is Algorithm 16 below.

Let us notice that Algorithm 16 can be simplified. We have $t_h = \frac{x_h}{y}(1+\epsilon_0)$ and $\pi_h = t_h y(1+\epsilon_1)$, with $|\epsilon_0|, |\epsilon_1| \leq u$. Hence,

$$(1-u)^2 x_h \leq \pi_h \leq (1+u)^2 x_h.$$

Therefore, as soon as $p \geq 2$ (i.e., $u \leq \frac{1}{4}$), π_h is within a factor 2 from x_h, and Sterbenz Lemma (Lemma 1.1.2) implies that $x_h - \pi_h$ is an exact floating-point number. As a deduction, we always have $\delta' = 0$ and $\delta_\ell = \delta'' = \mathrm{RN}(x_\ell - \pi_\ell)$, so line 3 of the algorithm can be replaced by a simple subtraction. As a consequence, Algorithm 16 can be simplified into Algorithm 17 (Figure 2.7)

Algorithm 16 – DWDivFP1(x_h, x_ℓ, y).

1: $t_h \leftarrow \mathrm{RN}(x_h/y)$
2: $(\pi_h, \pi_\ell) \leftarrow \mathsf{2Prod}(t_h, y)$
3: $(\delta_h, \delta') \leftarrow \mathsf{2Sum}(x_h, -\pi_h)$
4: $\delta'' \leftarrow \mathrm{RN}(x_\ell - \pi_\ell)$
5: $\delta_\ell \leftarrow \mathrm{RN}(\delta' + \delta'')$
6: $\delta \leftarrow \mathrm{RN}(\delta_h + \delta_\ell)$
7: $t_\ell \leftarrow \mathrm{RN}(\delta/y)$
8: $(z_h, z_\ell) \leftarrow \mathsf{Fast2Sum}(t_h, t_\ell)$
9: **return** (z_h, z_ℓ)

below, that is mathematically equivalent (always returns the same result) while being significantly simpler.

Algorithm 17 – DWDivFP2(x_h, x_ℓ, y).

1: $t_h \leftarrow \mathrm{RN}(x_h/y)$
2: $(\pi_h, \pi_\ell) \leftarrow \mathsf{2Prod}(t_h, y)$
3: $\delta_h \leftarrow x_h - \pi_h$ //exact operation
4: $\delta_\ell \leftarrow \mathrm{RN}(x_\ell - \pi_\ell)$
5: $\delta \leftarrow \mathrm{RN}(\delta_h + \delta_\ell)$
6: $t_\ell \leftarrow \mathrm{RN}(\delta/y)$
7: $(z_h, z_\ell) \leftarrow \mathsf{Fast2Sum}(t_h, t_\ell)$
8: **return** (z_h, z_ℓ)

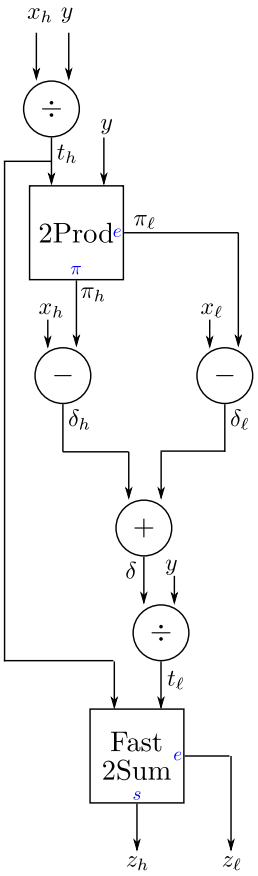


Figure 2.7 – Graphical representation of Algorithm 17. In the 2Prod and Fast2Sum calls the sum s or the product π are outputted downwards and the error e to the right.

The authors of [57] claim that their binary64 implementation of Algorithm 16 has a relative error bounded by $4 \cdot 2^{-106}$. That bound can be slightly improved. The bound that we are going to prove is given in Theorem 2.3.1 and it holds for both algorithms, given the fact that they return the same result.

Theorem 2.3.1. *If $p \geq 4$ and $y \neq 0$, the relative error of Algorithm 16 (DWDivFP1) and Algorithm 17 (DWDivFP2) is less than or equal to $3.5u^2$.*

In the proof of Theorem 2.3.1 we make use of Lemma 2.3.2 for which we also give a short proof.

Lemma 2.3.2. *Assume a radix-2, precision-p, floating-point arithmetic. Let x and y be floating-point numbers between 1 and 2. The distance between $\mathrm{RN}(\frac{x}{y})$ and $\frac{x}{y}$ is less than*

$$\begin{cases} u - \frac{2u^2}{y} & \text{if } \frac{x}{y} \geq 1; \\ \frac{u}{2} - \frac{u^2}{y} & \text{otherwise.} \end{cases} \tag{2.20}$$

Proof. It suffices to estimate the smallest possible distance between $\frac{x}{y}$ and a "midpoint" (i.e., a number exactly halfway between two consecutive floating-point numbers). Let $x = M_x \cdot 2^{-p+1}$, $y = M_y \cdot 2^{-p+1}$, with $2^{p-1} \leq M_x, M_y \leq 2^p - 1$.

- If $\frac{x}{y} \geq 1$, a midpoint μ between 1 and 2 has the form $\frac{2M_\mu+1}{2^p}$, with $2^{p-1} \leq M_\mu \leq 2^p - 1$. We have
$$\left|\frac{x}{y} - \mu\right| = \left|\frac{2^p M_x - M_y(2M\mu + 1)}{2^p M_y}\right|.$$
The numerator, $2^p M_x - M_y(2M\mu + 1)$, of that fraction cannot be zero: since $2M\mu + 1$ is odd, having $2^p M_x = M_y(2M\mu + 1)$ would require M_y to be a multiple of 2^p, which is impossible since $M_y \leq 2^p - 1$. Hence that numerator has absolute value at least 1. And so
$$\left|\frac{x}{y} - \mu\right| \geq \frac{1}{2^p M_y} = \frac{2u^2}{y}.$$
- If $\frac{x}{y} < 1$ the proof is similar. The only change is that a midpoint is of the form $\frac{2M_\mu+1}{2^{p+1}}$.

□

Note that in a more recent result [39, Table 1], a similar bound obtained for division could be used instead of Lemma 2.3.2, but we include it for completeness. Let us now prove Theorem 2.3.1.

Proof. (Of Theorem 2.3.1) The case where y is a power of 2 is straightforward, so we omit it. Without loss of generality, we assume $1 \leq x_h \leq 2 - 2u$, so that $|x_l| \leq u$, and $1 + 2u \leq y \leq 2 - 2u$. There on
$$\frac{1}{2-2u} \leq \frac{x_h}{y} \leq \frac{2-2u}{1+2u}. \tag{2.21}$$
The quotient $\frac{1}{2-2u}$ is always larger than $\frac{1}{2} + \frac{u}{2}$, and, as soon as $p \geq 4$, $\frac{2-2u}{1+2u}$ is less than $2 - 5u$. Therefore
$$\frac{1}{2} + u \leq t_h = \mathrm{RN}\left(\frac{x_h}{y}\right) \leq 2 - 6u. \tag{2.22}$$

1. If $x \geq y$, implying $x_h \geq y$ and $t_h \geq 1$, then Lemma 2.3.2 implies
$$\left|t_h - \frac{x_h}{y}\right| \leq u - \frac{2u^2}{y},$$
hence $|t_h y - x_h| \leq uy - 2u^2 \leq 2u - 4u^2$. Consequently
$$\pi_h = \mathrm{RN}(t_h y) \in \{x_h - 2u, x_h, x_h + 2u\}.$$

One might think that π_h could be $x_h - u$ in the case $x_h = 1$, but $x_h = 1$ is not compatible with our assumptions, $x \geq y$ and $y \geq 1 + 2u$. We also have,

$$|t_h y| \leq |t_h y - x_h| + |x_h| \leq 2u - 4u^2 + 2 - 2u = 2 - 4u^2, \tag{2.23}$$

which implies

$$|\pi_\ell| \leq \frac{1}{2}\,\mathrm{ulp}(t_h y) = u.$$

In all cases $x_h - \pi_h \in \{-2u, 0, 2u\}$, so that $\delta_h = x_h - \pi_h$. Also, from $|\pi_\ell| \leq u$ and $|x_\ell| \leq u$, we deduce $|\pi_\ell - x_\ell| \leq 2u$. As a consequence, $|\delta_\ell| \leq 2u$, and $\epsilon_1 = \delta_\ell - (x_\ell - \pi_\ell)$ satisfies

$$|\epsilon_1| = |\delta_\ell - (x_\ell - \pi_\ell)| \leq u^2. \tag{2.24}$$

Define $\epsilon_2 = \delta - (\delta_h + \delta_\ell)$.

- If $\delta_h = -2u$ then $x_h - \pi_h = -2u$, so that

  $$\pi_\ell = t_h y - \pi_h = (t_h y - x_h) + (x_h - \pi_h)$$

  satisfies

  $$\pi_\ell \leq (2u - 4u^2) + (-2u) \leq -4u^2.$$

  Hence $-u \leq \pi_\ell \leq -4u^2$, so that $-u + 4u^2 \leq \delta_\ell \leq 2u$, which implies $-3u + 4u^2 \leq \delta_h + \delta_\ell \leq 0$. Furthermore,

  — if $u \leq \delta_\ell \leq 2u$ then Sterbenz's Lemma implies that $\delta_h + \delta_\ell$ is a floating-point number, so that $\epsilon_2 = 0$;

  — if $-u + 4u^2 \leq \delta_\ell < u$ then $-3u + 4u^2 < \delta_h + \delta_\ell < -u$, hence

  $$|\epsilon_2| = |\delta - (\delta_h + \delta_\ell)| \leq \frac{1}{2}\,\mathrm{ulp}(3u) = 2u^2.$$

  However, in that case, since $|\delta_\ell| < u$, the bound (2.24) is improved and becomes $|\epsilon_1| \leq \frac{u^2}{2}$.

  Hence, if $\delta_h = -2u$, we always have $|\epsilon_1 + \epsilon_2| \leq \frac{5u^2}{2}$.

- Symmetrically, if $\delta_h = 2u$ we also always have $|\epsilon_1 + \epsilon_2| \leq \frac{5u^2}{2}$.

- If $\delta_h = 0$ then $|\delta_h + \delta_\ell| = |\delta_\ell| \leq 2u$. Since there is no error when adding δ_h and δ_ℓ, we have $\epsilon_2 = 0$.

Hence,

$$\begin{aligned} \delta &= (x_h - \pi_h) + (x_\ell - \pi_\ell) + \underbrace{\delta_\ell - (x_\ell - \pi_\ell)}_{\epsilon_1} + \underbrace{\delta - (\delta_h + \delta_\ell)}_{\epsilon_2}, \\ &= x - t_h y + \epsilon \end{aligned}$$

with $|\epsilon| = |\epsilon_1 + \epsilon_2| \leq \frac{5u^2}{2}$. We deduce

$$\frac{\delta}{y} = \frac{x}{y} - t_h + \frac{\epsilon}{y}. \tag{2.25}$$

Let us now bound the error committed when rounding $\frac{\delta}{y}$. For that purpose, we first try to find a reasonably tight bound on $\frac{\delta}{y}$ (tighter than the obvious bound we would obtain by dividing the upper bound $3u^2$ on δ by the lower bound $1+2u$ on y). We have

$$\begin{aligned}\left|\frac{x}{y} - t_h\right| &\leq \left|\frac{x_h}{y} - t_h\right| + \left|\frac{x_\ell}{y}\right| \\ &\leq u - \frac{2u^2}{y} + \frac{u}{y},\end{aligned}$$

and

$$\left|\frac{\epsilon}{y}\right| \leq \frac{5u^2}{2y}.$$

Therefore, using (2.25),

$$\left|\frac{\delta}{y}\right| \leq u + \frac{u^2}{2y} + \frac{u}{y} \leq u + \frac{u^2}{2(1+2u)} + \frac{u}{1+2u} = \frac{4u+5u^2}{2+4u} < 2u.$$

Hence $|t_\ell| \leq 2u$, which means, since $t_h \geq 1$, that Algorithm Fast2Sum introduces no error at line 7 of the algorithm. Also,

$$\left|t_\ell - \frac{\delta}{y}\right| = \left|\mathrm{RN}\left(\frac{\delta}{y}\right) - \frac{\delta}{y}\right| \leq u^2.$$

Therefore, using (2.25),

$$t_\ell = \frac{x}{y} - t_h + \frac{\epsilon}{y} + \epsilon', \tag{2.26}$$

with $|\epsilon'| \leq u^2$. We finally conclude that

$$\left|(z_h + z_\ell) - \frac{x}{y}\right| = \left|(t_h + t_\ell) - \frac{x}{y}\right| \leq \frac{5u^2}{2y} + u^2, \tag{2.27}$$

so that the relative error is bounded by

$$\frac{y}{x}\left(\frac{5u^2}{2y} + u^2\right) \leq \frac{5u^2}{2x} + \frac{u^2 y}{x} \leq \frac{7u^2}{2} = 3.5u^2,$$

since $\frac{y}{x} \leq 1$.

2. If $x < y$, implying $x_h \leq y$ and $t_h \leq 1$.

We first notice that the case $x_h = y$ is easily handled. It leads to $t_h = 1$, $\pi_h = x_h$, $\pi_\ell = 0$, $\delta = x_\ell$, and $z_h + z_\ell = t_h + t_\ell = \frac{x}{y} + \eta$, with $|\eta| \leq u\frac{|x_\ell|}{y} \leq u^2\frac{x}{y}$.

We can now focus on the case $x_h < y$. Notice that this case implies $x_h \leq y - 2u$, so that $x \leq y - u$, so that $\frac{x}{y} \leq 1 - \frac{u}{y} \leq 1 - \frac{u}{2-2u} < 1 - \frac{u}{2}$, which implies $t_h \leq 1 - u$.

The remainder of the proof is very similar to the proof of the case $x > y$, so we give it with less details. Lemma 2.3.2 implies

$$\left|t_h - \frac{x_h}{y}\right| \leq \frac{u}{2} - \frac{u^2}{y},$$

so that $|t_h y - x_h| \leq \frac{u}{2} \cdot y - u^2 \leq u - 2u^2$. This implies $\pi_h = \mathrm{RN}(t_h y) \in \{x_h - u, x_h\}$, so that $\delta_h \in \{0, u\}$. The case $\mathrm{RN}(t_h y) = x_h - u$ (i.e., $\delta_h = u$) being possible only when $x_h = 1$. We also have

$$|t_h y| \leq |t_h| \cdot |y| \leq (1-u) \cdot (2-2u) = 2 - 3u + 2u^2,$$

so that $|\pi_\ell| \leq \frac{1}{2}\,\mathrm{ulp}(t_h y) \leq u$.

As previously, define $\epsilon_1 = \delta_\ell - (x_\ell - \pi_\ell)$ and $\epsilon_2 = \delta - (\delta_h + \delta_\ell)$. From $|\pi_\ell| \leq u$ and $|x_\ell| \leq u$, we deduce $|\delta_\ell| \leq 2u$ and $\epsilon_1 \leq u^2$.

- If $\delta_h = 0$ then $\epsilon_2 = 0$.

- If $\delta_h = u$ (which implies $x_h = 1$), then, since $\pi_h = 1 - u$, we have $t_h y < 1$. This implies $|\pi_\ell| \leq \frac{1}{2}\,\mathrm{ulp}(t_h y) \leq \frac{u}{2}$. We also have
$$\pi_\ell = t_h y - \pi_h = (t_h y - x_h) + (x_h - \pi_h) \geq -u + 2u^2 + u = 2u^2,$$
hence,
$$2u^2 \leq \pi_\ell \leq \frac{u}{2}. \tag{2.28}$$
Also, $x_h = 1$ implies $-\frac{u}{2} \leq x_\ell \leq u$. Therefore $-u \leq x_\ell - \pi_\ell \leq u - 2u^2$, so that $-u \leq \delta_\ell \leq u - 2u^2$, and $|\epsilon_1| \leq \frac{u^2}{2}$. From
$$0 \leq \delta_\ell + \delta_h \leq 2u - 2u^2,$$
we deduce $|\epsilon_2| \leq u^2$.

Hence,
$$\begin{aligned}\delta &= (x_h - \pi_h) + (x_\ell - \pi_\ell) + \underbrace{\delta_\ell - (x_\ell - \pi_\ell)}_{\epsilon_1} + \underbrace{\delta - (\delta_h + \delta_\ell)}_{\epsilon_2} \\ &= x - t_h y + \epsilon,\end{aligned}$$
with $|\epsilon| = |\epsilon_1 + \epsilon_2| \leq \frac{3u^2}{2}$. We deduce
$$\left|\frac{\delta}{y}\right| \leq \left|\frac{x_h}{y} - t_h\right| + \left|\frac{x_\ell}{y}\right| + \left|\frac{\epsilon}{y}\right| \leq \frac{u}{2} + \frac{2u^2}{y} + \frac{u}{y} < 2u,$$
so that $|t_\ell| \leq 2u$. Hence, since $t_h \geq \frac{1}{2} + u$, Algorithm Fast2Sum introduces no error at line 7, and
$$\left|t_\ell - \frac{\delta}{y}\right| \leq u^2.$$
Therefore
$$t_\ell = \frac{x}{y} - t_h + \eta,$$
with
$$|\eta| \leq \frac{3u^2}{2y} + u^2,$$
hence $z_h + z_\ell = t_h + t_\ell$ approximates $\frac{x}{y}$ with a relative error bounded by
$$\frac{y}{x}\left(\frac{3u^2}{2y} + u^2\right) \leq \frac{3u^2}{2x} + u^2\frac{y}{x} \leq 3.5u^2.$$

□

The above bound is reasonably sharp, and this is showed by Example 2.3.3, for which we obtained the largest relative error while doing many random computations.

Example 2.3.3. *If we have as input for Algorithm 16 or Algorithm 17, with binary64 as underlying arithmetic, the values:*
$$\begin{aligned}x_h &= 4588860379563012, \\ x_\ell &= -4474949195791253/2^{53}, \text{ and} \\ y &= 4578284000230917,\end{aligned}$$
then the relative error of the algorithm is
$$2.95157083\ldots \times 2^{-106}.$$

2.3.2 Division of two double-word numbers

The algorithm implemented in the QD library for dividing two double-word numbers is Algorithm 18. It follows the same structure as Algorithm 16, the only difference is that the reminder is computed using Algorithm 10 (DWTimesFP1) for multiplying a double-word number by a floating-point one.

Algorithm 18 – DWDivDW1$(x_h, x_\ell, y_h, y_\ell)$.

1: $t_h \leftarrow \mathrm{RN}(x_h/y_h)$
2: $(r_h, r_l) \leftarrow$ DWTimesFP1(y_h, y_ℓ, t_h) //approximation to $(y_h + y_\ell)t_h$ with relative error $\leq 1.5u^2 + 4u^3$ using Alg. 10
3: $(\pi_h, \pi_\ell) \leftarrow$ 2Sum$(x_h, -r_h)$
4: $\delta_h \leftarrow \mathrm{RN}(\pi_\ell - r_\ell)$
5: $\delta_\ell \leftarrow \mathrm{RN}(\delta_h + x_\ell)$
6: $\delta \leftarrow \mathrm{RN}(\pi_h + \delta_\ell)$
7: $t_\ell \leftarrow \mathrm{RN}(\delta/y_h)$
8: $(z_h, z_\ell) \leftarrow$ Fast2Sum(t_h, t_ℓ)
9: **return** (z_h, z_ℓ)

Let us quickly analyze the beginning of Algorithm 18. This will lead us to suggest another algorithm, faster yet mathematically equivalent (as soon as $p \geq 3$, it always returns the very same result). Without loss of generality, we assume $x_h > 0$ and $y_h > 0$. Define ϵ_x and ϵ_y such that $x_h = x(1+\epsilon_x)$ and $y_h = \frac{y}{1+\epsilon_y}$. These two numbers ϵ_x and ϵ_y have an absolute value less than or equal to u. We have

$$t_h = \frac{x_h}{y_h}(1+\epsilon_0), \text{ with } |\epsilon_0| \leq u, \tag{2.29}$$

and, from Theorem 2.2.1,

$$r_h + r_\ell = t_h y(1+\eta), \text{ with } |\eta| \leq \frac{3}{2}u^2 + 4u^3. \tag{2.30}$$

There exists $|\epsilon_1| \leq u$ such that $r_h = (r_h + r_\ell)(1+\epsilon_1)$. This can be rewritten $r_\ell = -\epsilon_1(r_h + r_\ell)$, so that, using (2.30), $r_\ell = -\epsilon_1 t_h y(1+\eta)$. We finally obtain

$$\begin{aligned} r_h &= t_h y_h (1+\epsilon_y)(1+\epsilon_1)(1+\eta) \\ &= x_h(1+\epsilon_y)(1+\epsilon_0)(1+\epsilon_1)(1+\eta), \end{aligned} \tag{2.31}$$

so that

$$(1-u)^3(1-2u^2)x_h \leq r_h \leq (1+u)^3(\frac{3}{2}u^2 + 4u^3)x_h,$$

from which we deduce

$$|x_h - r_h| \leq (3u + \frac{9}{2}u^2 + \frac{19}{2}u^3 + \frac{33}{2}u^4 + \frac{27}{2}u^5 + 4u^6) \cdot x_h.$$

This implies

$$|x_h - r_h| \leq (3u + 6u^2) \cdot x_h \tag{2.32}$$

as soon as $p \geq 3$. One easily checks that for $p \geq 3$ (i.e., $u \leq \frac{1}{8}$), $3u + 6u^2$ is less than $\frac{1}{2}$. Hence, From Sterbenz Lemma (Lemma 1.1.2), the number $x_h - r_h$ is an exact floating-point number. Therefore, the number π_ℓ obtained at line 3 of the algorithm is always 0 and that line can be replaced by a simple, errorless, subtraction. This gives $\pi_h = x_h - r_h$, and $\delta_h = -r_\ell$. Thus, without changing the final result, we can replace Algorithm 18 by the simpler Algorithm 19 (Figure 2.8), below.

Algorithm 19 – DWDivDW2$(x_h, x_\ell, y_h, y_\ell)$.

1: $t_h \leftarrow \mathrm{RN}(x_h/y_h)$
2: $(r_h, r_l) \leftarrow$ DWTimesFP1(y_h, y_ℓ, t_h) //approximation to $(y_h + y_\ell)t_h$ with relative error $\leq 1.5u^2 + 4u^3$ using Alg. 10
3: $\pi_h \leftarrow x_h - r_h$ //exact operation
4: $\delta_\ell \leftarrow \mathrm{RN}(x_\ell - r_\ell)$
5: $\delta \leftarrow \mathrm{RN}(\pi_h + \delta_\ell)$
6: $t_\ell \leftarrow \mathrm{RN}(\delta/y_h)$
7: $(z_h, z_\ell) \leftarrow$ Fast2Sum(t_h, t_ℓ)
8: **return** (z_h, z_ℓ)

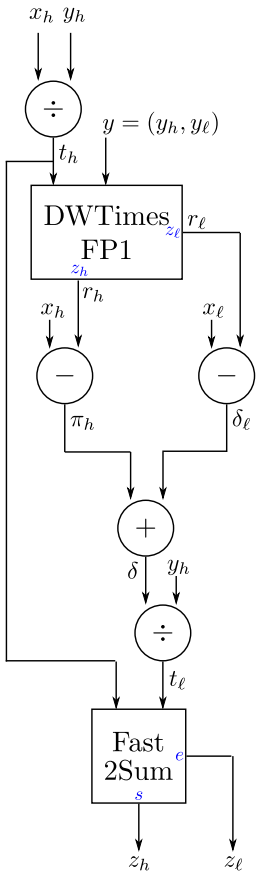


Figure 2.8 – Graphical representation of Algorithm 19. In the Fast2Sum and DWTimesFP1 calls the sum s or z_h is outputted downwards and the error e or z_l to the right.

If an FMA instruction is available, Algorithm 12 can be used at line 2 instead of Algorithm 10 without changing much the proof of Theorem 2.3.4 below.

Theorem 2.3.4. *If $p \geq 7$ and $y \neq 0$, the relative error of Algorithms 18 (DWDivDW1) and Algorithm 19 (DWDivDW2) is upper-bounded by $15u^2 + 56u^3$.*

Proof. For reasons of symmetry, we can assume that x and y are positive. We will use the results (2.29) to (2.32) obtained when analyzing the beginning of Algorithm 18. Assume $p \geq 7$. It follows

$$\delta_\ell = (x_\ell - r_\ell)(1 + \epsilon_2), \text{ with } |\epsilon_2| \leq u,$$

We have $|x_\ell| \leq u x_h$ and $|r_\ell| \leq u r_h$, so that

$$\begin{aligned} |x_\ell - r_\ell| &\leq |x_\ell| + |r_\ell| \\ &\leq u x_h + u r_h \\ &\leq u x_h + u\left((r_h - x_h) + x_h\right) \\ &\leq u x_h + u\left(|r_h - x_h| + x_h\right). \end{aligned}$$

Therefore, using (2.32) (which holds since we assume $p \geq 7$),

$$|x_\ell - r_\ell| \leq u x_h + u\left((3u + 6u^2)x_h + x_h\right),$$

which gives

$$|x_\ell - r_\ell| \leq (2u + 3u^2 + 6u^3)x_h. \tag{2.33}$$

We have

$$\delta = (\pi_h + \delta_\ell)(1 + \epsilon_3), \text{ with } |\epsilon_3| \leq u,$$

so that

$$\begin{aligned} \delta &= x_h - r_h + x_\ell - r_\ell + (x_\ell - r_\ell)(\epsilon_2 + \epsilon_3 + \epsilon_2\epsilon_3) + (x_h - r_h)\epsilon_3, \\ &= x - (r_h + r_\ell) + \alpha x_h, \end{aligned}$$

with (using (2.32) and (2.33))

$$\begin{aligned} |\alpha| &\leq (2u + 3u^2 + 6u^3)(2u + u^2) + (3u + 6u^2)u \\ &\leq 7u^2 + 15u^3 \end{aligned} \tag{2.34}$$

as soon as $p \geq 4$. Hence $\delta = x - t_h y(1 + \eta) + \alpha x_h$, so that

$$\frac{\delta}{y_h} = \frac{x - t_h y}{y} \cdot \frac{y}{y_h} - \frac{\eta t_h y}{y_h} + \alpha \frac{x_h}{y_h}. \tag{2.35}$$

The number $x - t_h y$ is equal to $x_h - t_h y_h + x_\ell - t_h y_\ell$. From (2.29), $x_h - t_h y_h$ is equal to $-x_h\epsilon_0$. Also, $|x_\ell|$ is less than or equal to ux_h, and

$$|t_h y_\ell| \leq |u t_h y_h| \leq u(1 + u)x_h.$$

Hence,

$$|x - t_h y| \leq x_h(u + u + u(1 + u)) = x_h(3u + u^2). \tag{2.36}$$

From (2.35), we deduce

$$\begin{aligned} \tfrac{\delta}{y_h} &= \tfrac{x - t_h y}{y}(1 + \epsilon_y) - \eta t_h(1 + \epsilon_y) + \alpha \tfrac{x_h}{y_h}, \\ &= \tfrac{x - t_h y}{y} + \beta, \end{aligned} \tag{2.37}$$

with

$$\begin{aligned} |\beta| &= \left| \epsilon_y \tfrac{x - t_h y}{y} - t_h(1 + \epsilon_y)\eta + \tfrac{x_h}{y_h} \right| \\ &\leq u(3u + u^2)\tfrac{x_h}{y} + (1 + u)(2u^2)\tfrac{x_h}{y_h} + (7u^2 + 15u^3)\tfrac{x_h}{y_h} \\ &\leq u(3u + u^2)(1 + u)\tfrac{x}{y} + (1 + u)^3(2u^2)\tfrac{x}{y} + (7u^2 + 15u^3)(1 + u)^2\tfrac{x}{y} \\ &= \left(12u^2 + 39u^3 + 44u^4 + 17u^5\right)\tfrac{x}{y}. \end{aligned} \tag{2.38}$$

Hence,

$$\begin{aligned} t_\ell &= \mathrm{RN}\left(\tfrac{\delta}{y_h}\right) \\ &= \tfrac{\delta}{y_h}(1 + \epsilon_4) \text{ with } |\epsilon_4| \leq u, \\ &= \left(\tfrac{x - t_h y}{y} + \beta\right)(1 + \epsilon_4) \\ &= \tfrac{x - t_h y}{y} + \gamma, \end{aligned} \tag{2.39}$$

with

$$\begin{aligned} |\gamma| &= \left|\tfrac{x-t_h y}{y}\epsilon_4 + \beta + \epsilon_4\beta\right| \\ &\leq \tfrac{x_h}{y}(3u+u^2)u + \beta + \beta u \\ &\leq (3u+u^2)u(1+u)\tfrac{x}{y} + \beta + \beta u \\ &= \left(15u^2+55u^3+84u^4+61u^5+17u^6\right)\tfrac{x}{y}. \end{aligned} \tag{2.40}$$

Hence

$$t_h + t_\ell = \frac{x}{y} + \gamma.$$

Since we straightforwardly have

$$t_h \geq \frac{x}{y}(1-u)^3, \tag{2.41}$$

we deduce

$$|t_\ell| \leq \frac{x}{y}\left((15u^2+55u^3+84u^4+61u^5+17u^6) + (3u-3u^2+u^3)\right). \tag{2.42}$$

From (2.41) and (2.42) we easily deduce that as soon as $p \geq 4$ (i.e., $u \leq 1/16$), t_h is larger than $|t_\ell|$, so that Algorithm Fast2Sum introduces no error at line 7 of the algorithm. Therefore,

$$z_h + z_\ell = t_h + t_\ell = \frac{x}{y} + \gamma,$$

so that the relative error is upper-bounded by

$$15u^2+55u^3+84u^4+61u^5+17u^6,$$

which is less than $15u^2+56u^3$ as soon as $p \geq 7$ (i.e., $u \leq 1/128$), which always holds in practice. □

The bound provided by Theorem 2.3.4 is almost certainly not optimal. However, during many random tests, we have encountered cases (Example 2.3.5) for which the relative error, although significantly less than the bound $15u^2+56u^3$, remains of a similar order of magnitude, i.e., more than half the bound.

Example 2.3.5. *If we have as input for Algorithm 18 or Algorithm 19, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4503607118141812, \\ x_\ell &= 4493737176494969/2^{53}, \\ y_h &= 4503600552333684, \text{ and} \\ y_\ell &= -562937972998161/2^{50}, \end{aligned}$$

then the relative error of the algorithm is

$$8.465\ldots \times 2^{-106}.$$

Now, notice that if an FMA instruction is available, it is possible to design an even more accurate algorithm. This is possible due to Property 2.3.6 that is easy to prove, and common knowledge among the designers of Newton-Raphson based division algorithms (we will extensively treat this subject in Chapter 3, Section 3.5).

Property 2.3.6. *If x is a nonzero floating-point number, and if $t = \mathrm{RN}(\frac{1}{x})$, then $xt-1$ is a floating-point number.*

Proof. Without loss of generality we assume $1 \leq x \leq 2 - 2u$, which implies that x is a multiple of $2^{-p+1} = 2u$. The number $\frac{1}{x}$ is between $\frac{1}{2-2u} = \frac{1}{2} + \frac{u}{2} + \frac{u^2}{2} + \cdots$ and 1, and so t is between $\frac{1}{2}$ and 1, so t is a multiple of $2^{-p} = u$. From

$$\frac{1-u}{x} \leq t \leq \frac{1+u}{x}$$

we deduce

$$-u \leq 1 - xt \leq u.$$

Hence, $1 - xt$ is a multiple of 2^{-2p+1} of absolute value less than or equal to 2^{-p}, which implies that it is an exact floating-point number. □

The improved algorithm, based on the above property, is the one given in Algorithm 20 and Figure 2.9, with the corresponding error bound in Theorem 2.3.7.

Algorithm 20 – DWDivDW3$(x_h, x_\ell, y_h, y_\ell)$.

1: $t_h \leftarrow \mathrm{RN}(1/y_h)$
2: $r_h \leftarrow \mathsf{fma}(-y_h, t_h, 1)$ //exact operation
3: $r_\ell \leftarrow \mathrm{RN}(-y_\ell \cdot t_h)$
4: $(e_h, e_\ell) \leftarrow \mathsf{Fast2Sum}(r_h, r_\ell)$
5: $(\delta_h, \delta_\ell) \leftarrow \mathsf{DWTimesFP3}(e_h, e_\ell, t_h)$ //approximation to $(e_h + e_\ell)t_h$ with relative error $\leq 2u^2$ using Alg. 12
6: $(m_h, m_\ell) \leftarrow \mathsf{DWPlusFP}(\delta_h, \delta_\ell, t_h)$ //approximation to $\delta_h + \delta_\ell + t_h$ with relative error $\leq 2u^2 + 5u^3$ using Alg. 7
7: $(z_h, z_\ell) \leftarrow \mathsf{DWTimesDW3}(x_h, x_\ell, m_h, m_\ell)$ //approximation to $(x_h + x_\ell)(m_h + m_\ell)$ with relative error $\leq 5u^2$ using Alg. 15
8: **return** (z_h, z_ℓ)

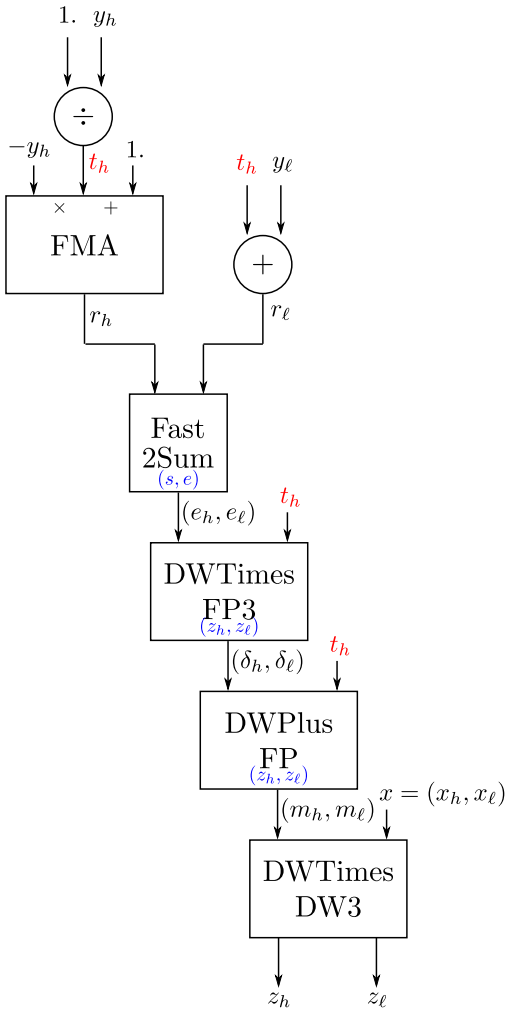


Figure 2.9 – Graphical representation of Algorithm 20. All algorithm calls output both parts of their results downwards.

Theorem 2.3.7. *If $p \geq 14$ and $y \neq 0$, the relative error of Algorithm 20 (DWDivDW3) is upper-bounded by $9.8u^2$.*

Proof. Roughly speaking, Algorithm 20 first approximates $\frac{1}{y}$ by $t_h = \mathrm{RN}(\frac{1}{y_h})$, then improves that approximation to $\frac{1}{y}$ by performing one step of Newton-Raphson iteration, and then multiplies the obtained approximation (m_h, m_ℓ) by x.

Without loss of generality, we assume $1 \le y_h \le 2 - 2u$, so that $\frac{1}{2} \le t_h \le 1$. We have

$$\left| t_h - \frac{1}{y_h} \right| \le \frac{u}{2},$$

and (from Property 2.3.6)

$$r_h = 1 - y_h t_h.$$

We also easily check that

$$\left(t_h(2 - y t_h) - \frac{1}{y} \right) = -y \left(t_h - \frac{1}{y} \right)^2. \tag{2.43}$$

Now, from $|y_\ell| \le u$ and $|t_h| \le 1$, we deduce $|y_\ell t_h| \le u$, so that $|r_\ell| \le u$, and

$$|r_\ell + y_\ell t_h| \le \frac{u^2}{2}.$$

This gives

$$e_h + e_\ell = r_h + r_\ell = 1 - y_h t_h - y_\ell t_h + \eta, \text{ with } |\eta| \le \frac{u^2}{2}. \tag{2.44}$$

Also, since $|y_h t_h - 1| = y_h |t_h - \frac{1}{y_h}| \le u$, we have $|r_h| \le u$, hence $|r_h + r_\ell| \le 2u$. This implies $|e_h| \le 2u$ and $|e_\ell| \le u^2$. Define $e = e_h + e_\ell = r_h + r_\ell$, we have $|e| \le 2u$.

Now, from Theorem 2.2.5, we have

$$\delta_h + \delta_\ell = e t_h (1 + \omega_1), \text{ with } |\omega_1| \le 2u^2, \tag{2.45}$$

and from Theorem 2.1.1, we have

$$m_h + m_\ell = (t_h + \delta_h + \delta_\ell)(1 + \omega_2), \text{ with } |\omega_2| \le 2u^2 + 5u^3. \tag{2.46}$$

Combining (2.45) and (2.46), we obtain

$$\begin{aligned} m_h + m_\ell &= (t_h + e t_h (1 + \omega_1))(1 + \omega_2) \\ &= t_h + e t_h + e t_h \omega_1 + \omega_2 t_h + \omega_2 e t_h + \omega_2 \omega_1 e t_h \\ &= t_h + e t_h + \alpha t_h, \end{aligned} \tag{2.47}$$

with

$$\begin{aligned} |\alpha| &= |e\omega_1 + \omega_2 + \omega_2 e + \omega_2 \omega_1 e| \\ &\le (2u)(2u^2) + (2u^2 + 5u^3) + (2u^2 + 5u^3)(2u) + (2u^2 + 5u^3)(2u^2)(2u) \\ &= 2u^2 + 13u^3 + 10u^4 + 8u^5 + 20u^6 \\ &\le 2u^2 + 14u^3 \text{ as soon as } p \ge 4. \end{aligned} \tag{2.48}$$

Therefore,

$$\begin{aligned} m_h + m_\ell &= t_h + e t_h + \alpha t_h \\ &= t_h + t_h(1 - y t_h + \eta) + \alpha t_h \\ &= t_h(2 - y t_h) + t_h(\eta + \alpha), \end{aligned}$$

which implies

$$\left| (m_h + m_\ell) - \frac{1}{y} \right| = \left| t_h(2 - y t_h) - \frac{1}{y} + t_h(\eta + \alpha) \right|,$$

so that, using (2.43) and the bounds on η and α,

$$\left|(m_h + m_\ell) - \frac{1}{y}\right| \leq y\left(t_h - \frac{1}{y}\right)^2 + t_h\left|\frac{5}{2}u^2 + 14u^3\right|. \tag{2.49}$$

Let us now consider

$$y^2\left(t_h - \frac{1}{y}\right).$$

That term is less than

$$y^2\left(\left(t_h - \frac{1}{y_h}\right) + \frac{y - y_h}{yy_h}\right)^2,$$

which is less than

$$y^2u^2\left(\frac{1}{2} + \frac{1}{y(y-u)}\right)^2.$$

The largest value of

$$y^2\left(\frac{1}{2} + \frac{1}{y(y-u)}\right)^2$$

for $1 \leq y < 2$ is always attained for $y = 1$, so that as soon as $p \geq 6$ (i.e., $u \leq \frac{1}{64}$), we have

$$y^2\left(t_h - \frac{1}{y}\right) \leq \left(\frac{1}{2} + \frac{1}{1 - \frac{1}{64}}\right)^2 u^2 = \frac{36481}{15876}u^2 \leq 2.298u^2.$$

Hence, from (2.49), we obtain

$$\left|(m_h + m_\ell) - \frac{1}{y}\right| \leq \frac{1}{y}2.298u^2 + t_h\left(\frac{5}{2}u^2 + 14u^3\right),$$

which implies

$$\left|x(m_h + m_\ell) - \frac{x}{y}\right| \leq \frac{x}{y}2.298u^2 + xt_h\left(\frac{5}{2}u^2 + 14u^3\right).$$

Notice that $|t_h| \leq \frac{1+u}{y_h} \leq \frac{(1+u)^2}{y}$, so that

$$\left|x(m_h + m_\ell) - \frac{x}{y}\right| \leq \frac{x}{y}\varphi(u), \tag{2.50}$$

with $\varphi(u) = 2.298u^2 + (1+u)^2\left(\frac{5}{2}u^2 + 14u^3\right)$. Now, from Theorem 2.2.5, we have

$$\begin{aligned} |z_h + z_\ell - x(m_h + m_\ell)| &\leq 5u^2|x(m_h + m_\ell)| \\ &\leq 5u^2\tfrac{x}{y} + 5u^2\left|\tfrac{x}{y} - x(m_h + m_\ell)\right| \\ &\leq \tfrac{x}{y}\left(5u^2 + 5u^2\varphi(u)\right). \end{aligned} \tag{2.51}$$

Combining (2.50) and (2.51) we finally obtain

$$\begin{aligned} \left|z_h + z_\ell - \tfrac{x}{y}\right| &\leq \tfrac{x}{y}(5u^2 + \varphi(u) + 5u^2\varphi(u)) \\ &\leq \tfrac{x}{y}\left(9.798u^2 + 19u^3 + 54.49u^4 + 109u^5 + 152.5u^6 + 70u^7\right) \\ &\leq 9.8u^2\tfrac{x}{y} \text{ as soon as } p \geq 14. \end{aligned}$$

□

This relative error bound is certainly a large overestimate, since we cumulate in its calculation the overestimates of the errors of Algorithms 12, 7, and 15. The largest error that we were able to obtain in practice is the one presented in Example 2.3.8.

Example 2.3.8. *If we have as input for Algorithm 20, with binary64 as underlying arithmetic, the values:*

$$\begin{aligned} x_h &= 4528288502329187, \\ x_\ell &= 1125391118633487/2^{51}, \\ y_h &= 4522593432466394, \text{ and} \\ y_\ell &= -9006008290016505/2^{54}, \end{aligned}$$

then the relative error of the algorithm is

$$5.922\ldots \times 2^{-106}.$$

2.4 Comparison and discussion

In this chapter we have proven relative error bounds for several basic building blocks of double-word arithmetic, suggested two new algorithms for multiplying two double-word numbers, suggested an improvement of the algorithms used in the QD library for dividing a double-word number by a floating-point number, and for dividing two double-word numbers. We have also suggested a new algorithm for dividing two double-word numbers when an FMA instruction is available.

Table 2.2 summarizes the obtained results. For the functions for which an error bound was already published, we always obtain a significantly smaller bound, except in one case, for which the previously known bound turned out to be slightly incorrect. Our results make it possible to have more trust in double-word arithmetic. They also allow us to give some recommendations:

- For adding two double-word numbers, one should *never* use Algorithm 8, unless one is certain that both operands have the same sign. Double-word numbers can be added very accurately using the (unfortunately more expensive) Algorithm 9.
- For multiplying a double-word number by a floating-point number, Algorithm 10 is the most accurate, while Algorithm 11 is slightly less accurate, yet slightly faster. Hence one cannot say that one is really better than the other one. The choice between them depends on whether one mainly needs speed or accuracy. If an FMA instruction is available, Algorithm 12 is a good candidate.
- For multiplying two double-word numbers, if an FMA instruction is available, Algorithm 15 is to be favoured. It is more accurate both from a theoretical (better error bound) and from a practical (smaller observed errors in our intensive testings) point of view.
- There is no point in using Algorithm 16 for dividing a double-word number by a floating-point number. Algorithm 17, introduced here, always returns the same result and it is faster.
- There is no point in using Algorithm 18 for dividing two double-word numbers. Algorithm 19, presented in this paper, always returns the same result and it is faster. If an FMA instruction is available, depending whether the priority is speed or accuracy, one might prefer Algorithm 20. It is almost certainly significantly more accurate (although we have no full proof of that: we can just say that our bounds are smaller, as well as the observed errors), however, it is slower.

Table 2.2 – Summary of the results presented in this chapter. For each algorithm, we give the previously known bound (when we are aware of it, and when the algorithm already existed), the bound we have proved, and the largest relative error observed in our fairly intensive tests.

Operation	Algorithm	Previously known bound	Our bound	Largest relative error observed in experiments
DW + FP	Algorithm 7	?	$2u^2 + 5u^3$	$2u^2 - 6u^3$
DW + DW	Algorithm 8	N/A	N/A	1
	Algorithm 9	$2u^2$ (incorrect)	$3u^2 + 13u^3$	$2.25u^2$
DW × FP	Algorithm 10	$4u^2$	$1.5u^2 + 4u^3$	$1.5u^2$
	Algorithm 11	?	$3u^2$	$2.517u^2$
	Algorithm 12	N/A	$2u^2$	$1.984u^2$
DW × DW	Algorithm 13	$11u^2$	$7u^2$	$4.9916u^2$
	Algorithm 14	N/A	$6u^2$	$4.9433u^2$
	Algorithm 15	N/A	$5u^2$	$3.936u^2$
DW ÷ FP	Algorithm 16	$4u^2$	$3.5u^2$	$2.95u^2$
	Algorithm 17	N/A	$3.5u^2$	$2.95u^2$
DW ÷ DW	Algorithm 18	?	$15u^2 + 56u^3$	$8.465u^2$
	Algorithm 19	N/A	$15u^2 + 56u^3$	$8.465u^2$
	Algorithm 20	N/A	$9.8u^2$	$5.922u^2$

In Table 2.3 we present an overview of the performance obtained using the algorithms presented in this chapter when run both on CPU and on GPU. [2] The values are given in Mop/s (Mega operations per second) and were obtained on the architectures described in Section 1.4.1. We also include the floating-point operation count.

In the below table you can see that most of the algorithms perform as expected, in accordance with their floating-point operations count. However, this is not the case of Algorithm 20 for which we obtained an exceptional performance. After analyzing the situation, we concluded that this is due to compiler optimizations at instruction level parallelism and also, the capacity of the algorithm to fill up the pipeline. This does not mean that we always recommend the use of this algorithm for dividing two double-word numbers; in a real world application the performance will depend heavily on the type of the application.

2. The values reflect the performance obtained on GPU using only one execution thread.

Table 2.3 – Performance in Mop/s for double-word addition, multiplication and division algorithms. For each algorithm, we give the values obtained when run on CPU and on GPU, and the number of floating-point operations required by the algorithm

Operation	Algorithm	Performance on CPU (Mop/s)	Performance on GPU (Mop/s)	♯ of Flops
DW + FP	Algorithm 7	229.866	5.379	10
DW + DW	Algorithm 8	208.961	5.379	11
	Algorithm 9	114.932	3.81	20
DW × FP	Algorithm 10	342.065	6.532	10
	Algorithm 11	484.239	9.145	7
	Algorithm 12	484.248	9.626	6
DW × DW	Algorithm 13	392.666	8.069	9
	Algorithm 14	473.746	8.703	8
	Algorithm 15	331.026	7.62	9
DW ÷ FP	Algorithm 16	80.7219	1.693	16
	Algorithm 17	81.6378	2.176	10
DW ÷ DW	Algorithm 18	80.615	1.784	24
	Algorithm 19	81.5674	1.988	18
	Algorithm 20	329.962	5.964	31

Chapter 3

Floating-Point Expansions Arithmetic

In this chapter we deal with the so-called *floating-point expansions*, i.e., the representation of real numbers as the unevaluated sum of several (more than two) standard machine precision floating-point numbers. Firstly we present two new (re-)normalization algorithms that are meant to ensure that the expansions satisfy their formal definition; these will be the base of all other arithmetic algorithms. We develop algorithms for all basic operations (addition/subtraction, multiplication, reciprocal/division and square root) and we provide them with correctness and error bound proofs.

This work was partially published in:
– *On the computation of the reciprocal of floating point expansions using an adapted Newton-Raphson iteration* [42], joint work with M. Joldes, and J.-M. Muller, published in Proceedings of the 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2014).
– The extended journal version *Arithmetic algorithms for extended precision using floating-point expansions* [41], joint work with M. Joldes, O. Marty, [1] and J.-M. Muller, published in the IEEE Transactions on Computers journal.
– *A New Multiplication Algorithm for Extended Precision Using Floating-Point Expansions* [69], joint work with J.-M. Muller and P. Tang, [2] published in Proceedings of the 23rd IEEE Symposium on Computer Arithmetic (ARITH 2016).

In the previous chapter we just doubled the available precision, but now we want to achieve arbitrary precision (or rather *moderately arbitrary* precision: for precisions of thousands of digits other solutions are preferred). A natural extension of the notion of double-word is the notion of *floating-point expansion*. If, starting from a set of floating-point inputs, we only perform exact additions, subtractions, or multiplications, then the values we obtain are always equal to finite sums of floating-point numbers. Such finite sums are called expansions and they are formally defined in Definition 3.0.1.

Definition 3.0.1. *A floating-point expansion x with n terms is the unevaluated sum of n floating-point numbers $x_0, \ldots, x_{n-1}$, in which all nonzero terms are ordered by magnitude (i.e., if y is the sequence obtained by removing all zeros in the sequence x, and if sequence y contains m terms, $|y_i| \geq |y_{i+1}|$, for all $0 \leq i < m-1$). Each x_i is called a component (or a term) of x.*

1. Student intern during collaboration, now engineer at Google, France.
2. Senior engineer at Intel Corporation.

A natural idea is to try to manipulate such expansions for performing calculations that are either exact, either approximate yet very accurate. The arithmetic on floating-point expansions was first developed by Priest [81], and in a slightly different way by Shewchuk [88].

One may notice that the notion of expansion is "redundant" since a nonzero number always has more than one representation as a floating-point expansion. To make the concept useful in practice and easy to manipulate, we must introduce a constraint on the components: the x_i's cannot "overlap". The notion of (non)overlapping varies depending on the authors. We present here three definitions that were already present in the literature (Definition 3.0.3 to 3.0.5) and we introduce a new one (Definition 3.0.6) that allows for a relatively relaxed handling of the floating-point expansions and keeps the redundancy to a minimum. An expansion may contain interleaving zeros, but the definitions that follow apply only to the non-zero terms of the expansion (i.e., the array y in Definition 3.0.1).

Definition 3.0.2. *($\mathcal{P}$-nonoverlapping floating-point numbers) 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 [82]) if $|e_y - e_x| \geq p$.*

Definition 3.0.3. *A floating-point expansion $x_0, x_1, \ldots, x_{n-1}$ is $\mathcal{P}$-nonoverlapping (that is, nonoverlapping according to Priest's definition [82]) if all of its components are mutually $\mathcal{P}$-nonoverlapping (i.e., $|x_i| < \mathrm{ulp}(x_{i-1})$, for all $0 < i < n$).*

Shewchuk [88] weakens this into *nonzero-overlapping* sequences, as it follows in Definition 3.0.4.

Definition 3.0.4. *A floating-point expansion $x_0, x_1, \ldots, x_{n-1}$ is $\mathcal{S}$-nonoverlapping (that is, nonoverlapping according to Shewchuk's definition [88]) if for all $0 < i < n$, we have $e_{x_{i-1}} - e_{x_i} \geq p - z_{x_{i-1}}$, where $e_{x_{i-1}}$ and e_{x_i} are the exponents of x_{i-1} and x_i, respectively, and $z_{x_{i-1}}$ is the number of trailing zeros of x_{i-1}. This can be translated into $|x_i| < \mathrm{uls}(x_{i-1})$, for all $0 < i < n$.*

In general, a $\mathcal{P}$-nonoverlapping expansion carries more information than a $\mathcal{S}$-nonoverlapping one with the same number of components. In the worst case, in radix 2, a $\mathcal{S}$-nonoverlapping expansion with 53 components may not contain more information than one binary64 floating-point number; it suffices to put one bit of information into every component.

When Priest first started developing the floating-point expansion arithmetic, he considered that all the computations were done in faithful floating-point arithmetic (see [82]), since round-to-nearest rounding mode was not so common. More recently, a slightly stronger sense of nonoverlapping was introduced by Hida, Li and Bailey [34] (see Definition 3.0.5).

Definition 3.0.5. *A floating-point expansion $x_0, x_1, \ldots, x_{n-1}$ is $\mathcal{B}$-nonoverlapping (that is, nonoverlapping according to Bailey's definition [34]) if for all $0 < i < n$, we have $|x_i| \leq \frac{1}{2}\,\mathrm{ulp}(x_{i-1})$.*

A visual representation of the above definitions, inspired from [79], in order of their "strength" (from (a) the strongest to (c) the weakest) is given in Figure 3.1.

Intuitively, the stronger the sense of the (non)overlapping definition, the more difficult it is to guarantee it in the output. In practice, even the $\mathcal{P}$-nonoverlapping property proved to be quite difficult and costly to obtain. In the same time, the $\mathcal{S}$-nonoverlapping property is not strong enough. On account of that we chose to compromise by using a different sense of (non)overlapping, referred to as ulp-nonoverlapping , that we formally introduce in Definition 3.0.6.

Definition 3.0.6. *A floating-point expansion $x_0, x_1, \ldots, x_{n-1}$ is* ulp*-nonoverlapping if for all $0 < i < n$, $|x_i| \leq \mathrm{ulp}(x_{i-1})$.*

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. This is showed in Figure 3.2.

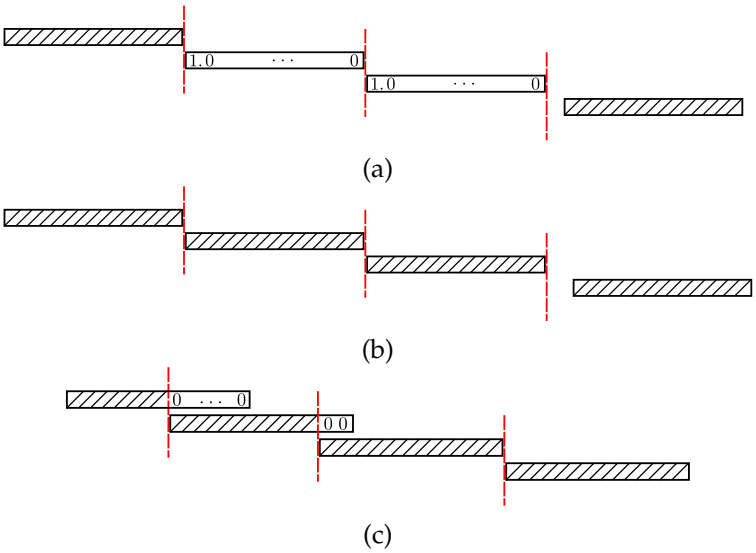


Figure 3.1 – Graphical representation of nonoverlapping sequences by (a) Bailey's scheme, (b) Priest's scheme, and (c) Shewchuk's scheme [79].

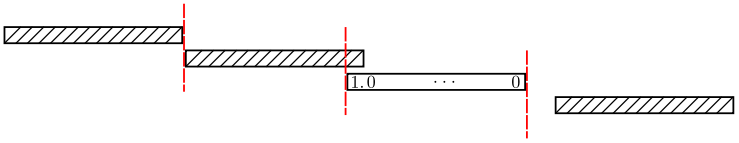


Figure 3.2 – Graphical representation of an ulp-nonoverlapping sequence by Definition 3.0.6.

Remark 3.0.7. *Note that for $\mathcal{P}$-nonoverlapping expansions we have*

$$|x_i| \leq \frac{2^p - 1}{2^p} \operatorname{ulp}(x_{i-1})$$

and for $\mathcal{S}$-nonoverlapping expansions

$$|x_i| \leq \frac{2^p - 1}{2^p} \operatorname{uls}(x_{i-1}).$$

Depending on the nonoverlapping type of an expansion, when using standard floating-point formats as underlying arithmetic, 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 $\mathcal{B}$-nonoverlapping , $\mathcal{P}$-nonoverlapping or ulp-nonoverlapping expansions, for the two most common floating-point formats, the constraints are:

- for binary64 (exponent range $[-1022, 1023]$) the maximum expansion size is 39;
- for binary32 (exponent range $[-126, 127]$) the maximum expansion size is 12.

In this thesis we will manipulate only ulp-nonoverlapping expansions. The only previously existing algorithms for manipulating arbitrary precision floating-point expansions that come with rigorous correctness proofs are the ones given by Priest in [81], so we use them for comparison.

The outline of the chapter is structured as follows: in Section 3.1 we introduce two classical algorithms that we make use of in different contexts. We then proceed to introducing algorithms for basic operations using floating-point expansions: renormalization (Section 3.2), addition (Section 3.3), multiplication (Section 3.4), reciprocal/division (Section 3.5) and square root (Section 3.6). We end the chapter by comparing the algorithms in Section 3.7.

3.1 Prerequisites

Before going further into the algorithms for performing arithmetic operations using floating-point expansions we need to introduce two classical algorithms of which we are going to make extensive use. We employ these algorithms in different contexts, considering different input/output constraints, so, for clarity, we present all their properties here, providing proofs.

Overlapping numbers. We need to formally define the concept of floating-point numbers that overlap by at most d digits. This is done in Definition 3.1.1, from where we can deduce and prove Property 3.1.2.

Definition 3.1.1. *Consider an array of floating-point numbers: $x_0, x_1, \ldots, x_{n-1}$. According to Priest's [81] definition, they overlap by at most d digits ($0 \leq d < p$) if and only if $\forall i, 0 \leq i \leq n-2, \exists k_i, \delta_i$ such that:*

$$2^{k_i} \leq |x_i| < 2^{k_i+1}, \tag{3.1}$$
$$2^{k_i-\delta_i} \leq |x_{i+1}| \leq 2^{k_i-\delta_i+1}, \tag{3.2}$$
$$\delta_i \geq p - d, \tag{3.3}$$
$$\delta_i + \delta_{i+1} \geq p - z_{i-1}, \tag{3.4}$$

where z_{i-1} is the number of trailing zeros at the end of x_{i-1} and for $i = 0$, $z_{-1} := 0$.

Loosely speaking, if the terms were to be written in positional notation, the digits of any two successive non-zero terms would coincide in at most d digit positions, and no three terms would mutually coincide in any digit position.

Property 3.1.2. *Let $x_0, x_1, \ldots, x_{n-1}$ be an array of floating-point numbers that overlap by at most d digits ($0 \leq d < p$). The following properties hold:*

$$|x_{i+1}| < 2^d \operatorname{ulp}(x_i), \tag{3.5}$$
$$\operatorname{ulp}(x_{i+1}) \leq 2^{d-p} \operatorname{ulp}(x_i), \tag{3.6}$$
$$|x_{i+2} + x_{i+1}| \leq (2^d + 2^{2d-p}) \operatorname{ulp}(x_i). \tag{3.7}$$

Proof. We have $\operatorname{ulp}(x_i) = 2^{k_i-p+1}$ and from (3.3) we get $|x_{i+1}| < 2^{k_i-\delta_i+1} < 2^{p-\delta_i} \operatorname{ulp}(x_i) < 2^d \operatorname{ulp}(x_i)$. This proves that (3.5) holds for all $0 \leq i < n-1$.

By applying (3.3) we get $\operatorname{ulp}(x_{i+1}) = 2^{k_i-\delta_i-p+1} \leq 2^{d-p} \operatorname{ulp}(x_i)$, which proves that (3.6) holds for all $0 \leq i < n-1$.

We have $|x_{i+1}| \leq 2^d \operatorname{ulp}(x_i)$ and $|x_{i+2}| \leq 2^d \operatorname{ulp}(x_{i+1}) \leq 2^{2d-p} \operatorname{ulp}(x_i)$ from which (3.7) follows. □

3.1.1 The VecSum algorithm

It first appeared as part of Priest's renormalization algorithm [81], but was coined as VecSum by Ogita et.al [88, 79]. The VecSum algorithm, presented in Algorithm 21 and Figure 3.3, is simply a chain of 2Sum that performs an error-free transform on n floating-point numbers. Kahan calls this a "distillation" algorithm.

This algorithm is of interest because it has many nice properties and, depending on the input constraints, it allows us to deduce some relationships between the outputted numbers. One can observe the following:

- $x_0 + \cdots + x_{n-1} = e_0 + \cdots + e_{n-1}$, i.e., the sums of the input and output arrays are equal;

Algorithm 21 – VecSum$(x_0, \ldots, x_{n-1})$.

1: $s_{n-1} \leftarrow x_{n-1}$
2: **for** $i \leftarrow n-2$ **to** 0 **do**
3: $\quad (s_i, e_{i+1}) \leftarrow \text{2Sum}(x_i, s_{i+1})$
4: **end for**
5: $e_0 \leftarrow s_0$
6: **return** $e_0, \ldots, e_{n-1}$

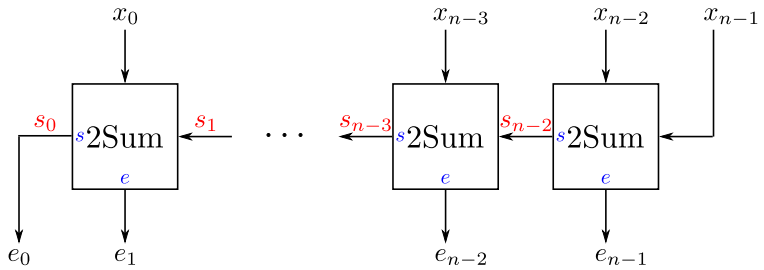


Figure 3.3 – Graphical representation of Algorithm 21. In the 2Sum calls the sum s is outputted to the left and the error e downwards.

- $e_0 = \text{RN}(x_0 + \text{RN}(x_1 + \text{RN}(\cdots + \text{RN}(x_{n-2} + x_{n-1}))))$, i.e., $e_0 = s_0$ is the result of the "naive" summation of $x_0, \ldots, x_{n-1}$.

Other properties, that we are going to use in Section 3.2, are given in Theorems 3.1.3, 3.1.7, and 3.1.9. Each theorem considers a different input for the algorithm and shows the constrains on the output. In the proofs we use the notations showed in Figure 3.3.

Theorem 3.1.3 was first proven by Priest [81], who showed that the algorithm can transform an array of overlapping numbers into an $\mathcal{S}$-nonoverlapping expansion. We are going to use this theorem in Section 3.2.

Theorem 3.1.3. *Let $x = (x_0, x_1, \ldots, x_{n-1})$ be an array of floating-point numbers that overlap by at most d digits ($d \leq p-2$) and that may contain interleaving* 0*s. Provided that no underflow/overflow occurs during computations, and that $\frac{2^d}{1-2^{d-p}}(1 + (n-2)2^{-p}) \leq 2^{p-1}$ (which always holds in practice), when applying Algorithm 21 on x, the output array $e = (e_0, e_1, \ldots, e_{n-1})$ is an $\mathcal{S}$-nonoverlapping expansion that may contain interleaving zeros.*

Proof. We can quickly observe that if $x_i = 0$, then the sum $s_i = s_{i-1}$ and $e_{i-1} = 0$. This implies that in the result we can also have interleaving zeros.

Now, since $s_i = \text{RN}(x_i + s_{i+1})$, s_i is closer to $x_i + s_{i+1}$ than x_i. This gives us $|(x_i + s_{i+1}) - s_i| \leq |(x_i + s_{i+1}) - x_i|$, and so $|e_{i+1}| \leq |s_{i+1}|$. Similarly, s_i is closer to $x_i + s_{i+1}$ than s_{i+1}, so $|e_{i+1}| \leq |x_i|$. From (3.5) we get:

$$\begin{aligned} |x_{i+1}| \quad &+ \quad |x_{i+2}| + \cdots \leq \\ &\leq \quad [2^d + 2^{2d-p} + 2^{3d-2p} + 2^{4d-3p} + \cdots]\,\text{ulp}(x_i) \\ &\leq \quad \frac{2^d}{1-2^{d-p}}\,\text{ulp}(x_i). \end{aligned} \tag{3.8}$$

We know that $s_{i+1} = \text{RN}(x_{i+1} + \text{RN}(\cdots + x_{n-1}))$ and, using a property given by Jeannerod and Rump in [40], we get:

$$\begin{aligned} |s_{i+1} \quad &- \quad (x_{i+1} + \cdots + x_{n-1})| \\ &\leq \quad (n-i-2)\cdot 2^{-p}\cdot(|x_{i+1}| + \cdots + |x_{n-1}|)\,. \end{aligned} \tag{3.9}$$

From (3.8) and (3.9) we obtain:

$$|s_{i+1}| \leq \frac{2^d}{1-2^{d-p}}(1+(n-i-2)2^{-p})\,\mathrm{ulp}(x_i).$$

It is easily seen that

$$\frac{2^d}{1-2^{d-p}}(1+(n-i-2)2^{-p}) \leq 2^{p-1}, \tag{3.10}$$

is satisfied for $p \geq 4$ and $n \leq 10$, for $p \geq 5$ and $n \leq 18$, for $p \geq 6$ and $n \leq 34$, and so on. This includes all practical cases, when $d \leq p-2$, so that $\mathrm{ulp}(s_{i+1}) < \mathrm{ulp}(x_i)$. Therefore x_i and s_{i+1} are multiples of $\mathrm{ulp}(s_{i+1})$, thus $x_i + s_{i+1}$ is multiple of $\mathrm{ulp}(s_{i+1})$, hence $\mathrm{RN}(x_i + s_{i+1})$ is multiple of $\mathrm{ulp}(s_{i+1})$ and $|e_{i+1}| = |x_i + s_{i+1} - \mathrm{RN}(x_i + s_{i+1})|$ is multiple of $\mathrm{ulp}(s_{i+1})$.

Also, by definition of 2Sum, we have $|e_{i+2}| \leq \frac{1}{2}\,\mathrm{ulp}(s_{i+1})$. Now, we can compare $|e_{i+1}|$ and $|e_{i+2}|$. Since $|e_{i+1}|$ is a multiple of $\mathrm{ulp}(s_{i+1})$, either $e_{i+1} = 0$ or e_{i+1} is larger than $2\,|e_{i+2}|$ and multiple of 2^k, such that $2^k > |e_{i+2}|$. This implies that the array $e = (e_0, e_1, \ldots, e_{n-1})$ is $\mathcal{S}$-nonoverlapping and may have interleaving zeros. □

Remark 3.1.4. *We observe that the calls to 2Sum can be replaced by calls to Fast2Sum. This is possible because we have* $|s_{i+1}| \leq 2^{p-1}\,\mathrm{ulp}(x_i)$, *for* $p \geq 4$ *and* $n \leq 10$, *for* $p \geq 5$ *and* $n \leq 18$, *for* $p \geq 6$ *and* $n \leq 34$, *and so on. Also* $\mathrm{ulp}(x_i) \leq 2^{-p+1}\,|x_i|$. *As a deduction* $|s_{i+1}| \leq |x_i|$.

Remark 3.1.5. *Theorem 3.1.3 also holds if we have the weaker condition* $d \leq p-1$, *provided that* $n \leq 12$ *for binary32 and* $n \leq 39$ *for binary64. However, if this is the case, the above remark does not hold anymore, so we cannot replace the 2Sum algorithm with Fast2Sum.*

In Example 3.1.6, using a toy system with $p = 5$, we present the behavior of the algorithm in this setting.

Example 3.1.6. *Consider a floating-point system with precision* $p = 5$. *Let* x *be a* 5*-term array:*

$$\begin{aligned} x_0 &= -1.0011_2 \times 2^{-1}, \\ x_1 &= -2^{-3}, \\ x_2 &= 1.1110_2 \times 2^{-6}, \\ x_3 &= -1.1010_2 \times 2^{-8}, \\ x_4 &= 1.1110_2 \times 2^{-11}. \end{aligned}$$

Using x *as input for Algorithm 21 will result in the* $\mathcal{S}$*-nonoverlapping array* e*:*

$$\begin{aligned} e_0 &= -1.0110_2 \times 2^{-1}, \\ e_1 &= -2^{-7}, \\ e_2 &= 0.0, \\ e_3 &= -2^{-11}, \\ e_4 &= -2^{-13}. \end{aligned}$$

The second setting that we consider is going to be used in Section 3.2 and 3.3 later on.

Theorem 3.1.7. *Let* $x = (x_0, x_1, \ldots, x_{n-1})$ *be an array of floating-point numbers that overlap by at most* $d \leq p-2$ *digits and may contain pairs of at most* 2 *consecutive terms that overlap by* p *digits. The array may also contain interleaving* 0*s. Provided that no underflow/overflow occurs during computations, when applying Algorithm 21 on* x, *the output array* $e = (e_0, e_1, \ldots, e_{n-1})$ *satisfies* $|e_0| > |e_1| \geq \cdots > |e_{i-1}| \geq |e_i| > |e_{i+1}| \geq |e_{i+2}| > \cdots$ *and the* e_i*s are* $\mathcal{S}$*-nonoverlapping for strict inequality or they are equal to a power of* 2.

Proof. The proof is done by induction, and for an easier following we include Figure 3.4 that illustrates an excerpt of the algorithm.

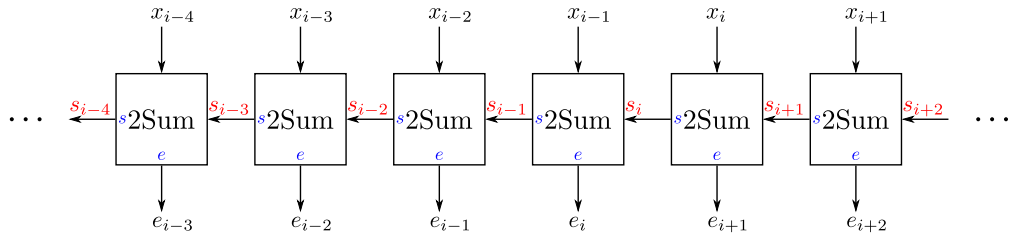


Figure 3.4 – Excerpt of Algorithm 21.

Step 1. We consider that the input array contains only one pair of p-overlapping numbers, say x_i and x_{i+1}.

The same as in the proof of Theorem 3.1.3, we have $\mathrm{ulp}(s_{i+2}) < \mathrm{ulp}(x_{i+1})$. Since $s_{i+1} = \mathrm{RN}(s_{i+2} + x_{i+1})$ we can deduce that $\mathrm{ulp}(s_{i+1}) \leq 2\,\mathrm{ulp}(x_{i+1}) = 2\,\mathrm{ulp}(x_i)$. We tackle this by splitting it into two cases.

(i) $\mathrm{ulp}(s_{i+1}) = 2\,\mathrm{ulp}(x_{i+1}) = 2\,\mathrm{ulp}(x_i)$. It follows that $s_{i+1} > x_i$, but we can make the following statement: $(e_0, \ldots, e_{i+1}) = \mathrm{VecSum}(x_0, \ldots, x_{i-1}, s_{i+1}, x_i)$, and consider them swapped, since $\mathrm{2Sum}(x_i, s_{i+1})$ is the same as $\mathrm{2Sum}(s_{i+1}, x_i)$. Also, $(s_{i+1}, e_{i+2}, \ldots, e_{n-1}) = \mathrm{VecSum}(x_{i+1}, \ldots, x_{n-1})$. From Theorem 3.1.3 and Remark 3.1.5 it follows that the two portions of the array e are $\mathcal{S}$-nonoverlapping . Now we need to find a relationship between e_{i+1} and e_{i+2}.

From 2Sum, $|e_{i+2}| \leq \frac{1}{2}\,\mathrm{ulp}(s_{i+1}) = \mathrm{ulp}(x_i)$. We have $s_i + e_{i+1} = x_i + s_{i+1}$, so s_i and e_{i+1} are both multiples of $\mathrm{ulp}(x_i)$, hence $|e_{i+1}| \geq \mathrm{ulp}(x_i)$. It holds that:

$$|e_{i+2}| \leq \mathrm{ulp}(x_i) \leq |e_{i+1}|\,. \tag{3.11}$$

If $|e_{i+2}| < \mathrm{ulp}(x_i)$, the two numbers are $\mathcal{S}$-nonoverlapping and it follows that the entire array e is $\mathcal{S}$-nonoverlapping . Otherwise $|e_{i+2}| = \mathrm{ulp}(x_i) = |e_{i+1}|$, meaning that we have one pair of terms that are equal to a power of 2 and the rest of the array e is *$\mathcal{S}$-nonoverlapping*. For this case the theorem holds.

(ii) $\mathrm{ulp}(s_{i+1}) \leq \mathrm{ulp}(x_{i+1}) = \mathrm{ulp}(x_i)$.
We have $s_i = \mathrm{RN}(x_i + s_{i+1})$, so it follows that $\mathrm{ulp}(s_i) \leq 2\,\mathrm{ulp}(x_i)$. Also, from the hypothesis we know that $\mathrm{ulp}(x_i) \leq \frac{1}{4}\,\mathrm{ulp}(x_{i-1})$. This implies $\mathrm{ulp}(s_i) \leq \frac{1}{2}\,\mathrm{ulp}(x_{i-1})$, meaning that s_i and x_{i-1} overlap by at most $p-1$ digits.

We know that $(e_0, \ldots, e_i) = \mathrm{VecSum}(x_0, \ldots, x_{i-1}, s_i)$ and $(s_{i+1}, e_{i+2}, \ldots, e_{n-1}) = \mathrm{VecSum}(x_{i+1}, \ldots, x_{n-1})$, and from Theorem 3.1.3 and Remark 3.1.5 it follows that they are $\mathcal{S}$-nonoverlapping . Now we need to find a relationship between e_i, e_{i+1} and e_{i+2}.

From 2Sum, $|e_{i+1}| \leq \frac{1}{2}\,\mathrm{ulp}(s_i)$. Also, $s_{i-1} + e_i = x_{i-1} + s_i$, so e_i is multiple of $\mathrm{ulp}(s_i)$, hence e_i and e_{i+1} are $\mathcal{S}$-nonoverlapping .

Following the same reasoning $|e_{i+2}| \leq \frac{1}{2}\,\mathrm{ulp}(s_{i+1})$ and e_{i+1} is multiple of $\mathrm{ulp}(s_{i+1})$, hence e_{i+1} and e_{i+2} are $\mathcal{S}$-nonoverlapping . This makes the entire array e $\mathcal{S}$-nonoverlapping , proving that the theorem also holds for this case.

Step 2. We consider at most 2 consecutive pairs of p-overlapping numbers, say x_{i-2} overlaps with x_{i-1} and x_i overlaps with x_{i+1}.

Same as above we have $(e_0, \ldots, e_{i-1}) = \mathrm{VecSum}(x_0, \ldots, x_{i-3}, s_{i-1}, x_{i-2})$ and $(e_{i+2}, \ldots, e_{n-1}) = \mathrm{VecSum}(x_{i+1}, \ldots, x_{n-1})$, with both portions of the array e $\mathcal{S}$-nonoverlapping .

We know that $s_{i+1} = \mathrm{RN}(x_{i+1} + s_{i+2})$, so $\mathrm{ulp}(s_{i+1}) \leq 2\,\mathrm{ulp}(x_{i+1}) = 2\,\mathrm{ulp}(x_i)$. If they are equal we either have e_{i+1} $\mathcal{S}$-nonoverlapping with e_{i+2} or we have the limit case $|e_{i+1}| = |e_{i+2}| = \mathrm{ulp}(x_i) = \mathrm{ulp}(x_{i+1})$.

Also $s_i = \mathrm{RN}(x_i + s_{i+1})$, so $\mathrm{ulp}(s_i) \leq 2\,\mathrm{ulp}(s_{i+1}) \leq \mathrm{ulp}(x_{i-1})$. Since $s_{i-1} + e_i = x_{i-1} + s_i$, s_{i-1} and e_i are multiples of $\mathrm{ulp}(s_i)$. Also $|e_{i+1}| \leq \frac{1}{2}\,\mathrm{ulp}(s_i)$, from where we deduce that e_i and e_{i+1} are $\mathcal{S}$-nonoverlapping .

Now, $\mathrm{ulp}(s_{i-1}) \leq 2\,\mathrm{ulp}(x_{i-1}) = 2\,\mathrm{ulp}(x_{i-2})$. In case they are equal, the limit case $|e_{i-1}| = |e_i| = \mathrm{ulp}(x_{i-2}) = \mathrm{ulp}(x_{i-1})$ can occur again.

This proves that the array e is *$\mathcal{S}$-nonoverlapping* and may contain two consecutive pairs of powers of 2, so the theorem holds.

Step 3. As the last step of the induction consider N consecutive pairs of p-digit overlapping numbers, starting from x_i overlapping with x_{i+1} and so on.

For $(e_{i-1}, e_i, e_{i+1}, \ldots, e_{n-1})$ the same things hold as proven in the previous case.

Now x_{i-4} and x_{i-3} can overlap by at most p digits, hence $\mathrm{ulp}(s_{i-2}) \leq \mathrm{ulp}(x_{i-3}) = \mathrm{ulp}(x_{i-4})$. We know that $|e_{i-1}| \leq \frac{1}{2}\,\mathrm{ulp}(s_{i-2})$ and because $s_{i-3} + e_{i-2} = x_{i-3} + s_{i-2}$, s_{i-3} and e_{i-2} are multiples of $\mathrm{ulp}(s_{i-2})$. If follows that e_{i-2} and e_{i-1} are $\mathcal{S}$-nonoverlapping .

We have the case in which x_{i-4}, x_{i-3} and s_{i-2} overlap by at most p digits, so we get $\mathrm{ulp}(s_{i-3}) \leq \mathrm{ulp}(x_{i-4}) = \mathrm{ulp}(x_{i-3})$ and the limit case $|e_{i-3}| = |e_{i-2}| = \mathrm{ulp}(x_{i-4})$ can occur again.

We know that we can have only pairs of 2 floating-point numbers that overlap by p digits and that the other numbers in the input sequence overlap by at most $p - 2$ digits. This ensures that we can only get pairs of 3 floating-point numbers of the form (x_{j-1}, x_j, s_{j+1}) that overlap by p digits. In this case we will either have $e_j = e_{j+1}$ equal to a power of 2, either e_j and e_{j+1} are $\mathcal{S}$-nonoverlapping , as showed before.

By this the induction holds and we have at most pairs of two e_js that are equal to a power of 2 and the theorem is proven. □

In Example 3.1.8 we show the algorithm's behavior under these assumptions.

Example 3.1.8. *Consider a floating-point system with precision $p = 5$. Let x be a 5-term array:*

$$\begin{aligned} x_0 &= -1.0001_2 \times 2^{-3}, \\ x_1 &= -1.1111_2 \times 2^{-3}, \\ x_2 &= -1.0110_2 \times 2^{-6}, \\ x_3 &= -1.1000_2 \times 2^{-6}, \\ x_4 &= -1.0110_2 \times 2^{-9}. \end{aligned}$$

Using x as input for Algorithm 21 will result in the array e:

$$\begin{aligned} e_0 &= -1.1010_2 \times 2^{-2}, \\ e_1 &= 2^{-7}, \\ e_2 &= 2^{-7}, \\ e_3 &= 2^{-10}, \\ e_4 &= 2^{-12}. \end{aligned}$$

Shewchuk [88] used the same algorithm (under the name Grow-Expansion) for adding an $\mathcal{S}$-nonoverlapping expansion with a random floating-point number. Theorem 3.1.9 shows that when doing this, the resulted expansion is also $\mathcal{S}$-nonoverlapping .

Theorem 3.1.9. *Let $x = (x_0, x_1, \ldots, x_{n-1})$ be an $\mathcal{S}$-nonoverlapping floating-point expansion that may contain interleaving* 0*s and t a random floating-point number. Provided that no underflow/overflow occurs during computations, when applying Algorithm 21 on (x, t), the output array $e = (e_0, e_1, \ldots, e_n)$ is going to be $\mathcal{S}$-nonoverlapping .*

Proof. We know that s_{n-1} is the nearest floating-point number to $x_{n-1} + t$, closer than t. This means that $|s_{n-1} - (x_{n-1} + t)| \leq |(x_{n-1} + t) - t|$, hence $|e_n| \leq |x_{n-1}|$. There exists two integers k' and k'' such that:

$$\begin{aligned} &s_{n-1} \text{ is multiple of } 2^{k'} \text{ s.t. } |e_n| \leq \frac{1}{2}2^{k'}, \text{and} \\ &x_{n-2} \text{ is multiple of } 2^{k''} \text{ s.t. } |x_{n-1}| \leq \frac{1}{2}2^{k''}. \end{aligned} \tag{3.12}$$

We define $k = \min(k', k'')$ and we observe that e_{n-1} is multiple of 2^k. Since $|e_n|$ is less than both $2^{k'}$ and $2^{k''}$, we proved that e_{n-1} and e_n are $\mathcal{S}$-nonoverlapping .

Now, we can prove by induction that for all $n \geq i > 0$, $|e_i| \leq \frac{1}{2}\,\mathrm{uls}(e_{i-1})$ (i.e., e_i and e_{i-1} are $\mathcal{S}$-nonoverlapping). We proved that it holds for $i = n$. Following the same reasoning we can deduce that e_{i-1} is multiple of $2^{\hat{k}}$, where $\hat{k}$ is an integer, and $|e_i| \leq 2^{\hat{k}}$. This shows that e_{i-1} and e_i are $\mathcal{S}$-nonoverlapping and the induction holds. □

We give a numerical example in Example 3.1.10.

Example 3.1.10. *Consider a floating-point system with precision $p = 5$. Let x be a 5-term $\mathcal{S}$-nonoverlapping array:*

$$\begin{aligned} x_0 &= -1.1001_2 \times 2^{-2}, \\ x_1 &= 2^{-8}, \\ x_2 &= 0.0, \\ x_3 &= 2^{-12}, \\ x_4 &= 1.1011_2 \times 2^{-14}, \end{aligned}$$

and $t = -1.1010_2 \times 2^{-8}$ an arbitrary number;

Using (x, t) as input for Algorithm 21 will result in the $\mathcal{S}$-nonoverlapping array e:

$$\begin{aligned} e_0 &= -1.1001_2 \times 2^{-2}, \\ e_1 &= -1.001_2 \times 2^{-9}, \\ e_2 &= 0.0, \\ e_3 &= 0.0, \\ e_4 &= 0.0, \\ e_5 &= 1.1011_2 \times 2^{-14}. \end{aligned}$$

In the worst case, Algorithm 21 performs $n - 1$ calls to 2Sum or Fast2Sum. This accounts for a total of $V(n) = 6n - 6$ or $V^{fast}(n) = 3n - 3$ floating-point operations.

3.1.2 The VecSumErrBranch algorithm

Algorithm 22, illustrated in Figure 3.5, is a variation of VecSum presented above, consisting also in a chain of 2Sum, but instead of starting from the least significant, we start from the most significant component. Also, instead of propagating the sums we propagate the errors. If however, the error after a 2Sum block is zero, then we propagate the sum.

This algorithm can render different type of inputs ulp-nonoverlapping . The properties given in Theorem 3.1.11 and 3.1.14 are going to be used in Section 3.2, and the one in Theorem 3.1.17 in Section 3.4, for one of the two multiplication algorithms.

Algorithm 22 – VecSumErrBranch$(e_0, \ldots, e_{n-1}, m)$.

1: $j \leftarrow 0$
2: $\varepsilon_0 = e_0$
3: **for** $i \leftarrow 0$ **to** $n-2$ **do**
4: $\quad (r_j, \varepsilon_{i+1}) \leftarrow \text{2Sum}(\varepsilon_i, e_{i+1})$
5: $\quad$ **if** $\varepsilon_{i+1} \neq 0$ **then**
6: $\quad\quad$ **if** $j \geq m-1$ **then**
7: $\quad\quad\quad$ **return** $r_0, r_1, \ldots, r_{m-1}$ //enough output terms
8: $\quad\quad$ **end if**
9: $\quad\quad j \leftarrow j+1$
10: $\quad$ **else**
11: $\quad\quad \varepsilon_{i+1} \leftarrow r_j$
12: $\quad$ **end if**
13: **end for**
14: **if** $\varepsilon_{n-1} \neq 0$ **and** $j < m$ **then**
15: $\quad r_j \leftarrow \varepsilon_{n-1}$
16: **end if**
17: **return** $r_0, r_1, \ldots, r_{m-1}$

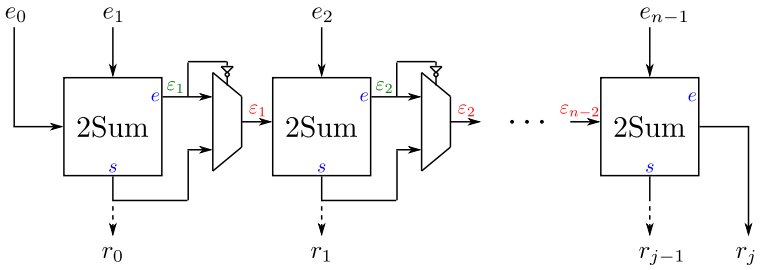


Figure 3.5 – Graphical representation of Algorithm 22. In the 2Sum calls the sum s is outputted downwards and the error e to the right.

Theorem 3.1.11. *Let $e = (e_0, \ldots, e_{n-1})$ an $\mathcal{S}$-nonoverlapping input expansion that may contain interleaving* 0*s and let m be an integer input parameter, with $1 \leq m \leq n-1$, the required number of output terms. Provided that no underflow/overflow occurs during computations, Algorithm 22 returns $r = (r_0, \ldots, r_{m-1})$, an* ulp*-nonoverlapping expansion, i.e., it satisfies $|r_{i+1}| \leq \text{ulp}(r_i)$ for all $0 \leq i < m-1$.*

Proof. The case when e contains 1 or 2 elements is trivial. Consider now at least 3 elements. By definition of 2Sum, we have $|\varepsilon_1| \leq \frac{1}{2}\,\text{ulp}(r_0)$ and by definition of $\mathcal{S}$-nonoverlapping ,

$$\begin{aligned} e_0 &= E_0 \cdot 2^{k_0} \text{ with } |e_1| < 2^{k_0}, \\ e_1 &= E_1 \cdot 2^{k_1} \text{ with } |e_2| < 2^{k_1}. \end{aligned}$$

Hence, r_0 and ε_1 are both multiples of 2^{k_1}. Two possible cases may occur:

(i) $\varepsilon_1 = 0$. If we choose to propagate directly $\varepsilon_1 = 0$, then $r_1 = e_2$ and $\varepsilon_2 = 0$. This implies by induction that $r_i = e_{i+1}, \forall i \geq 1$. So, directly propagating the error poses a problem, since the whole remaining chain of 2Sum is executed without any change on the input array. So, as shown in line 11, when $\varepsilon_{i+1} = 0$ we propagate the sum r_j.

(ii) $\varepsilon_1 \neq 0$. Then $|e_2| < |\varepsilon_1|$ and $|\varepsilon_1 + e_2| < 2\,|\varepsilon_1|$, from where we get $|r_1| = |\mathrm{RN}(\varepsilon_1 + e_2)| \leq 2\,|\varepsilon_1| \leq \mathrm{ulp}(r_0)$.

Now, we prove by induction the following statement: at step $i > 0$ of the loop in Algorithm 22, both r_{j-1} and ε_i are multiples of 2^{k_i} with $|e_{i+1}| < 2^{k_i}$. We proved above that for $i = 1$ it holds. Suppose now it holds for i and prove it for $i + 1$. Since r_{j-1} and ε_i are multiples of 2^{k_i} with $|e_{i+1}| < 2^{k_i}$ and $e_{i+1} = E_{i+1} \cdot 2^{k_{i+1}}$ with $|e_{i+2}| < 2^{k_{i+1}}$ (by definition of $\mathcal{S}$-nonoverlapping), it follows that both r_j and ε_{i+1} are multiples of $2^{k_{i+1}}$ (by definition of 2Sum).

Finally, we prove the relation between r_j and r_{j-1}. If $\varepsilon_{i+1} = 0$, we propagate r_j, i.e., $\varepsilon_{i+1} = r_j$. Otherwise $|e_{i+1}| < |\varepsilon_i|$, so $|e_{i+1} + \varepsilon_i| < 2\,|\varepsilon_i|$ and finally $|r_j| = |\mathrm{RN}(e_{i+1} + \varepsilon_i)| \leq 2\,|\varepsilon_i| \leq \mathrm{ulp}(r_{j-1})$. □

Remark 3.1.12. *We observe that in all practical cases the calls to 2Sum can be replaced by calls to Fast2Sum. This is possible because we have* $|s_{i+1}| \leq 2^{p-1}\,\mathrm{ulp}(x_i)$*, for* $p \geq 4$ *and* $n \leq 10$*, for* $p \geq 5$ *and* $n \leq 18$*, for* $p \geq 6$ *and* $n \leq 34$*, and so on, and our constraints on the expansion size are* $n \leq 39$ *for binary64 and* $n \leq 12$ *for binary32. Also* $\mathrm{ulp}(x_i) \leq 2^{-p+1}\,|x_i|$*. As a deduction* $|s_{i+1}| \leq |x_i|$*.*

In Example 3.1.13 we show the behavior described above using a floating-point toy system.

Example 3.1.13. *Consider a floating-point system with precision* $p = 5$*. Let* e *be a* 5*-term* $\mathcal{S}$*-nonoverlapping array:*

$$\begin{aligned} e_0 &= 1.0110_2 \times 2^{-1}, \\ e_1 &= -2^{-7}, \\ e_2 &= 0.0, \\ e_3 &= 2^{-10}, \\ e_4 &= 2^{-13}, \end{aligned}$$

and $m = 3$ *an integer. Using* e, m *as input for Algorithm 22 will result in the* ulp*-nonoverlapping array* r*:*

$$\begin{aligned} r_0 &= 1.0110_2 \times 2^{-1}, \\ r_1 &= -1.1100_2 \times 2^{-8}, \\ r_2 &= 2^{-13}. \end{aligned}$$

Theorem 3.1.14. *Let* $e = (e_0, e_1, \ldots, e_{n-1})$ *that satisfies* $|e_0| > |e_1| \geq \cdots > |e_{i-1}| \geq |e_i| > |e_{i+1}| \geq |e_{i+2}| > \cdots$ *and the* e_i*s are* $\mathcal{S}$*-nonoverlapping for strict inequality or they are equal to a power of* 2 *and let* m *be an integer input parameter, with* $1 \leq m \leq n - 1$*, the required number of output terms. Provided that no underflow/overflow occurs during computations, when applying Algorithm 22 on* e*, the output array* $r = (r_0, \ldots, r_{m-1})$ *is an* ulp*-nonoverlapping expansion, i.e., it satisfies* $|r_{i+1}| \leq \mathrm{ulp}(r_i)$ *for all* $0 \leq i < m - 1$*.*

Proof. The proof follows the exact same structure as the proof of Theorem 3.1.11, with very little modifications, so we do not detail it here. The main changes are:

- we can also have $|e_1| = |e_2| = 2^{k_1}$;
- we assume that $\cdots \leq |e_i| < |e_{i+1}| \leq |e_{i+2}| < \cdots$, so $|e_{i+1}| < 2^{k_i}$ and $e_{i+1} = E_{i+1} \cdot 2^{k_{i+1}}$ with $|e_{i+2}| \leq 2^{k_{i+1}}$;
- $|r_j| \leq \mathrm{ulp}(r_{j-1})$;
- $|e_{i+1}| \leq |\varepsilon_i|$, so $|e_{i+1} + \varepsilon_i| \leq 2\,|\varepsilon_i|$, with equality when $|e_{i+1}| = |e_{i+2}| = |\varepsilon_i| = 2^{k_{i+1}}$.

□

Remark 3.1.15. *We observe that the calls to 2Sum can still be replaced by calls to Fast2Sum, since, if $e_i = e_{i+1}$ they are equal to a power of 2, in which case the result is exact.*

A numerical example is given in Example 3.1.16.

Example 3.1.16. *Consider a floating-point system with precision $p = 5$. Let e be a 5-term array:*

$$\begin{aligned} e_0 &= -1.0011_2 \times 2^2, \\ e_1 &= 2^{-5}, \\ e_2 &= 2^{-5}, \\ e_3 &= 2^{-7}, \\ e_4 &= 1.1_2 \times 2^{-9}, \end{aligned}$$

and $m = 3$ an integer. Using e, m as input for Algorithm 22 will result in the ulp*-nonoverlapping array r:*

$$\begin{aligned} r_0 &= -1.0011_2 \times 2^2, \\ r_1 &= 1.0011_2 \times 2^{-4}, \\ r_2 &= -2^{-10}. \end{aligned}$$

The next theorem is going to be used for proving the correctness of the "accurate" multiplication algorithm in Section 3.4.

Theorem 3.1.17. *Let $e = (e_0, e_1, \ldots, e_{n-1})$ that satisfies: e_i is multiple of $2^{t_0 - ib} = 2^{t_i}$ and $|e_{i+1}| < 2^{t_i + c + 1}$, for all $0 \leq i < n-1$, where $b + c = p - 1$ ($c << b$). Also $|e_0| > |e_1|$ and $|e_0| < 2^{t_0 + p + 1}$. And let m be an integer input parameter, with $1 \leq m \leq n-1$, the required number of output terms. Provided that no underflow/overflow occurs during computations, when applying Algorithm 22 on e, the output array $r = (r_0, \ldots, r_{m-1})$ is an* ulp*-nonoverlapping expansion.*

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

$$\begin{aligned} e_0 &= E_0 \cdot 2^{t_0}, \\ e_1 &= E_1 \cdot 2^{t_1} \text{ with } t_1 = t_0 - b. \end{aligned}$$

Hence, r_0 and ε_1 are both multiples of $2^{t_0 - b}$. Two possible cases may occur:

(i) $\varepsilon_1 = 0$. If we choose to propagate directly ε_1, then $r_1 = e_2$ and $\varepsilon_2 = 0$. This implies, by induction, that $r_i = e_{i+1}, \forall i \geq 1$. So, directly propagating the error poses a problem, since the whole remaining chain of 2Sum is executed without any change between the input and the output. So, as shown line 11, when $\varepsilon_i = 0$ we propagate the sum r_j, so $\varepsilon_i \leftarrow r_j$.

(ii) $\varepsilon_1 \neq 0$. By definition of 2Sum, we have $|\varepsilon_1| \leq \frac{1}{2}\,\mathrm{ulp}(r_0)$. We also have $|e_0| > |e_1|$, so $|r_0| = |\mathrm{RN}(e_0 + e_1)| \leq 2\,|e_0|$. Hence: $|\varepsilon_1| < 2^{t_0 + 2}$.

We now prove by induction the following statement: at each step $i \geq 1$ of the loop in Algorithm 22, both r_{j-1} and ε_i are multiples of 2^{t_i} and $\varepsilon_i = 0$ or $|\varepsilon_i| < 2^{t_i + b + 2}$, 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 $\varepsilon_i + e_{i+1} = r_j + \varepsilon_{i+1}$. Since ε_i is a multiple of 2^{t_i} and e_{i+1} is a multiple of $2^{t_{i+1}}$, with $t_{i+1} = t_i - b$, then both r_j and ε_{i+1} are multiples of $2^{t_{i+1}}$. Two cases may occur:
– if $|\varepsilon_i| < 2^{t_i + c + 1}$ then $\varepsilon_i + e_{i+1}$ is a floating-point number, which implies $r_j = \varepsilon_i + e_{i+1}$ exactly

and $\varepsilon_{i+1} = 0$, in which case we propagate $r_j < 2^{t_i+c+2}$.
– if $|\varepsilon_i| > 2^{t_i+c}$ we have $|r_j| \leq 2\,|\varepsilon_i|$ and we get (by definition of 2Sum):

$$\begin{aligned} |\varepsilon_{i+1}| &\leq \frac{1}{2}\,\mathrm{ulp}(r_j) \\ &< 2^{-p} \cdot 2 \cdot 2^{t_i+b+2} \\ &< 2^{t_i-c+2} \end{aligned}$$

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

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

Remark 3.1.18. *In this case also we can replace the 2Sum calls with Fast2Sum calls, since we showed that the addition is exact for* $|e_{i+1}|, |\varepsilon_i| < 2^{t_i+c+1}$.

For an easier understanding of this setting we give a numerical example in Example 3.1.19.

Example 3.1.19. *Consider a floating-point system with precision* $p = 5$ *and* $b = 3, c = 1$. *Let* e *be a* 5*-term array:*

$$\begin{aligned} e_0 &= 1.0111_2 \times 2, \\ e_1 &= 1.1110_2 \times 2^{-3}, \\ e_2 &= 2^{-9}, \\ e_3 &= -1.011_2 \times 2^{-9}, \\ e_4 &= 1.1_2 \times 2^{-12}, \end{aligned}$$

and $m = 3$ *an integer. Using* e, m *as input for Algorithm 22 will result in the* ulp*-nonoverlapping array* r*:*

$$\begin{aligned} r_0 &= 1.1001_2 \times 2, \\ r_1 &= -1.0001_2 \times 2^{-6}, \\ r_2 &= 1.01_2 \times 2^{-11}. \end{aligned}$$

Remark 3.1.20. *After applying Algorithm 22 on an input that satisfies any of the above constraints, the output expansion cannot have interleaving zeros; zeros may appear only at the end of the expansion.*

In the worst case, Algorithm 22 performs $n - 1$ calls to Fast2Sum and $n - 2$ comparisons. This accounts for a total of $V_{err}^{fast}(n, m) = 4n - 5$ floating-point operations.

Now that we have all the prerequisites that we need, we can start looking into more complex algorithms for floating-point arithmetic.

3.2 Renormalization of floating-point expansions

As we explained, in order to ensure that the expansions carry significant information we require them to be nonoverlapping. Even if the input expansions satisfy this requirement, this property is often "broken" during the calculations. So, in order to ensure the precision related requirements, we may need to perform a (re-)normalization after each operation. This types of algorithms are an important brick for manipulating floating-point expansions.

In what follows we will present a renormalization algorithm given by Priest, followed by two new algorithms, that we developed.

3.2.1 Priest's renormalization algorithm

While several renormalization algorithms have been proposed in literature, Priest's [81] algorithm was the only one provided with a complete correctness proof. The proposed algorithm, with a slightly modified inner loop proposed by Nievergelt [76] [3] is the one showed in Algorithm 23. For an easier understanding of how the algorithm works we also illustrate an example with $n = 5$ in Figure 3.6. The "boxes" with a gradient color represent a 2Sum call followed by a conditional branch.

Algorithm 23 – Renormalize_Priest$(x_0, x_1, \ldots, x_{n-1})$.

$c \leftarrow x_{n-1}$
for $i \leftarrow n-2$ **to** 0 **do**
  $(c, f_{i+1}) \leftarrow \text{2Sum}(c, x_i)$
end for
$r_0 \leftarrow c$; $k \leftarrow 1$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $(r_{k-1}, d) \leftarrow \text{2Sum}(r_{k-1}, f_i)$
  if $d \neq 0$ **then**
    $\ell \leftarrow k-1$; $k \leftarrow k+1$
    while $\ell \geq 1$ **do**
      $(r_{\ell-1}, d') \leftarrow \text{2Sum}(r_{\ell-1}, r_\ell)$
      if $d' = 0$ **then**
        $k \leftarrow k-1$
      else
        $r_\ell \leftarrow d'$
      end if
      $\ell \leftarrow \ell - 1$
    end while
    $r_{k-1} \leftarrow d$
  end if
end for
return $r_0, r_1, \ldots r_{k-1}$

Priest proved that Theorem 3.2.1 holds, but for space constrains we do not include the proof.

Theorem 3.2.1. *Let $x_0, x_1, \ldots, x_{n-1}$ be an array of floating-point numbers that overlap by at most $p-2$ digits. Provided that no underflow/overflow occurs during the calculations, Algorithm 23 returns a $\mathcal{P}$-nonoverlapping expansion $r = (r_0, r_1, \ldots r_{k-1})$ with k terms, were $k \leq n$.*

In the worst case, Algorithm 23 performs a total of $R_{priest}(n) = 20(n-1)$ floating-point operations. One can easily see that the algorithm has many conditional branches, which make it even slower in practice. Even though modern processors have optimized branch prediction, this is not the case for GPUs. Furthermore, the decisions depend heavily on the input values, which may cause branch prediction to fail often.

3. Priest's version of the algorithm had a missed condition that would cause the inner loop to turn infinitely in some cases.

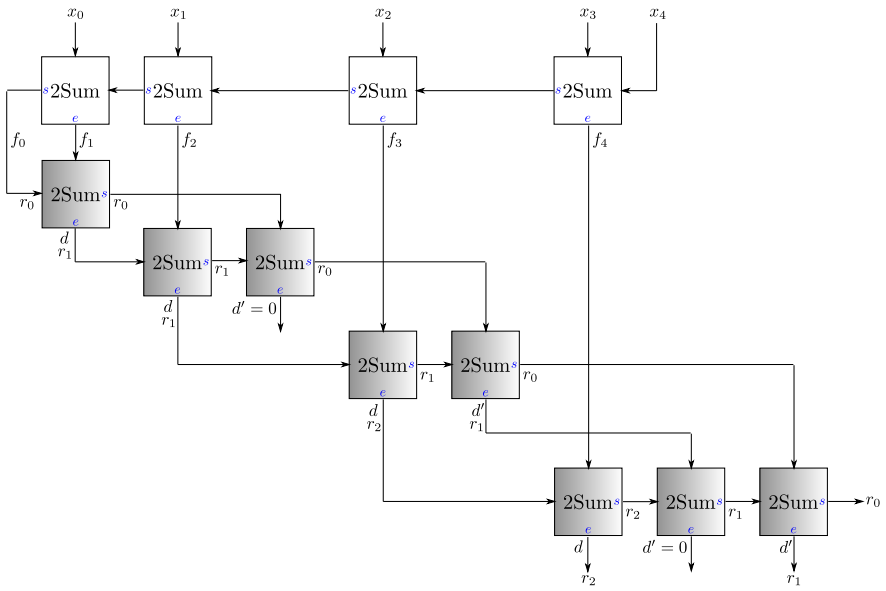


Figure 3.6 – Example of execution of Algorithm 23, for an input array with $n = 5$ terms.

3.2.2 A new renormalization algorithm

The algorithm that follows is actually a reduced version of Algorithm 6 in [41]. There we presented an algorithm with $m + 1$ levels that would render the result as an m-term $\mathcal{P}$-nonoverlapping floating-point expansion. Even though the last $m - 1$ levels do not use any branching, and take advantage of the pipeline, after tests and discussions, we decided that it is too expensive to ensure $\mathcal{P}$-nonoverlapping expansions. In practice, we are happy to use ulp-nonoverlapping expansion only, this is why we do not include here the full initial algorithm.

This step is crucial for assuring the "quality" of the expansion, but the proofs are complex and tedious and errors may have been left unnoticed, so having a formal proof would be ideal. To this end we collaborated with S. Boldo [4], for building a formal proof using the Coq proof assistant and the Floq library [86]. We were also able to prove that the algorithm also works in the presence of underflow, since the additions are exact (see Lemma 1.1.4), hypothesis that we dismiss here. This work is presented in [10], submitted to the 8th International Conference on Interactive Theorem Proving (ITP 2017).

Algorithm 24 (ilustrated in Figure 3.7) is based on different layers of chained 2Sum, that we grouped in simpler layers based on VecSum. It renders an array of overlapping numbers into an ulp-nonoverlapping expansion. Using some of the theorems proved in the previous section we will prove that this algorithm works in three different contexts.

Algorithm 24 – Renormalize$(x_0, x_1, \ldots, x_{n-1}, m)$.

1: $e[0 : n-1] \leftarrow \mathrm{VecSum}(x[0 : n-1])$
2: $r[0 : m-1] \leftarrow \mathrm{VecSumErrBranch}(e[0 : n-1], m)$
3: **return** $r_0, r_1, \ldots, r_{m-1}$

Remark 3.2.2. *Note that the first Fast2Sum in VecSumErrBranch can be skipped. We know that e_0 and e_1 are the result of the last 2Sum in the VecSum call. This means that $|e_1| \leq \frac{1}{2}\,\mathrm{ulp}(e_0)$, so $(r_0, \varepsilon_0) \leftarrow$ 2Sum(e_0, e_1) will return $r_0 = e_0$ and $\varepsilon_1 = e_1$.*

4. Researcher in the Toccata project-team, Inria Saclay, Paris, France.

Theorem 3.2.3 shows the general context in which the algorithm renders an array of overlapping numbers arranged in decreasing order of magnitude, into an ulp-nonoverlapping expansion.

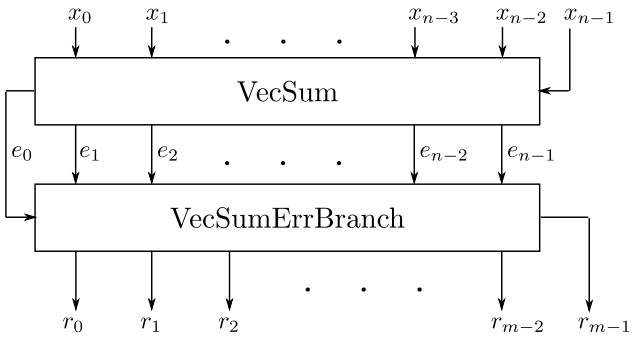


Figure 3.7 – Graphical representation of Algorithm 24.

Theorem 3.2.3. *Let $x_0, x_1, \ldots, x_{n-1}$ be an array of floating-point numbers that overlap by at most $d \leq p-2$ digits that may contain interleaving* 0s *and let m be an integer input parameter, with $1 \leq m \leq n-1$. Provided that no underflow/overflow occurs during the calculations, Algorithm 24 returns a "truncation" to m terms of an* ulp*-nonoverlapping floating-point expansion $r = r_0 + \cdots + r_{n-1}$ such that $x_0 + \cdots + x_{n-1} = r$.*

Proof. The proof is straightforward, using the properties in the previous section. From Theorem 3.1.3 we know that the first level transforms the overlapping input sequence into an $\mathcal{S}$-nonoverlapping expansion. After passing through the second level, from Theorem 3.1.11, we know that the output is an ulp-nonoverlapping expansion. □

In this context we can use the Fast2Sum algorithm for both levels, so in the worst case Algorithm 24 performs $R^{fast}(n,m) = V^{fast}(n) + V^{fast}_{err}(n,m) - 3 = 7n - 11$ floating-point operations.

Theorem 3.2.4 presents a different context in which the algorithm works. This is the case in which pairs of two numbers that overlap by p digits may appear in the input. We use this property when performing additions of floating-point expansions (for details see Section 3.3, Algorithm 27).

Theorem 3.2.4. *Let $x_0, x_1, \ldots, x_{n-1}$ be an array of floating-point numbers that overlap by at most $d \leq p-2$ digits that may contain pairs of at most 2 consecutive terms that overlap by p digits. The array may also contain interleaving* 0s*. Let m an integer input parameter, with $1 \leq m \leq n-1$. Provided that no underflow/overflow occurs during the calculations, Algorithm 24 returns a "truncation" to m terms of an* ulp*-nonoverlapping floating-point expansion $r = r_0 + \cdots + r_{n-1}$ such that $x_0 + \cdots + x_{n-1} = r$.*

Proof. The proof is similar to the proof of Theorem 3.2.3, but this time we use Theorem 3.1.7 and Theorem 3.1.14 from the previous section. □

Another (simpler) context in which the renormalization works is presented in Theorem 3.2.5. This is actually used for a special case of the addition, i.e., the addition of a floating-point expansion with a floating-point number.

Theorem 3.2.5. *Let $x_0, x_1, \ldots, x_{n-1}$ be an $\mathcal{S}$-nonoverlapping floating-point expansion that may contain interleaving* 0s*, t an arbitrary floating-point number and m be an integer input parameter, with $1 \leq m \leq n$. Provided that no underflow/overflow occurs during the calculations, Algorithm 24 returns a "truncation" to m terms of an* ulp*-nonoverlapping floating-point expansion $r = r_0 + \cdots + r_n$ such that $x_0 + \cdots + x_{n-1} + t = r$.*

Proof. The proof is straightforward using Theorem 3.1.9 and Theorem 3.1.11. □

In these last two contexts we cannot use Fast2Sum on the first level, so in the worst case Algorithm 24 performs $R(n,m) = V(n) + V_{err}^{fast}(n,m) - 3 = 10n - 14$ floating-point operations.

We give an overview of all the cases for which this renormalization algorithm returns an ulp-nonoverlapping expansion in Proposition 3.2.6.

Proposition 3.2.6. *Provided that Algorithm 24 receives as input one of the following:*

- *an n-term array x of floating-point numbers that overlap by at most $p-2$ digits that may contain interleaving* 0s*;*
- *an n-term array x of floating-point numbers that overlap by at most $p-2$ digits that may contain pairs of at most 2 consecutive terms that overlap by p digits and that may contain interleaving* 0s*;*
- *an $\mathcal{S}$-nonoverlapping floating-point expansion x with $n-1$ terms that may contain interleaving* 0s *and a random floating-point number x_{n-1};*

along with an integer m, the result $r = r_0, \ldots, r_{m-1}$ is going to be an ulp*-nonoverlapping floating-point expansion equal to x truncated to m terms.*

3.2.3 Renormalization of random numbers

Sometimes, in practice, we are not able to prove a uniform relationship between the numbers of an array (e.g. the monotony of the array). Even if this is the case we sometimes still need to render them ulp-nonoverlapping . To this end, we developed Algorithm 25 (illustrated in Figure 3.8) that satisfies Theorem 3.2.8.

Algorithm 25 – Renormalize_random$(x_0, x_1, \ldots, x_{n-1}, m)$.

1: $e_0^{(0)} \leftarrow x_0$
2: **for** $i \leftarrow 1$ **to** $n-1$ **do**
3: $\quad e^{(i)}[0:i] \leftarrow \text{VecSum}(e^{(i-1)}[0:i-1], x_i)$
4: **end for**
5: $r[0:m-1] \leftarrow \text{VecSumErrBranch}(e[0:n-1], m)$
6: **return** $r_0, r_1, \ldots, r_{m-1}$

Remark 3.2.7. *Note that we can also skip the first Fast2Sum in the VecSumErrBranch call, for the same reasons as for the previous algorithm.*

Theorem 3.2.8. *Let $x_0, x_1, \ldots, x_{n-1}$ be an array of random floating-point numbers that may contain interleaving* 0s *and let m be an integer input parameter, with $1 \leq m \leq n-1$. Provided that no underflow/overflow occurs during the calculations, Algorithm 25 returns a "truncation" to m terms of an* ulp*-nonoverlapping floating-point expansion $r = r_0 + \cdots + r_{n-1}$ such that $x_0 + \cdots + x_{n-1} = r$.*

Proof. It can be easily seen that during the first iteration of the for loop (lines 2 to 4), the VecSum algorithm is actually reduced to a call to 2Sum. Thence, we have $e_1^{(1)} \leq \frac{1}{2}\,\text{ulp}(e_0^{(1)})$, which implies that the two numbers are $\mathcal{S}$-nonoverlapping . In the second level we call VecSum on an $\mathcal{S}$-nonoverlapping expansion with 2 terms $(e_0^{(1)}, e_1^{(1)})$ and a random number, x_2.

By induction, using Theorem 3.1.9, it follows that at each iteration i of the loop we call VecSum on an $\mathcal{S}$-nonoverlapping expansion $(e_0^{(i-1)}, \ldots, e_{i-1}^{(i-1)})$ and a random number, x_i and we get the result $e_0^{(i)}, \ldots, e_i^{(i)}$, also an $\mathcal{S}$-nonoverlapping expansion.

In line 5 we apply VecSumErrBranch on an $\mathcal{S}$-nonoverlapping expansion, $e^{(n-1)}$, and from Theorem 3.1.11 we get that the resulted expansion r is ulp-nonoverlapping . □

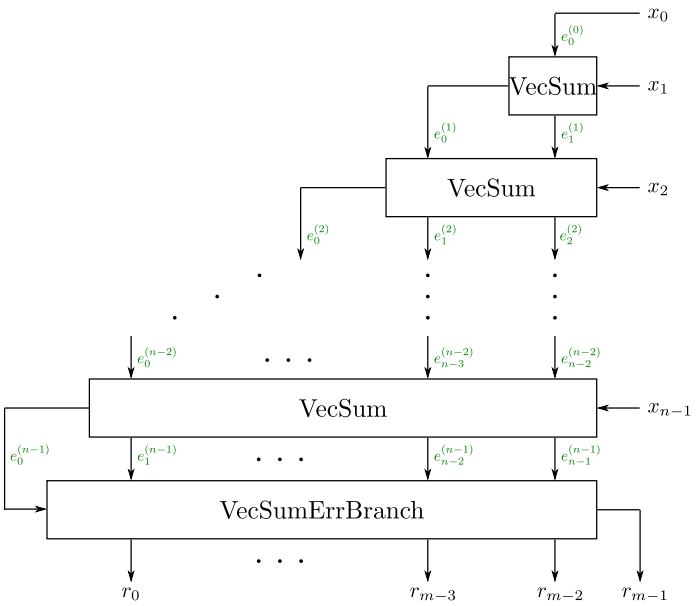


Figure 3.8 – Graphical representation of Algorithm 25.

In the worst case, Algorithm 25 calls $n-1$ times VecSum on i terms ($2 \leq i \leq n$), followed by a call to VecSumErrBranch using Fast2Sum. This accounts for a total of $R_{rand}(n,m) = \sum_{i=2}^{n} V(i) + V_{err}^{fast}(n,m) - 3 = 3n^2 + n - 8$ floating-point operations.

3.3 Addition of floating-point expansions

In general, an algorithm that performs the addition of two expansions x and y with n and m terms, respectively, will return a floating-point expansion with at most $n+m$ terms. This poses a problem when successive computations are done using the result, this is why a reduction of terms is required. This is done, using "truncation" and normalization methods, both *on-the-fly* or *a-posteriori*.

Note that, in this setting, subtraction can be performed simply by negating the floating-point terms in y.

3.3.1 Priest's addition algorithm

Many variants of algorithms that compute the sum of two expansions have been presented in the literature [81, 88, 34, 85]. One of the oldest algorithms was given by Priest [81]. His algorithm, Algorithm 26, is "merge-sorting the components of the two expansions by increasing magnitude and adding in this order", followed by applying his renormalization algorithm, in order to render the expansion $\mathcal{P}$-nonoverlapping .

This algorithm uses many conditional branches, and it has a worst case operation count of $A_{priest}(n,m) = 27(n+m) - 19$.

Algorithm 26 – Addition_Priest$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})$.

$i \leftarrow n-1, j \leftarrow m-1$
if $|x_i| < |y_i|$ **then**
  while $i > 0$ **and** $|x_{i-1}| \leq |y_j|$ **do**
    $e_{i+j} \leftarrow x_i, i \leftarrow i-1$
  end while
else if $|x_i| > |y_i|$ **then**
  while $j > 0$ **and** $|y_{j-1}| \leq |x_i|$ **do**
    $e_{i+j} \leftarrow y_j, j \leftarrow j-1$
  end while
end if
$a \leftarrow x_i, b \leftarrow y_j$
while $i > 0$ **or** $j > 0$ **do**
  $(c, e_{i+j}) \leftarrow \text{2Sum}(a, b)$
  $a \leftarrow c$
  if $i = 0$ **or** $(j > 0$ **and** $|y_{j-1}| < |x_{i-1}|)$ **then**
    $b \leftarrow y_{j-1}, j \leftarrow j-1$
  else
    $b \leftarrow x_{i-1}, i \leftarrow i-1$
  end if
end while
$(c, e_1) \leftarrow \text{2Sum}(a, b)$
$e_0 \leftarrow c$
$s[0:\ldots] \leftarrow \text{Renormalize\_Priest}(e[0:m+n-1])$ //using Alg. 23
return $s_0, s_1, \ldots$

In what follows we will present two algorithms for adding floating-point expansions, a new one and one that is a generalization of the algorithm used in the QD library [35]. For both algorithms we use the same *on-the-fly* truncation method, i.e., when computing the sum of two expansions with n and m terms, respectively, if the result is expected to have r terms, we take into consideration only the first r terms of each of the expansions. We give here, in Theorem 3.3.1, the error bound on the error caused by this truncation, since we are going to use it for both algorithms.

Theorem 3.3.1. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n and m terms, respectively. If, when computing the sum $x + y$ we "truncate" the input expansions to the most significant r terms, the error satisfies:*

$$\left| \sum_{i=r}^{n-1} x_i + \sum_{j=r}^{m-1} y_j \right| \leq (|x_0| + |y_0|) \frac{2^{-(p-1)r}}{1 - 2^{-(p-1)}}.$$

Proof. From the hypothesis we know that

$$|x_i| \leq 2^{-pi+i} |x_0|, \text{for all } 0 < i \leq n-1;$$
$$\text{and} \quad |y_j| \leq 2^{-pj+j} |y_0|, \text{for all } 0 < j \leq m-1.$$

Hence

$$\begin{aligned}\left|\sum_{i=r}^{n-1} x_i + \sum_{j=r}^{m-1} y_j\right| &\leq |x_0| \sum_{i=r}^{n-1} 2^{-(p-1)i} + |y_0| \sum_{j=r}^{m-1} 2^{-(p-1)j} \\ &\leq |x_0| \sum_{i=0}^{n-r-1} 2^{-(p-1)(i+r)} + |y_0| \sum_{j=0}^{m-r-1} 2^{-(p-1)(j+r)} \\ &\leq 2^{-(p-1)r} \left(|x_0| \sum_{i=0}^{n-r-1} 2^{-(p-1)i} + |y_0| \sum_{j=0}^{m-r-1} 2^{-(p-1)j} \right).\end{aligned}$$

When applying the formula $\sum_{k=0}^{\infty} t^k = \frac{1}{1-t}$ with $t = 2^{-(p-1)}$ we get

$$\left|\sum_{i=r}^{n-1} x_i + \sum_{j=r}^{m-1} y_j\right| \leq (|x_0| + |y_0|) \frac{2^{-(p-1)r}}{1 - 2^{-(p-1)}}.$$

□

3.3.2 "Accurate" addition algorithm

We first mentioned the algorithm that follows in [41], but a full correctness and accuracy proof was not previously published.

This first algorithm that we consider for adding two floating-point expansions consists in merging the two expansions in decreasing order of magnitude, and applying the renormalization algorithm (Algorithm 24) on the obtained array. The full algorithm is given in Algorithm 27 and illustrated in Figure 3.9.

Algorithm 27 – Addition_accurate$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}, r)$.

1: $n' \leftarrow \min(n, r)$; $m' \leftarrow \min(m, r)$
2: $f[0 : m' + n' - 1] \leftarrow$ Merge$(x[0 : n'], y[0 : m'])$ //using Alg. 28
3: $s[0 : r - 1] \leftarrow$ Renormalize$(f[0 : m' + n' - 1], r)$ //using Alg. 24
4: **return** $s_0, s_1, \ldots, s_{r-1}$

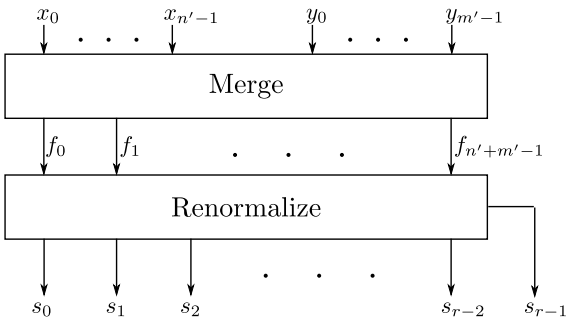


Figure 3.9 – Graphical representation of Algorithm 27.

For merging the input expansions we use the classical algorithm given in Algorithm 28 that has a time complexity of $\mathcal{O}(n + m)$, and in the worst case scenario performs $n + m$ floating-point comparisons.

Algorithm 28 – Merge$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})$.

```
1: i, j ← 0
2: for t ← 0 to n + m − 1 do
3:    if i = n or (j < m and |y_j| > |x_i|) then
4:       f_t ← y_j; j ← j + 1
5:    else
6:       f_t ← x_i; i ← i + 1
7:    end if
8: end for
9: return f_0, f_1, ..., f_{n+m−1}
```

Theorem 3.3.2. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n and m terms, respectively. Assume $p \geq 4$, which always holds in practice. Provided that no underflow/overflow occurs during the calculations, when computing their sum using Algorithm 27, the result s is an* ulp*-nonoverlapping floating-point expansion with r terms that satisfies:*

$$|x + y - s| < \frac{9}{2} 2^{-(p-1)r} (|x| + |y|).$$

Proof. After merging the two input arrays we obtain f that may contain pairs of at most 2 terms that overlap by p bits. From Theorem 3.2.4 we know that the renormalization (Algorithm 24) also works under these constraints, and the output is going to be an ulp-nonoverlapping expansion.

The array f contains the merged truncations to r terms of x and y, and from Theorem 3.3.1 we have

$$|x + y - f| \leq (|x_0| + |y_0|)\gamma_r, \tag{3.13}$$

with $\gamma_r = \frac{2^{-(p-1)r}}{1-2^{-(p-1)}}$.

Let $f' = f'_0, f'_1, \ldots$ be the ulp-nonoverlapping expansion equal to f one would obtain with a renormalization not limited to r-terms. Consequently, s is going to be a truncation to r terms of f'. The error caused by truncating (renormalizing) satisfies:

$$\begin{aligned} \left|f' - s\right| = \sum_{k=r}^{\infty} f'_k &\leq \left|f'_0\right| \sum_{k=r}^{\infty} 2^{-(p-1)k} \\ &\leq \left|f'_0\right| \sum_{k=0} 2^{-(p-1)(k+r)} \\ &\leq 2^{-(p-1)r} \left|f'_0\right| \sum_{k=0} 2^{-(p-1)k} \\ &\leq \left|f'_0\right| \frac{2^{-(p-1)r}}{1 - 2^{-(p-1)}} = \left|f'_0\right| \gamma_r. \end{aligned} \tag{3.14}$$

From (3.13) and (3.14) we get:

$$|x + y - s| \leq (|x_0| + |y_0| + \left|f'_0\right|)\gamma_r. \tag{3.15}$$

We want to express this as a function of x and y. We have $x - x_0 = \sum_{i=1}^{n-1} x_i \leq |x_0|\,\gamma_1$, hence $x_0 = \frac{x}{1-\varepsilon_1}$, with $|\varepsilon_1| < \gamma_1$. Analog for y_0 and f'_0. We can now rewrite (3.15) as

$$|x + y - s| \leq (|x| + |y| + \left|f'\right|)\frac{\gamma_r}{1 - \gamma_1}.$$

We know that $|f'| \leq ||x+y| + |\eta||$, with $|\eta| \leq (|x_0| + |y_0|)\gamma_r \leq (|x| + |y|)\frac{\gamma_r}{1-\gamma_1}$. Hence,

$$\begin{aligned} |x+y-s| &\leq \left[(|x| + |y|) \underbrace{\left(1 + \frac{\gamma_r}{1-\gamma_1}\right)}_{<2} + |x+y| \right] \frac{\gamma_r}{1-\gamma_1} \\ &< (2(|x| + |y|) + |x+y|) \frac{\gamma_r}{1-\gamma_1}, \end{aligned} \tag{3.16}$$

which is less than

$$\frac{9}{2} 2^{-(p-1)r} (|x| + |y|)$$

as soon as $p \geq 4$. □

In the worst case, Algorithm 27 performs $2r$ comparisons for merging, followed by a call to Renormalize on $2r$ terms. This accounts for a total of $A_{accurate}(n, m, r) = 2r + R(2r, r) = 22r - 14$ floating-point operations.

"Accurate" addition of a floating-point expansion with a floating-point number. In this case the aforementioned algorithm can be simplified: we observe that, by Theorem 3.2.5 (the third case of Proposition 3.2.6), the addition of an ulp-nonoverlapping expansion with a floating-point number can be performed by a simple renormalization using Algorithm 24. This allows us to give a tighter error bound on the result in Theorem 3.3.3 and to speed up the addition by not performing a merge.

Theorem 3.3.3. *Let x be an* ulp*-nonoverlapping floating-point expansion with n terms, and y a floating-point number. Provided that no underflow/overflow occurs during the calculations, when computing their sum using Algorithm 24, the result s is going to be an* ulp*-nonoverlapping floating-point expansion with r terms that satisfies:*

$$|x + y - s| < 2 \cdot 2^{-(p-1)r}(2|x| + |y|).$$

In the worst case this addition performs $A_{accurate}(n, 1, r) = R(r+1, r) = 10r - 4$ floating-point operations.

3.3.3 "Quick-and-dirty" addition algorithm

The algorithm that follows was mentioned in [46], but, the same as for the previous one, a full correctness and accuracy proof was not given.

The addition algorithm presented in Algorithm 29 and Figure 3.10 is a generalization of the algorithm for double-double and quad-double addition implemented in the QD library [34], combined with a different renormalization algorithm. Although we have implemented a fully customized version of the algorithm, that uses the same truncation method mentioned before, for simplicity, we give here only the "r input - r output" variant of it.

At step $n = 0$ we compute the exact sum $x_0 + y_0 = f_0 + e_0$. Since $|e_0| \leq \frac{1}{2}\,\mathrm{ulp}(f_0)$, we use the following intuition: let $\varepsilon = \frac{1}{2}\,\mathrm{ulp}(f_0)$, then, roughly speaking, if f_0 is of order of $\mathcal{O}(\Lambda)$, then e_0 is of order $\mathcal{O}(\varepsilon\Lambda)$. At each step $n = 1, \ldots, r$ we compute the exact result of $x_n + y_n = s'_n + e_n$, where s'_n and e_n are of order $\mathcal{O}(\varepsilon^n\Lambda)$ and $\mathcal{O}(\varepsilon^{n+1}\Lambda)$, respectively. From previous steps we have already obtained n error terms of order $\mathcal{O}(\varepsilon^n\Lambda)$ that we add together with s'_n to obtain the term f_n of order $\mathcal{O}(\varepsilon^n\Lambda)$ for the renormalization step. This addition is done in line 4, using VecSum. In the renormalization step, line 10, we use an extra error correction term, so we perform our "error-free transformation scheme" $r + 1$ times. The $(r + 1)$-th component f_r is obtained by a simple summation of the previously obtained terms of order $\mathcal{O}(\varepsilon^k\Lambda)$.

Algorithm 29 – Addition_quick$(x_0, \ldots, x_{r-1}, y_0, \ldots, y_{r-1}, r)$.

1: $(f_0, e_0) \leftarrow \text{2Sum}(x_0, y_0)$
2: **for** $n \leftarrow 1$ **to** $r-1$ **do**
3: $\quad (s_n, e_n) \leftarrow \text{2Sum}(x_n, y_n)$
4: $\quad (f_n, e[0:n-1]) \leftarrow \text{VecSum}(s_n, e[0:n-1])$
5: **end for**
6: $f_r \leftarrow 0$
7: **for** $i \leftarrow 0$ **to** $r-1$ **do**
8: $\quad f_r \leftarrow f_r + e_i$
9: **end for**
10: $s[0:r-1] \leftarrow \text{Renormalize\_random}(f[0:r], r)$ //using Alg. 25
11: **return** $s_0, s_1, \ldots, s_{r-1}$

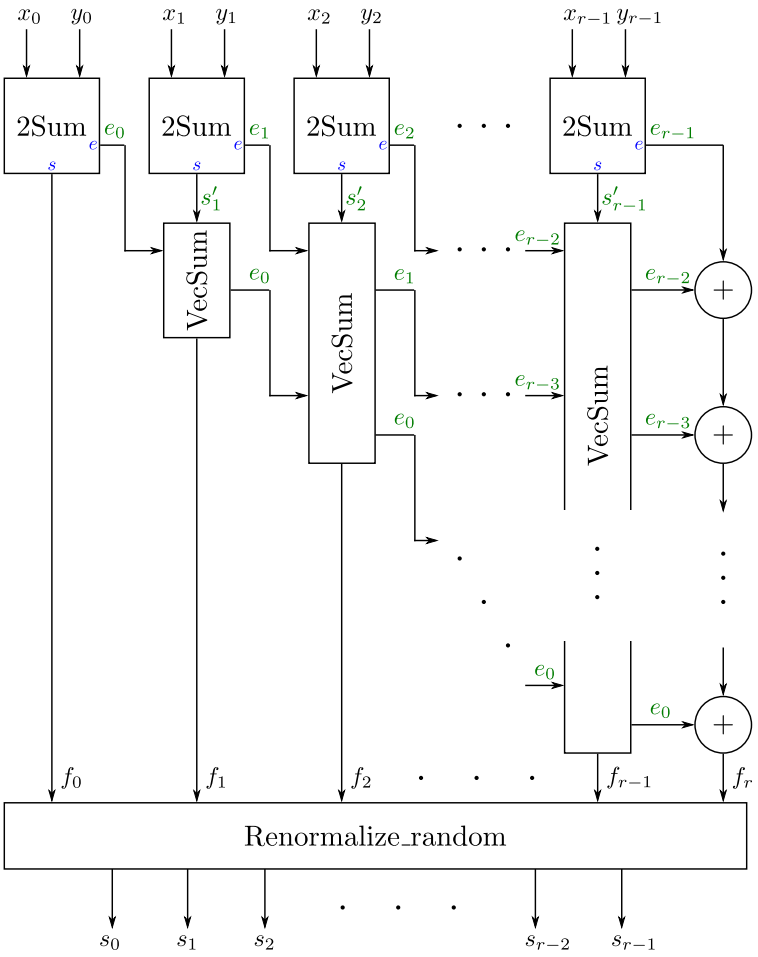


Figure 3.10 – Graphical representation of Algorithm 29.

Theorem 3.3.4. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n, and m terms, respectively. Assume $p \geq 4$, which always holds in practice. Provided that no underflow/overflow occurs during the calculations, when computing their sum using Algorithm 29, the result s is an* ulp*-nonoverlapping floating-point expansion with r terms that satisfies:*

$$|x + y - s| < 24 \cdot 2^{-(p-1)r}(|x| + |y|).$$

Proof. The use of Algorithm 25 (Renormalize_random) in the last step of the algorithm ensures that the output is an r-term ulp-nonoverlapping expansion.

As a first error source we have the truncation of the inputs. From Theorem 3.3.1 we have:

$$\left|x + y - (f_0 + s'_1 + \cdots + s'_{r-1})\right| \leq (|x_0| + |y_0|)\gamma_r, \tag{3.17}$$

with $\gamma_r = \frac{2^{-rp+r}}{1-2^{-(p-1)}} < 2^{-pr+r+1}$.

From the hypothesis we know that

$$\begin{aligned} |x_i| &\leq 2^{-pi+i}\,|x_0|\,, \text{for all } 0 < i \leq n-1; \\ \text{and} \quad |y_j| &\leq 2^{-pj+j}\,|y_0|\,, \text{for all } 0 < j \leq m-1, \end{aligned}$$

so $|x_n + y_n| \leq 2^{-np+n}(|x_0| + |y_0|)$.

A second source of error is the simple addition used for computing f_r. At this step we add $|e_{r-1}| \leq \frac{1}{2}\,\mathrm{ulp}(s'_{r-1}) \leq 2^{-rp+r-1}(|x_0| + |y_0|)$ with $r-1$ terms obtained from the VecSum in step $r-1$. All the terms are of the same order, and it can be shown by induction that e_{r-1} is the largest term in this addition, so the $r-1$ terms are bounded by the same value. As a deduction, at this step we perform $r-1$ simple additions that may cause an error bounded by

$$(r-1)2^{-(r+1)p+r}(|x_0| + |y_0|). \tag{3.18}$$

From (3.17) and (3.18) we get

$$|x + y - f| \leq (1 + (r-1)2^{-(p+1)})2^{-rp+r+1}(|x_0| + |y_0|). \tag{3.19}$$

Now, let $f' = f'_0, f'_1, \ldots$ be the ulp-nonoverlapping expansion equal to f one would obtain with a renormalization not limited to r-terms. Consequently, s is going to be a truncation to r terms of f'. The error caused by truncating (renormalizing) is the same as the one given in (3.14), i.e.,

$$\left|f' - s\right| \leq \left|f'_0\right|\gamma_r < 2^{-rp+r+1}\left|f'_0\right|, \tag{3.20}$$

From (3.19) and (3.20) we get:

$$|x + y - s| \leq (2 + (r-1)2^{-(p+1)})2^{-rp+r+1}(|x_0| + |y_0| + |f'_0|). \tag{3.21}$$

We want to express this as a function of x and y. We have $|x - x_0| = \left|\sum_{i=1}^{n-1} x_i\right| \leq |x_0|\,\gamma_1$, hence $x_0 = \frac{x}{1-\varepsilon_1}$, with $|\varepsilon_1| < \gamma_1$. Analog for y_0 and f'_0. We can now rewrite 3.21 as

$$|x + y - s| \leq (|x| + |y| + \left|f'\right|)\frac{(2 + (r-1)2^{-(p+1)})2^{-rp+r+1}}{1-\gamma_1}. \tag{3.22}$$

We know that $|f'| \leq ||x + y| + |\eta||$, with $|\eta| \leq (1 + (r-1)2^{-(p+1)})2^{-rp+r+1}(|x_0| + |y_0|) \leq (|x| + |y|)\frac{(1+(r-1)2^{-(p+1)})2^{-rp+r+1}}{1-\gamma_1}$.

Hence,

$$\begin{aligned} |x+y-s| &\leq \left[(|x|+|y|)\underbrace{\left(1+\frac{(1+(r-1)2^{-(p+1)})2^{-rp+r+1}}{1-\gamma_1}\right)}_{<2,\text{ if } p\geq 3}+|x+y|\right]\cdot \\ &\quad \cdot\frac{(2+(r-1)2^{-(p+1)})2^{-rp+r+1}}{1-\gamma_1} \\ &< (2(|x|+|y|)+|x+y|)\,\frac{\gamma_r}{1-\gamma_1}, \end{aligned}$$

which is less than

$$24\cdot 2^{-(p-1)r}(|x|+|y|),$$

as soon as $p \geq 4$. □

In the worst case, Algorithm 29 performs r 2Sum calls, $\sum_{i=1}^{r-1}$ calls to VecSum on i elements, $r-1$ simple additions, followed by the renormalization. This accounts for a total of $A_{quick}(n,m,r) = 6r + \sum_{i=1}^{r-1} V(i+1) + r - 1 + R_{rand}(r+1,r) = 6r^2 + 11r - 5$ floating-point operations.

Remark 3.3.5. *For the special case of adding a floating-point expansion with a floating-point number we use Theorem 3.3.3 which results in an important simplification:* $s_0,\ldots,s_{r-1} \leftarrow$ *Renormalize*$(x_0,\ldots,x_{n-1},y,r)$.

We have not explain yet why we call this algorithm "quick-and-dirty". It may be clear that "dirty" refers to the large error bound, but when looking at the operation count, the "quick" does not seem justified. The high cost of the algorithm comes mostly form the Renormalize_random algorithm that is used to ensure the ulp-nonoverlapping requirement on the output. There exist corner cases, as the one given in Example 3.3.6, in which cancellation happens on one of the f_is, causing them to be out of order. Because of cases like these we are unable to prove a strict relationship between the f_is, so, in theory, we are obliged to consider them random. Despite this, in practice, we can still get reliable results when replacing the call to Renormalize_random (Algorithm 25) with a call to Renormalize (Algorithm 24). In our implementation we offer both versions of the algorithm (see Section 1.4 for more details), and we prefer the later one if the computation is not critical, or if we can verify the result a-posteriori. We fill call this the Fast "quick-and-dirty" addition.

Example 3.3.6. *Consider the two 4-term expansions, with binary32 as underlying arithmetic:*

$$\begin{aligned} x_0 &= 1.00001101010100000110_2\times 2^{-1}, & y_0 &= -1.00001101010100000110011_2\times 2^{-1},\\ x_1 &= 2^{-24}, & y_1 &= 2^{-24},\\ x_2 &= 2^{-47}, & y_2 &= 2^{-47},\\ x_3 &= 2^{-70}; & y_3 &= -1.01110001001110000101111_2\times 2^{-72}. \end{aligned}$$

Using these expansions as input for Algorithm 29 will result in the array f:

$$\begin{aligned} f_0 &= 2^{-24},\\ f_1 &= 2^{-23},\\ f_2 &= 2^{-46},\\ f_3 &= 1.01000111011000111101_2\times 2^{-71}, \text{ and}\\ f_4 &= 2^{-95}. \end{aligned}$$

Since $f_0 < f_1$ we cannot ensure that the array f is ordered.

When replacing the call to Renormalize_random with a call to Renormalize, in the worst case, Algorithm 29 performs $A^{fast}_{quick}(n, m, r) = 3r^2 + 11r - 5$ floating-point operations.

Remark 3.3.7. *Note that even if the operation count still looks big, during execution, the VecSum levels will take great advantage of the computer pipeline, making Algorithm 29 faster that Algorithm 27 in most practical cases (see Section 3.7).*

3.4 Multiplication of floating-point expansions

In general, an algorithm that performs the multiplication of two expansions x and y with n and m terms, respectively, will return a floating-point expansion with at most $2nm$ terms. The same as in the case of addition this poses a problem when successive computations are done using the result, this is why we use "truncation" and renormalization.

The basic idea behind the multiplication algorithms is to compute partial products, the same as in the "paper-and-pencil" method, and accumulate them. The main difference between the algorithms consists in the method of choice for accumulating these partial products.

3.4.1 Priest's multiplication algorithm

One of the only algorithms provided with a full correctness proof was given by Priest [81]. He proposed an algorithm that uses the following reasoning:

- split x in two arrays, one containing the high order part of each term and another for the low order parts;
- split y in three arrays, using the same principle;
- compute all partial products $x[i] \times y$ into arrays; Priest proved that the partial products are exact, so this is done using simple multiplication;
- accumulate all the arrays to the result.

We present it in Algorithm 30. We denote by $b[0 : \ldots]$ and expansion b whose number of terms is not known in advance.

The cost of the algorithm is very high, because the accumulation of each array of partial products is done using successive addition and renormalization calls. The worst case operation count for this algorithm is $M_{priest}(n, m) = 81mn^2 + 747nm + 2m - 233n$. Note that the algorithm is not symmetrical, its cost depends on the order of the input expansions.

Algorithm 30 – Multiplication_Priest$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1})$.

Constant: $k_2 = \lfloor p/2 \rfloor,\ k_3 = \lfloor p/3 \rfloor + 1$

for $i \leftarrow 0$ **to** $n-1$ **do**
  $(x'_i, x''_i) \leftarrow \text{Split}(x_i, k_2)$ //using Alg. 4
end for
for $i \leftarrow 0$ **to** $m-1$ **do**
  $(y'_i, z) \leftarrow \text{Split}(y_i, k_3)$
  $(y''_i, y'''_i) \leftarrow \text{Split}(z, k_3)$
end for
$p_1 \leftarrow 0, k \leftarrow 1$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $m-1$ **do**
    $a_j^{(1)} \leftarrow \text{RN}(x'_i \cdot y'_j)$
    $a_j^{(2)} \leftarrow \text{RN}(x'_i \cdot y''_j)$
    $a_j^{(3)} \leftarrow \text{RN}(x'_i \cdot y'''_j)$
    $a_j^{(4)} \leftarrow \text{RN}(x''_i \cdot y'_j)$
    $a_j^{(5)} \leftarrow \text{RN}(x''_i \cdot y''_j)$
    $a_j^{(6)} \leftarrow \text{RN}(x''_i \cdot y'''_j)$
  end for
  $b[0:\ldots] \leftarrow \text{Renormalize\_Priest}(a^{(1)}[0:m-1])$ //using Alg. 23
  $c[0:\ldots] \leftarrow \text{Renormalize\_Priest}(a^{(2)}[0:m-1])$
  $b[0:\ldots] \leftarrow \text{Addition\_Priest}(c[0:\ldots], b[0:\ldots])$ //using Alg. 26
  $c[0:\ldots] \leftarrow \text{Renormalize\_Priest}(a^{(3)}[0:m-1])$
  $b[0:\ldots] \leftarrow \text{Addition\_Priest}(c[0:\ldots], b[0:\ldots])$
  $c[0:\ldots] \leftarrow \text{Renormalize\_Priest}(a^{(4)}[0:m-1])$
  $b[0:\ldots] \leftarrow \text{Addition\_Priest}(c[0:\ldots], b[0:\ldots])$
  $c[0:\ldots] \leftarrow \text{Renormalize\_Priest}(a^{(5)}[0:m-1])$
  $b[0:\ldots] \leftarrow \text{Addition\_Priest}(c[0:\ldots], b[0:\ldots])$
  $c[0:\ldots] \leftarrow \text{Renormalize\_Priest}(a^{(6)}[0:m-1])$
  $b[0:\ldots] \leftarrow \text{Addition\_Priest}(c[0:\ldots], b[0:\ldots])$
  $\pi[0:\ldots] \leftarrow \text{Addition\_Priest}(b[0:\ldots], \pi[0:\ldots])$
end for
return $\pi_0, \pi_1, \ldots$

In what follows we will present two multiplication algorithms, a new one and one that is a generalization of the algorithm used in the QD library for quad-double multiplication, each using a different method for accumulating the partial products.

Both algorithms use an "on-the-fly" truncation method, following the same reasoning as in the case of addition. Consider computing the product π with r terms of two floating-point expansions x and y, with n and m terms, respectively. Consider that the product x_0y_0 is of order $\mathcal{O}(\Lambda)$, then, for the product $(\pi', e) = 2\text{Prod}(x_i, y_j)$, π' is of order $\mathcal{O}(\varepsilon^k\Lambda)$ and e of order $\mathcal{O}(\varepsilon^{k+1}\Lambda)$, where $k = i + j$. In order to gain performance we discard the partial products that have an order of magnitude less than π_r, i.e., $0 \leq k \leq r$, meaning that we compute the approximate result based on the first $\sum_{k=0}^{r}(k+1)$ partial products. The products with the same order of magnitude as π_r are intended as an extra error correction term, that is why we compute them using only standard floating-point multiplication.

In Theorem 3.4.1, we bound the error caused by this truncation. The same bound will apply to both algorithms that we are going to present in this section.

Theorem 3.4.1. *Let x and y be two* ulp*-nonoverlapping floating-point 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 products, where r is the required size of the final result, then the error satisfies:*

$$\left| xy - \sum_{k=0}^{r} \sum_{i+j=k} x_i y_j \right| \leq |x_0 y_0|\, 2^{-(p-1)(r+1)} \left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}} \right).$$

Proof. From the definition of the ulp-nonoverlapping expansion we have $x_1 \leq \text{ulp}(x_0) \leq 2^{-p+1}\,|x_0|$ 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:

$$\begin{aligned}
\sum_{k=r+1}^{m+n-2} \sum_{i+j=k} a_i b_j &\leq \sum_{k=r+1}^{m+n-2} \sum_{i+j=k} 2^{-p(i+j)+i+j}\,|x_0 y_0| \\
&\leq |x_0 y_0| \sum_{k=r+1}^{m+n-2} \sum_{i+j=k} 2^{-(p-1)k} \\
&\leq |x_0 y_0| \sum_{k=r+1}^{m+n-2} (m+n-1-k) 2^{-(p-1)k} \\
&\leq |x_0 y_0| \sum_{k'=0}^{m+n-r-3} (m+n-k'-r-2) 2^{-(p-1)(k'+r+1)} \\
&\leq |x_0 y_0|\, 2^{-(p-1)(r+1)} \sum_{k'=0}^{m+n-r-3} (m+n-r-2-k') 2^{-(p-1)k'}.
\end{aligned} \tag{3.23}$$

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 (3.23) we get:

$$\sum_{k=r+1}^{m+n-2}\sum_{i+j=k} x_i y_j \leq |x_0 y_0|\, 2^{-(p-1)(r+1)}\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}}\right),$$

which concludes the proof. □

3.4.2 "Accurate" multiplication algorithm

Article [69], was fully dedicated to the algorithm presented in Algorithm 31. The algorithms called inside it are going to be detailed as we explain the reasoning behind it.

The main idea of the algorithm is to accumulate numbers of size at most b in "containers", referred to as "bins", that are floating-point 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 we present here a general setting with precision p, our implementation uses standard available formats. We made the following choices:

- when using binary64 we defined bins of size $b = 45$, which 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 binary32 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.

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

After the initialization step is done we start the actual computations. For each partial product computed using 2Prod, line 7, we get the pair (π', e) and, using the formula $\lfloor (t - t_{x_i} - t_{y_j})/b \rfloor$, we determine the corresponding bins in which we have to accumulate it, where t_{x_i} and t_{y_j} are the exponents of x_i and y_j, respectively.

Algorithm 31 – Multiplication_accurate$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}, r)$.

$t \leftarrow t_{x_0} + t_{y_0}$
for $i \leftarrow 0$ **to** $\lfloor r \cdot p/b \rfloor + 1$ **do**
  $B_i \leftarrow 1.5 \cdot 2^{t-(i+1)b+p-1}$
end for
for $i \leftarrow 0$ **to** $\min(n-1, r)$ **do**
  for $j \leftarrow 0$ **to** $\min(m-1, r-1-i)$ **do**
    $(\pi', e) \leftarrow$ 2Prod(x_i, y_j)
    $\ell \leftarrow t - t_{x_i} - t_{y_j}$
    $sh \leftarrow \lfloor \ell/b \rfloor$
    $\ell \leftarrow \ell - sh \cdot b$
    $B \leftarrow$ Accumulate(π', e, B, sh, ℓ) //using Alg. 32
  end for
  if $j < m - 1$ **then**
    $\pi' \leftarrow x_i \cdot y_j$
    $\ell \leftarrow t - t_{x_i} - t_{y_j}$
    $sh \leftarrow \lfloor \ell/b \rfloor$
    $\ell \leftarrow \ell - sh \cdot b$
    $B \leftarrow$ Accumulate$(\pi', 0., B, sh, \ell)$ //using Alg. 32
  end if
end for
for $i \leftarrow 0$ to $\lfloor r \cdot p/b \rfloor + 1$ **do**
  $B_i \leftarrow B_i - 1.5 \cdot 2^{t-(i+1)b+p-1}$
end for
$\pi[0 : r-1] \leftarrow$ VecSumErrBranch$(B[0 : \lfloor r \cdot p/b \rfloor + 1], r)$ //using Alg. 22
return $\pi_0, \pi_1, \ldots, \pi_{r-1}$

We know that one pair of floating-point numbers can "fall" into at most four bins, since $3b > 2p$, and we can deduce three different cases (see Figure 3.11):

- 2:2 case, in which both π' and e fall into two bins each;
- 2:3 case, in which π' falls into two bins and e into three;
- 3:2 case, in which π' falls into three bins and e into two.

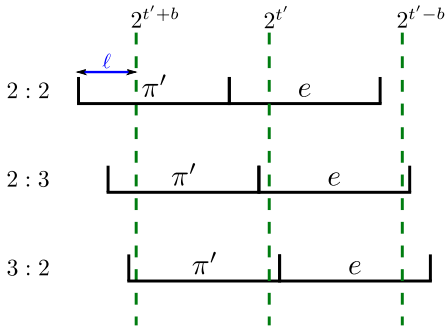


Figure 3.11 – Cases for accumulating the partial products into the bins.

These cases are dealt with in Algorithm 32 that accumulates the partial products. Apart from the (π', 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 $t - t_{x_i} - t_{y_j} - sh \cdot b$, 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 (π', e) pair.

Algorithm 32 – Accumulate(π', e, B, sh, ℓ).

if $\ell < b - 2c - 1$ **then**
  $(B_{sh}, B_{sh+1}) \leftarrow$ Deposit(π') // that is, $(B_{sh}, \pi') \leftarrow$Fast2Sum$(B_{sh}, \pi')$, and
    // $B_{sh+1} \leftarrow B_{sh+1} + \pi'$
  $(B_{sh+1}, B_{sh+2}) \leftarrow$ Deposit(e)
else if $\ell < b - c$ **then**
  $(B_{sh}, B_{sh+1}) \leftarrow$ Deposit(π')
  $(B_{sh+1}, e) \leftarrow$ Fast2Sum(B_{sh+1}, e)
  $(B_{sh+2}, B_{sh+3}) \leftarrow$ Deposit(e)
else
  $(B_{sh}, p) \leftarrow$ Fast2Sum(B_{sh}, π')
  $(B_{sh+1}, B_{sh+2}) \leftarrow$ Deposit(π')
  $(B_{sh+2}, B_{sh+3}) \leftarrow$ Deposit(e)
end if
return B

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 3.4.2. *For simplicity, we consider that the extra error correction partial products, the ones that are computed using only standard floating-point multiplication, are dealt with the same way, using the pair $(\pi', 0)$. This is not the case in our implementation, where we save operations by accumulating only the π' term.*

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 (VecSumErrBranch algorithm). This type of addition was first used by Rump (in [84]) for adding the elements of an array of floating-point 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 32. We consider all the values that fall into the same bins B_{sh}, B_{sh+1} as an array of floating-point numbers $x_1, \ldots, x_n$ that satisfy $|x_i| < 2^{t+b}$, where 2^t 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}.

Theorem 3.4.3. *Let $x_1, \ldots, x_n$ an array of floating-point numbers that satisfy $|x_i| < 2^{t+b}$, for all $0 < i \leq n$, where $b < p - 1$. We initialize a floating-point container with $s_0 = 1.5 \cdot 2^{t+p-1}$, we compute $s_1 = \mathrm{RN}(s_0 + x_1); \ldots; s_i = \mathrm{RN}(s_{i-1} + x_i); \ldots; s_n = \mathrm{RN}(s_{n-1} + x_n)$; and we return the value $\mathrm{RN}(s_n - s_0)$. For each x_i we also compute the lower part $x_i^\ell = \mathrm{RN}(\mathrm{RN}(s_{i-1} - s_i) + x_i)$. When using this method, no significant informations is lost in the process, so 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:

$$1.5 \cdot 2^{t+p-1} - i \cdot 2^{t+b} \leq s_i \leq 1.5 \cdot 2^{t+p-1} + i \cdot 2^{t+b}. \tag{3.24}$$

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:

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

Hence, since rounding is a monotonic function:

$$\begin{aligned} \mathrm{RN}(1.5 \cdot 2^{t+p-1} - (i+1)2^{t+b}) &\leq s_{i+1} \\ &\leq \mathrm{RN}(1.5 \cdot 2^{t+p-1} + (i+1)2^{t+b}). \end{aligned}$$

The value $1.5 \cdot 2^{t+p-1} - (i+1)2^{t+b}$ is an exact floating-point number because it is a multiple of 2^t and it is less than 2^{t+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 binary64, $p = 53$ and $b = 45$, and
- $i \leq 111$ for binary32, $p = 24$ and $b = 18$.

The same holds for $1.5 \cdot 2^{t+p-1} + (i+1)2^{t+b}$ provided that $i \leq 2^{p-b-2} - 1$, which also holds in all practical cases (with our parameters: $i \leq 63$ for binary64 and $i \leq 15$ for binary32).

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

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

such that, by induction,

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

which implies

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

From (3.24), we easily find that s_n and s_0 are within a factor 2 (in practice, much less), such that (from Sterbenz lemma - Lemma 1.1.2), their difference is exactly computed: $\mathrm{RN}(s_n - s_0) = s_n - s_0$. We therefore conclude that

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

□

The VecSumErrBranch (Algorithm 22) call in the last step of the multiplication has the role of a renormalization. By Theorem 3.1.17 that we proved in Section 3.1, we see that this step is enough for rendering the bins array into an ulp-nonoverlapping expansion.

Theorem 3.4.4. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n, and m terms, respectively. Provided that no underflow/overflow occurs during the calculations, when computing their product using Algorithm 31, the result π is an* ulp*-nonoverlapping floating-point expansion with r terms that satisfies:*

$$|xy - \pi| \leq |x_0 y_0|\, 2^{-(p-1)r} \left[1 + (r+1)2^{-p} + 2^{-(p-1)} \left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}}\right)\right].$$

Proof. When using Algorithm 31 we "truncate" the result by discarding the partial products with an order of magnitude less than π_r. From Theorem 3.4.1 we know that this causes a maximum error that is less or equal to

$$|x_0 y_0|\, 2^{-(p-1)(r+1)} \left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}}\right). \tag{3.25}$$

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 $|x_i y_j| \leq 2^{-(p-1)(i+j)} |x_0 y_0|$, from where $|x_i y_j - \mathrm{RN}(x_i y_j)| \leq 2^{-(p-1)r} |x_0 y_0|\, 2^{-p}$. This implies that the maximum error caused doing this is less or equal to:

$$(r+1)\, |x_0 y_0|\, 2^{-(p-1)r} \cdot 2^{-p}. \tag{3.26}$$

Apart from these two possible errors we also need to account for the error caused by the renormalization step. In Theorem 3.1.17 we showed that Algorithm 22 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:

$$|x_0 y_0|\, 2^{-(p-1)r}. \tag{3.27}$$

To get the final error bound we have to add the bounds on all the possible errors that can occur, i.e., (3.25), (3.26) and (3.27), and we get:

$$|x_0 y_0|\, 2^{-(p-1)r} \left[1 + (r+1)2^{-p} + 2^{-(p-1)} \left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}}\right)\right]$$

This concludes our proof. □

During the execution of Algorithm 31 we perform the following steps:

- 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 floating-point operation, which we call $2r$ times;
- the first step of the algorithm consists in initializing the bins; we allocate $binNr = \left\lfloor \frac{r \cdot p}{b} \right\rfloor + 2$ 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 $binNr - 1$ floating-point multiplications;
- during the main loop of the algorithm we compute $\sum_{i=1}^{r} i$ partial products using 2Prod, which we accumulate into the bins using 3 Fast2Sum calls and 2 floating-point additions. During this same loop we also compute $r-1$ correction terms using simple floating-point multiplication, which we can accumulate using only 2 Fast2Sum calls and one floating-point addition.

- in the last part of the algorithm we first unbias the bins, by subtracting their initial values from each one of them, then we renormalize the result.

In the worst case, this account for a total of $M_{accurate}(n, m, r) = \frac{13}{2}r^2 + \frac{33}{2}r + 6(\lfloor \frac{r \cdot p}{b} \rfloor + 2) + 54$ floating-point operations.

3.4.3 "Quick-and-dirty" multiplication algorithm

The below algorithm was first mentioned in [46] and after that in [41], but a full correctness and accuracy prove was never given.

The algorithm presented in this section is also a generalization of the quad-double multiplication implemented in the QD library, coupled with a different renormalization algorithm. In Algorithm 33 and Figure 3.12 we present the *"r input - r output"* variant, although in our implementation we have a fully customized version.

Algorithm 33 – Multiplication_quick$(x_0, \ldots, x_{r-1}, y_0, \ldots, y_{r-1}, r)$.

1: $(f_0, e_0) \leftarrow$ 2Prod(x_0, y_0)
2: **for** $k \leftarrow 1$ **to** $r - 1$ **do**
3: **for** $i \leftarrow 0$ **to** k **do**
4: $(\pi'_i, \hat{e}_i) \leftarrow$ 2Prod(x_i, y_{k-i})
5: **end for**
6: $(f_k, e[0 : k^2 + k - 1]) \leftarrow$ VecSum$(\pi'[0 : k], e[0 : k^2 - 1])$
7: $e[0 : k^2 + 2k] \leftarrow (e[0 : k^2 + k - 1], \hat{e}[0 : k])$
8: **end for**
9: $f_r \leftarrow 0$
10: **for** $i \leftarrow 1$ **to** $r - 1$ **do**
11: $f_r \leftarrow f_r + x_i \cdot y_{r-i}$
12: **end for**
13: **for** $i \leftarrow 0$ **to** $r^2 - 1$ **do**
14: $f_r \leftarrow f_r + e_i$
15: **end for**
16: $\pi[0 : r - 1] \leftarrow$ Renormalize_random$(f[0 : r], r)$
17: **return** $\pi_0, \pi_1, \ldots, \pi_{r-1}$

We perform the same "on-the-fly" truncation as in the previous algorithm, but we accumulate the values using a different method, based on the VecSum algorithm, in which we add all the products with the errors of the same order resulted from the previous step. So, we already established that for each k, $0 \leq k \leq r$, we have $k+1$ products to compute (line 4). Besides these we also have k^2 terms resulting from the previous iteration. We accumulate all these terms using VecSum, to obtain f_k in line 6. The remaining error terms are concatenated with the errors from the $k + 1$ products, and the entire array $e_0, \ldots, e_{k^2+2k}$ is propagated to the next iteration. The $(r + 1)$-st component f_r is obtained by simple summation of all remaining errors with the simple products of order $\mathcal{O}(\varepsilon^k \Lambda)$, where $\mathcal{O}(\Lambda)$ is the order of $x_0 \times y_0$. Error-free transforms are not needed in the last step since the errors are not reused. The array f is then reused in order to render the result ulp-nonoverlapping .

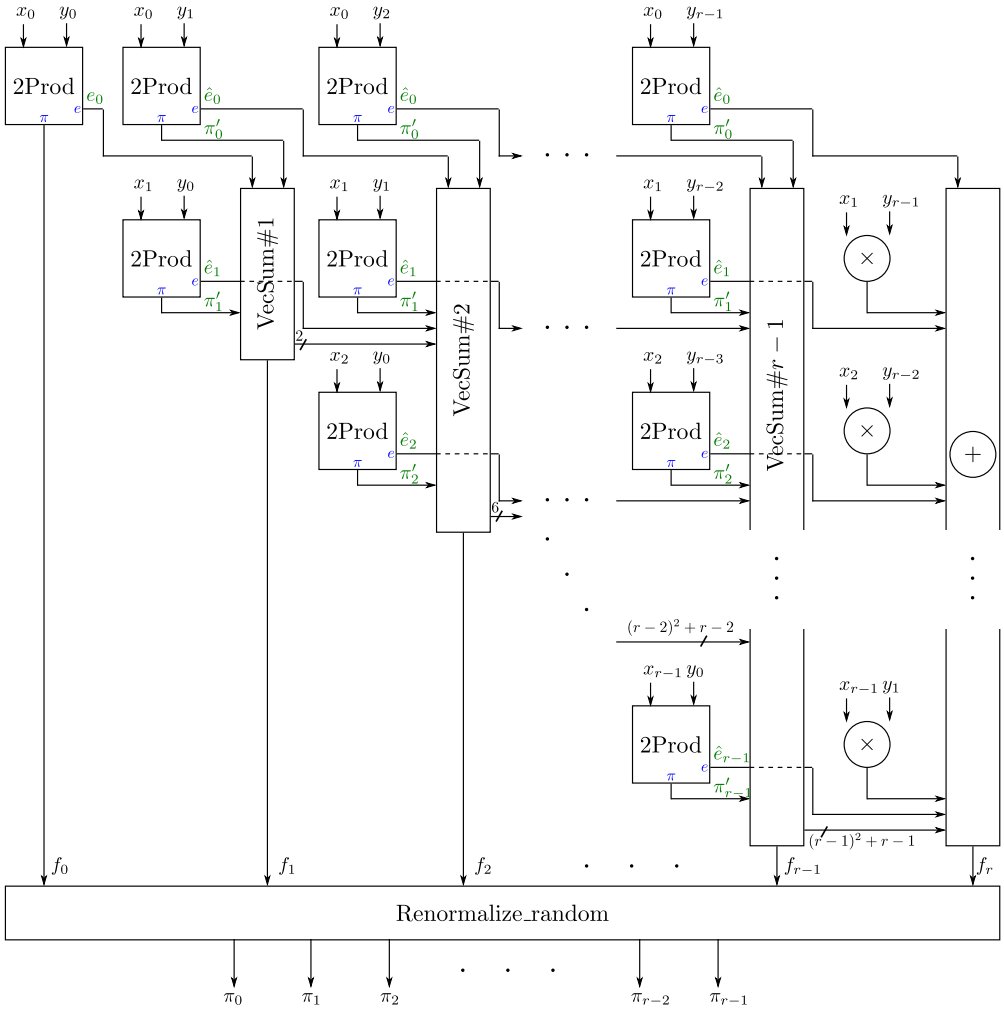


Figure 3.12 – Graphical representation of Algorithm 33.

Theorem 3.4.5. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n, and m terms, respectively. Assume $p \geq 8$. Provided that no underflow/overflow occurs during the calculations, when computing their product using Algorithm 33, the result π is an* ulp*-nonoverlapping floating-point expansion with r terms that satisfies:*

$$|xy - \pi| \leq |x_0 y_0|\, 2^{-(p-1)(r+1)} \left(\frac{128}{127}(m+n) - \frac{129}{254}r - \frac{385}{254} + 2^{p-1} + 2^{-p-r}(r^2+r)((r+1)!)^2 \right).$$

Proof. When using Algorithm 33 we "truncate" the result by discarding the partial products with an order of magnitude less than π_r. From Theorem 3.4.1 we know that this causes a maximum error that is less than or equal to

$$|x_0 y_0|\, 2^{-(p-1)(r+1)} \left(\underbrace{\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2}}_{<0} + (m+n-r-2) \underbrace{\frac{1}{1-2^{-(p-1)}}}_{<\frac{128}{127} \text{ for } p \geq 8} \right). \tag{3.28}$$

From the definition we know that $|x_i| \leq 2^{-i(p-1)} |x_0|$ and $|y_j| \leq 2^{-j(p-1)} |y_0|$, so we can deduce

$$|x_i y_j| \leq 2^{-(p-1)(i+j)} |x_0 y_0|.$$

For computing f_0 we use only 2Prod(x_0, y_0), and we get $|f_0| \leq |x_0 y_0|\,(1 + 2^{-p})$.

For computing f_1 we use VecSum#1, with 3 entries: the error from the previous step, and two partial products, which are less than $2^{-(p-1)}\,|x_0 y_0|\,(1 + 2^{-p})$. It is easily seen that all these entries are bounded by the same value. We define the following notation:

$$\Omega_1 = 2^{-(p-1)}\,|x_0 y_0|\,(1 + 2^{-p}). \tag{3.29}$$

It follows that $f_1 < 3 \cdot 2^{-(p-1)}\,|x_0 y_0|\,(1 + 2^{-p})^3$ and the outputted errors are less than $\Omega_2 = 3 \cdot 2^{-2p+1}\,|x_0 y_0|\,(1 + 2^{-p})^3$.

For computing f_2 we use VecSum#2 which is going to have 7 entries: 2 errors outputted by VecSum#1, bounded by Ω_2; 2 errors from the previous step's partial products, which are less than $2^{-2p+1}\,|x_0 y_0|\,(1 + 2^{-p})$; and 3 partial products, less than $2^{-2(p-1)}\,|x_0 y_0|\,(1 + 2^{-p})$. We observe once more that all the entries are less than Ω_2.

For the induction step we consider VecSum#$i - 1$. For computing f_{i-1} we have $(i-1)^2 + i$ entries, which we assume are all less than Ω_{i-1}. It follows:

$$\begin{aligned} f_{i-1} &< (2\Omega_{i-1}(1 + 2^{-p}) + \Omega_{i-1})(1 + 2^{-p}) + \cdots \\ &< (i^2 - i + 1)\Omega_{i-1}(1 + 2^{-p})^{i^2 - i} \end{aligned}$$

and also, the largest error term outputted and implicitly all the others are less than $\frac{1}{2}\,\mathrm{ulp}(f_{i-1})$.

This implies that all error terms are less than:

$$\begin{aligned} & (i^2 - i + 1)\Omega_{i-1}(1 + 2^{-p})^{i^2 - i} \cdot 2^{-p}; \\ = \;& 2^{-(p-1)}\,|x_0 y_0|\,(1 + 2^{-p}) \prod_{n=1}^{i} (n^2 - n + 1)(1 + 2^{-p})^{n^2 - n} 2^{-p}; \\ = \;& 2^{-(p-1)-ip}\,|x_0 y_0| \underbrace{(1 + 2^{-p})^{1+2+\cdots+(i^2-i)}}_{<2 \text{ always true in practice}} \prod_{n=1}^{i} (n^2 - n + 1); \\ < \;& 2^{-(i+1)p+2}\,|x_0 y_0|\,(i!)^2. \end{aligned} \tag{3.30}$$

and this last value will define Ω_i.

This implies that, since we use only simple summation for computing f_r, in the last step we neglect $r^2 + r$ terms, all less than Ω_{r+1}.

We also have to account for the errors that occur when computing the last partial products using only simple multiplication. This means $r + 1$ terms less than $|x_0 y_0|\,2^{-(p-1)r-p} = \frac{1}{2}\,|x_0 y_0|\,2^{-(p-1)(r+1)}$.

When adding all these errors we get the following bound:

$$\begin{aligned} |xy - f| &\leq |x_0 y_0|\,2^{-(p-1)(r+1)} \left(\frac{128}{127}(m + n - r - 2) + (r^2 + r)2^{-p-r+1}((r+1)!)^2 + \frac{r+1}{2} \right) \\ &\leq |x_0 y_0|\,2^{-(p-1)(r+1)} \left(\frac{128}{127}(m + n) - \frac{129}{254}r - \frac{385}{256} + (r^2 + r)2^{-p-r+1}((r+1)!)^2 \right). \end{aligned} \tag{3.31}$$

Now, let $f' = f'_0, f'_1, \ldots$ be an ulp-nonoverlapping expansion equal to f. Consequently, π is going to be a truncation to r terms of f'. For renormalizing the result we use Algorithm 25, which ensures that the result is going to be 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:

$$|x_0 y_0|\,2^{-(p-1)r} = |x_0 y_0|\,2^{-(p-1)(r+1)+(p-1)}.$$

To get the final error bound we have to add the two bounds obtained above and we get:

$$|xy - \pi| \leq |x_0 y_0|\, 2^{-(p-1)(r+1)} \left(2^{p-1} + \frac{128}{127}(m+n) - \frac{129}{254}r - \frac{385}{254} + (r^2+r)2^{-p-r+1}((r+1)!)^2\right).$$

And this concludes our proof. □

We ask the reader to keep in mind that this bound is a big overestimate to the error. When bounding the error given by the simple addition in the last step of the algorithm we assume that all the terms added are bounded by the same value, even if this is not the case. In practice we still obtain reliable results using this algorithm.

During the execution of Algorithm 33 we perform the following steps:

- we compute $\sum_{i=1}^{r} i$ partial products using 2Prod;
- for computing each f_i, with $1 \leq i \leq r-1$ we perform $\sum_{n=1}^{r-1} V(n^2+n+1)$ floating-point operations;
- we compute $r-1$ partial product using only simple multiplication;
- we perform $r^2 + r - 1$ simple additions in order to get the extra error correction term, f_r;
- followed by the renormalization of the result using Renormalize_random.

In the worst case, this account for a total of $M_{quick}(n,m,r) = 2r^3 + 5r^2 + 8r - 6$ floating-point operations.

Algorithm 33 can be slightly improved if an FMA instruction is available, by performing the accumulation in line 11 using $f_r \leftarrow \text{fma}(x_i, y_{r-1}, f_r)$. This change would imply that in the last step of the algorithm we ignore only $r^2 - 1$ terms less than Ω_{r+1} instead of $r^2 + r$, which slightly improves the error bound:

$$|xy - \pi| \leq |x_0 y_0|\, 2^{-(p-1)(r+1)} \left(2(m+n) - r - 2 + (r^2-1)((r+1)!)^2\right).$$

The floating-point operation count becomes $M_{quick}[fma](n,m,r) = 2r^3 + 5r^2 + 7r - 5$ in the worst case.

The "quick" appellative has the same meaning as for addition. In this case we also replace the call to Renormalize_random with a call to Renormalize (for details see Section 1.4), so we get $M^{fast}_{quick}(n,m,r) = 2r^3 + 2r^2 + 8r - 6$ or $M^{fast}_{quick}[fma](n,m,r) = 2r^3 + 2r^2 + 7r - 5$, respectively.

3.5 Division of floating-point expansions

There are two classes of algorithms for performing division: the so-called *digit-recurrence algorithms* [23], that generalize the "paper-and-pencil" method, and the algorithms based on the Newton-Raphson iteration [98, 17]. The algorithms employed so far for dividing expansions belong to the former class, as Priest's algorithm presented below. In our work we focused on the possible use of the latter class, since its very fast, quadratic convergence is appealing when high precision is at stake.

3.5.1 Classical long division algorithm

In [81] Priest's division is done using the classical long division algorithm, which is recalled in Algorithm 34. We denote by $f[0:\ldots]$ and expansion f whose number of terms is not known in advance.

Algorithm 34 – Division_Priest$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}, d)$.

1: $q_0 = \mathrm{RN}(x_0/y_0)$
2: $r^{(0)}[0 : n-1] \leftarrow x[0 : n-1]$
3: **for** $i \leftarrow 1$ **to** $d-1$ **do**
4: $\quad f[0 : \ldots] \leftarrow$ Multiplication_Priest $(q_{i-1}, y[0 : m-1])$ //using Alg. 30
5: $\quad r^{(i)}[0 : \ldots] \leftarrow$ Addition_Priest$(r^{(i-1)}[0 : \ldots], -f[0 : \ldots]))$ //using Alg. 26
6: $\quad q_i = \mathrm{RN}(r_0^{(i)}/y_0)$
7: **end for**
8: $q[0 : \ldots] \leftarrow$ Renormalize_Priest $(q[0 : d-1])$ //using Alg. 23
9: **return** $q_0, q_1, \ldots$ with at most d terms

The division algorithm implemented in the QD library [34] is similar. For instance, let $x = x_0 + x_1 + x_2 + x_3$ and $y = y_0 + y_1 + y_2 + y_3$ be quad-double numbers. First, one approximates the quotient $q_0 = x_0/y_0$, then computes the remainder $r = x - q_0 y$ in quad-double. The next correction term is $q_1 = r_0/y_0$. Subsequent terms q_i are obtained by continuing this process. At each step when computing r, full quad-double multiplication and subtraction are performed since most of the bits will be canceled out when computing q_3 and q_4. A renormalization step is performed only at the end, on $q_0 + q_1 + q_2 + ...$ in order to ensure the nonoverlapping requirement. No error bound is given in [34].

Note that in Algorithm 34 a renormalization step is performed after each computation of $r = r - q_i y$. Priest proved in [81] the error bound given in Theorem 3.5.1.

Theorem 3.5.1. *Let x and y be two $\mathcal{P}$-nonoverlapping floating-point expansions, with n, and m terms, respectively and d an integer parameter, the required number of output terms. Provided that no underflow/overflow occurs during the calculations, Algorithm 34 computes a quotient $\mathcal{P}$-nonoverlapping expansion $q = q_0 + \cdots + q_{d-1}$ that satisfies:*

$$\left| \frac{q - x/y}{x/y} \right| < 2^{1-\lfloor (p-4)d/p \rfloor}. \tag{3.32}$$

Daumas and Finot [20] modify Priest's division algorithm by using only estimates of the most significant component of the remainder r_0 and storing the less significant components of the remainder and the terms $-q_i y$ unchanged in a set that is managed with a priority queue. While the asymptotic complexity of this algorithm is better, in practical simple cases Priest's algorithm is faster due to the control overhead of the priority queue [20]. The error bound obtained with Daumas' algorithm is (using the same notations as above):

$$\left| \frac{q - x/y}{x/y} \right| < 2^{-d(p-1)} \prod_{i=0}^{d-1} (4i + 6). \tag{3.33}$$

In the worst case, Algorithm 34 requires d simple divisions, $(d-1)$ multiplications with 1 and m terms, $\sum_{i=0}^{d-1}$ additions $A_{priest}(n + 2m(i-1), 2m)$, and a final renormalization on d terms. This accounts for a total of $D_{priest}(n, m, d) = d + (d-1)M_{priest}(1, m) + \sum_{i=0}^{d-1} A_{priest}(n + 2m(i-1), 2m) + R_{priest}(d) = 27d^2m + (803m + 27n - 231)d - 830m + 213$ floating-point operations.

3.5.2 Newton-Raphson based reciprocal algorithm

The following algorithm was presented in [42], and in the extended journal version [41], where we proved that it works in the context of $\mathcal{B}$-nonoverlapping and $\mathcal{P}$-nonoverlapping floating-point expansions, but the ulp-nonoverlapping context is first considered in this manuscript.

The classical Newton-Raphson iteration for computing reciprocals [98, 17, 68, Chap. 2] is based on the general Newton-Raphson iteration for computing the roots of a given function f, which is:

$$r_{n+1} = r_n - \frac{f(r_n)}{f'(r_n)}. \tag{3.34}$$

When r_0 is close to a root α, $f'(\alpha) \neq 0$, the iteration converges quadratically. For computing $\frac{1}{x}$ we choose $f(r) = \frac{1}{r} - x$, which gives

$$r_{n+1} = r_n(2 - xr_n). \tag{3.35}$$

The iteration converges to $\frac{1}{x}$ for all $r_0 \in (0, \frac{2}{x})$. However, taking any point in $(0, \frac{2}{x})$ as the starting point r_0 would be a poor choice. A much better choice is to choose r_0 equal to a floating-point number very close to $\frac{1}{x}$. This only requires one floating-point division. The quadratic convergence of (3.35) is deduced from $r_{n+1} - \frac{1}{x} = -x(r_n - \frac{1}{x})^2$. This iteration is *self-correcting* because rounding errors do not modify the limit value.

While iteration (3.35) is well known, in Algorithm 35 we use an adaptation for computing reciprocals of floating-point expansions, with "truncated" operations, and we prove a tight error bound on the result in Theorem 3.5.2.

Algorithm 35 – Reciprocal$(x_0, x_1, \ldots, x_{2^k-1}, 2^q)$.

1: $r_0 = \mathrm{RN}(1/x_0)$
2: **for** $i \leftarrow 0$ **to** $q - 1$ **do**
3: $\quad \hat{v}[0 : 2^{i+1} - 1] \leftarrow$ Multiplication_accurate$(r[0 : 2^i - 1],\ x[0 : 2^{i+1} - 1],\ 2^{i+1})$ //using Alg. 31
4: $\quad \hat{w}[0 : 2^{i+1} - 1] \leftarrow$ Renormalize$(-\hat{v}[0 : 2^{i+1} - 1],\ 2.0,\ 2^{i+1})$ //using Alg. 24
5: $\quad r[0 : 2^{i+1} - 1] \leftarrow$ Multiplication_accurate$(r[0 : 2^i - 1],\ \hat{w}[0 : 2^{i+1} - 1],\ 2^{i+1})$
6: **end for**
7: **return** $r_0, r_1, \ldots, r_{2^q-1}$

Theorem 3.5.2. *Let x be an* ulp*-nonoverlapping floating-point expansions, with $n = 2^k$ terms and $q \geq 0$, an integer parameter such that 2^q is the required number of output terms. Assume $p \geq 6$ and $q \leq \frac{p}{2} - 2$, which always holds in practice. Provided that no underflow/overflow occurs during the calculations, Algorithm 35 computes an approximation of $\frac{1}{x}$ as $r = r_0 + \cdots + r_{2^q-1}$, an* ulp*-nonoverlapping floating-point expansion that satisfies:*

$$\left| r - \frac{1}{x} \right| \leq \frac{2^{-2^q(p-4)-2}}{|x|\,(1 - 2^{-p+1})}. \tag{3.36}$$

Property 3.5.3. *Consider an* ulp*-nonoverlapping expansion $u = u_0 + u_1 + \cdots + u_k$ with $k + 1 > 0$. Denote $u^{(i)} = u_0 + u_1 + \cdots + u_i, i \geq 0$, i.e., "a truncation" of u to $i + 1$ terms. The following inequalities hold for $0 \leq i \leq k$:*

$$|u_i| \leq 2^{-i(p-1)}\,|u_0|\,, \tag{3.37}$$

$$\left| u - u^{(i)} \right| \leq 2^{-i(p-1)}\,|u|\,\frac{\eta}{1-\eta}, \tag{3.38}$$

$$\left(1 - 2^{-i(p-1)}\frac{\eta}{1-\eta}\right)|u| \leq \left|u^{(i)}\right| \leq \left(1 + 2^{-i(p-1)}\frac{\eta}{1-\eta}\right)|u|\,, \tag{3.39}$$

$$\left| \frac{1}{u} - \frac{1}{u_0} \right| \leq \frac{1}{|u|}\eta, \tag{3.40}$$

where

$$\eta = \sum_{j=0}^{\infty} 2^{(-j-1)(p-1)} = \frac{2^{-(p-1)}}{1-2^{-(p-1)}} = \frac{1}{2^{p-1}-1}.$$

Proof. By definition of an ulp-nonoverlapping expansion and since for any normal binary floating-point number u_i, $\mathrm{ulp}(u_i) \leq 2^{-p+1}\,|u_i|$ we have $|u_i| \leq \mathrm{ulp}(u_{i-1}) \leq 2^{-p+1}\,|u_{i-1}|$ and (3.37) follows by induction.

Consequently we have $|u-u_0| = |u_1+u_2+\cdots+u_k| \leq 2^{-(p-1)}\,|u_0| + 2^{-2(p-1)}\,|u_0| + \cdots + 2^{-k(p-1)}\,|u_0| \leq |u_0|\,\eta$. One easily observes that u and u_0 have the same sign. One possible proof is by noticing that $1-\eta > 0$ and $-|u_0|\,\eta \leq u-u_0 \leq |u_0|\,\eta$. Suppose $u_0 > 0$, then $-u_0\eta \leq u-u_0 \leq u_0\eta$, and hence $u_0(1-\eta) \leq u \leq u_0(1+\eta)$ which implies $u > 0$. The case $u_0 < 0$ is similar. It follows that

$$\frac{|u|}{1+\eta} \leq |u_0| \leq \frac{|u|}{1-\eta}. \tag{3.41}$$

For (3.38) we use (3.41) together with:

$$\left|u-u^{(i)}\right| \leq \sum_{j=0}^{\infty} 2^{(-i-j-1)(p-1)}\,|u_0| \leq 2^{-i(p-1)}\eta\,|u_0|\,,$$

and (3.39) is a simple consequence of (3.38). Similarly, (3.40) follows from $\left|\frac{1}{u}-\frac{1}{u_0}\right| = \frac{1}{|u|}\left|\frac{u_0-u}{u_0}\right| \leq \frac{1}{|u|}\eta$. □

Remark 3.5.4. *Note that given an expansion u which satisfies the properties listed in Proposition 3.2.6, one can obtain its* truncation *$u^{(i)}$, with $i+1$ terms, by applying the renormalization Algorithm 24:*

$$u^{(i)} \leftarrow \textit{Renormalize}(u, i+1).$$

Proof. (of Theorem 3.5.2) Let $f_i = 2^{i+1}-1$ and $x^{(f_i)} = x_0+x_1+\cdots+x_{f_i}$, i.e., a "truncation" of x to f_i+1 terms, with $0 \leq i$.

For computing $\frac{1}{x}$ we use the Newton-Raphson iteration: $r_0 = \mathrm{RN}(\frac{1}{x_0})$, $r_{i+1} = r_i(2 - x^{(f_i)}r_i)), i \geq 0$ by truncating each operation involving floating-point expansions in the following way:

- let $v_i := x^{(f_i)} \cdot r_i$ be the exact product; we compute a truncation $\hat{v}_i$ with Algorithm 31, such that it has 2^{i+1} terms and from Theorem 3.4.4 and eq. (3.41) it satisfies:

$$\begin{aligned}
|v_i - \hat{v}_i| &\leq \left|x_0^{(f_i)}r_0\right| 2^{-(p-1)2^{i+1}} \left[1 + (2^{i+1}+1)2^{-p} + 2^{-(p-1)}\left(\frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{2^i-2}{1-2^{-(p-1)}}\right)\right] \\
&\leq \left|x^{(f_i)}r_i\right| 2^{-(p-1)2^{i+1}} \left(1 + \frac{2^i(2^{-(p-1)}+\eta)+2^{-p}}{(1-\eta)^2}\right) \\
&\leq \left|x^{(f_i)}r_i\right| \frac{2^{-(p-1)2^{i+1}}}{1-2^{-p+i+3}},
\end{aligned} \tag{3.42}$$

  as soon as $p \geq 3$ and $i < p-3$.

- let $w_i := 2 - \hat{v}_i$ be the exact result of the subtraction; we compute a truncation $\hat{w}_i$ by directly applying the renormalization Algorithm 24, such that it has 2^{i+1} terms. This is correct by the third case of Proposition 3.2.6: $\hat{v}_i$ is ulp-nonoverlapping and 2.0 plays the role of an arbitrary number;

- let $\tau_i := r_i \cdot \hat{w}_i$ be the exact product; we compute a truncation $r_{i+1} = \tau_i^{(2^{i+1}-1)}$ with Algorithm 31, such that it has 2^{i+1} terms and, similarly to (3.42) we have:

$$|\tau_i - r_{i+1}| \le |r_i \hat{w}_i| \frac{2^{-(p-1)2^{i+1}}}{1 - 2^{-p+i+3}}. \tag{3.43}$$

Let us first prove a simple upper bound for the approximation error in r_0:

$$\varepsilon_0 = \left| r_0 - \frac{1}{x} \right| \le \frac{2\eta}{|x|}. \tag{3.44}$$

Since $r_0 = \mathrm{RN}(\frac{1}{x_0})$, then $\left|r_0 - \frac{1}{x_0}\right| \le 2^{-p} \left|\frac{1}{x_0}\right|$, so, from (3.41):

$$\left| r_0 - \frac{1}{x} \right| \le 2^{-p} \left| \frac{1}{x_0} \right| + \left| \frac{1}{x} - \frac{1}{x_0} \right| \le \frac{(1+\eta)2^{-p} + \eta}{|x|} \le \frac{2\eta}{|x|}.$$

Now let us deduce an upper bound for the approximation error in r at step $i+1$, $\varepsilon_{i+1} = \left|r_{i+1} - \frac{1}{x}\right|$. For this, we will use a chain of triangular inequalities that make the transition from our "truncated" Newton error, to the "untruncated" one. Let

$$\gamma_i = 2^{-(2^{i+1}-1)(p-1)} \frac{\eta}{1-\eta} = \frac{2^{-2^{i+1}(p-1)}}{1 - 2^{-p+2}},$$

and

$$\beta_i = \frac{2^{-2^{i+1}(p-1)}}{1 - 2^{-p+i+3}}.$$

From Property 3.5.3, eq. (3.38) and eq. (3.42) and (3.43) we deduce:

$$|r_{i+1} - \tau_i| \le \beta_i \left| r_i \hat{w}_i \right|, \tag{3.45}$$

$$|w_i - \hat{w}_i| \le \gamma_i \left| w_i \right| \le \gamma_i \left| 2 - \hat{v}_i \right|, \tag{3.46}$$

$$|v_i - \hat{v}_i| \le \beta_i \left| x^{(f_i)} r_i \right|, \tag{3.47}$$

$$\left| x - x^{(f_i)} \right| \le \gamma_i \left| x \right|. \tag{3.48}$$

From (3.45) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\le |r_{i+1} - \tau_i| + \left| \tau_i - \frac{1}{x} \right| \\
&\le \beta_i \left| r_i \hat{w}_i \right| + \left| r_i \hat{w}_i - \frac{1}{x} \right| \\
&\le \beta_i \left| r_i (w_i - \hat{w}_i) \right| + \beta_i \left| r_i w_i \right| + \left| r_i \hat{w}_i - \frac{1}{x} \right| \\
&\le (1 + \beta_i) \left| r_i \right| \left| w_i - \hat{w}_i \right| + \beta_i \left| r_i w_i \right| + \left| r_i w_i - \frac{1}{x} \right|.
\end{aligned}$$

Using (3.46) and (3.47):

$$\begin{aligned}
\varepsilon_{i+1} &\le \left| r_i w_i - \frac{1}{x} \right| + (\gamma_i(1+\beta_i) + \beta_i) \left| r_i w_i \right| \\
&\le \left| r_i (2 - v_i) - \frac{1}{x} \right| + |r_i| \cdot |v_i - \hat{v}_i| + (\gamma_i(1+\beta_i) + \beta_i) \left| r_i \right| (|2 - v_i| + |v_i - \hat{v}_i|) \\
&\le \left| r_i (2 - x^{(f_i)} r_i) - \frac{1}{x} \right| + (1 + \gamma_i(1+\beta_i) + \beta_i)\beta_i \left| r_i^2 \right| \left| x^{(f_i)} \right| + (\gamma_i(1+\beta_i) + \beta_i) \left| r_i (2 - x^{(f_i)} r_i) \right|.
\end{aligned}$$

By (3.48), we have:

$$\left|r_i(2 - x^{(f_i)}r_i) - \frac{1}{x}\right| \leq |x| \left|r_i - \frac{1}{x}\right|^2 + \gamma_i |r_i|^2 |x|,$$

$$|r_i|^2 \left|x^{(f_i)}\right| \leq (1 + \gamma_i) |r_i|^2 |x|, \text{ and}$$

$$\left|r_i(2 - x^{(f_i)}r_i)\right| \leq |x| \left|r_i - \frac{1}{x}\right|^2 + \gamma_i |r_i|^2 |x| + \frac{1}{|x|}.$$

Hence we have:

$$\begin{aligned} \varepsilon_{i+1} &\leq (\gamma_i + 1)(\beta_i + 1) |x| \left|r_i - \frac{1}{x}\right|^2 \\ &+ (1 + \beta_i)(1 + \gamma_i)(\gamma_i(1 + \beta_i) + \beta_i) |r_i|^2 |x| \\ &+ (\gamma_i(1 + \beta_i) + \beta_i)\frac{1}{|x|}. \end{aligned} \tag{3.49}$$

We now prove by induction that for all $0 \leq i \leq \frac{p}{2} - 2$ and $p \geq 6$:

$$\varepsilon_i = \left|r_i - \frac{1}{x}\right| \leq \frac{2^{-2^i(p-4)-2}}{|x|\,(1 - 2^{-p+1})}. \tag{3.50}$$

For $i = 0$, this holds from (3.44). For the induction step, we have from (3.49):

$$\begin{aligned} \varepsilon_{i+1} &\leq (1 + \gamma_i)(1 + \beta_i) |x| |\varepsilon_i|^2 \\ &+ (1 + \beta_i)(1 + \gamma_i)(\gamma_i(1 + \beta_i) + \beta_i) (1 \pm \varepsilon_i |x|)^2 \frac{1}{|x|} \\ &+ (\gamma_i(1 + \beta_i) + \beta_i)\frac{1}{|x|}, \end{aligned} \tag{3.51}$$

which implies

$$\begin{aligned} |x|\,(1 - 2^{-p+1})\varepsilon_{i+1} &\leq \frac{(1 + \gamma_i)(1 + \beta_i)2^{-2^{i+1}(p-4)-4}}{1 - 2^{-p+1}} + \\ &+ (1 + \beta_i)(1 + \gamma_i)(\gamma_i(1 + \beta_i) + \beta_i)(1 \pm \varepsilon_i |x|)^2(1 - 2^{-p+1}) \\ &+ (\gamma_i(1 + \beta_i) + \beta_i)(1 - 2^{-p+1}). \end{aligned} \tag{3.52}$$

Now, we bound

$$\begin{aligned} \frac{\gamma_i}{2^{-2^{i+1}(p-4)-2}} &= \frac{2^{-2^{i+1}(p-1)}}{(1 - 2^{-p+2})2^{-2^{i+1}(p-4)-2}} \\ &= \frac{2^{-2^{i+1}(p-1-p+4)+2}}{1 - 2^{-p+2}} \\ &= \frac{2^{-3\cdot 2^{i+1}+2}}{1 - 2^{-p+2}} \\ &\leq \frac{2^{-4}}{1 - 2^{-p+2}} \\ &\leq \frac{1}{16(1 - 2^{-p+2})}, \end{aligned} \tag{3.53}$$

and

$$\begin{aligned}\gamma_i &= \frac{2^{-2^{i+1}(p-1)}}{1-2^{-p+2}} \\ &\leq \frac{2^{-2(p-1)}}{1-2^{-p+2}} \\ &\leq 2^{-2p+2+p-2}\frac{1}{2^{p-2}-1} \\ &\leq 2^{-2p+3} \leq \frac{1}{512} \;\; \forall\; p \geq 6.\end{aligned}$$

We also have:

$$\begin{aligned}\beta_i &= \frac{2^{-(p-1)2^{i+1}}}{1-2^{-p+i+3}} \\ &\leq \frac{\gamma_i(1-2^{-p+2})}{1-2^{-p+i+3}} \\ &\leq \gamma_i(1+2^{i-p+3}) \;\; \forall\; i \leq \frac{p}{2}-2 \text{ and } p \geq 6. \end{aligned} \tag{3.54}$$

Also $2^{i-p+3} \leq 2^{-p/2+1} \leq \frac{1}{4} \;\; \forall\; p \geq 6$, so

$$\beta_i \leq \gamma_i\left(1+\frac{1}{4}\right) \;\; \forall\; i \leq \frac{p}{2}-2 \text{ and } p \geq 6.$$

From (3.52) and (3.53) we have:

$$\begin{aligned}\frac{|x|\,(1-2^{-p+1})\varepsilon_{i+1}}{2^{-2^{i+1}(p-4)-2}} &\leq \frac{(1+\gamma_i)(1+\gamma_i+\frac{1}{4}\gamma_i)}{4(1-2^{-p+1})} \\ &+ \frac{1}{16}(1+\gamma_i+\frac{1}{4}\gamma_i)(1+\gamma_i)(\gamma_i+\frac{1}{4}\gamma_i+\frac{1}{4}+2)(1\pm\varepsilon_i\,|x|)^2\frac{1-2^{-p+1}}{1-2^{-p+2}} \\ &+ \frac{1}{16}(\gamma_i+\frac{1}{4}\gamma_i+\frac{1}{4}+2)\frac{1-2^{-p+1}}{1-2^{-p+2}}.\end{aligned} \tag{3.55}$$

We denote:

$$\begin{aligned}\omega_1 &= \frac{(1+\gamma_i)(1+\gamma_i+\frac{1}{4}\gamma_i)}{4(1-2^{-p+1})}; \\ \omega_2 &= \frac{1}{16}(1+\gamma_i+\frac{1}{4}\gamma_i)(1+\gamma_i)(\gamma_i+\frac{1}{4}\gamma_i+\frac{1}{4}+2)(1\pm\frac{2^{-2^i(p-4)-2}}{1-2^{-p+1}})^2\frac{1-2^{-p+1}}{1-2^{-p+2}}; \\ \omega_3 &= \frac{1}{16}(\gamma_i+\frac{1}{4}\gamma_i+\frac{1}{4}+2)\frac{1-2^{-p+1}}{1-2^{-p+2}} \\ &\text{s.t. } \frac{|x|\,(1-2^{-p+1})\varepsilon_{i+1}}{2^{-2^{i+1}(p-4)-2}} \leq \omega_1+\omega_2+\omega_3.\end{aligned} \tag{3.56}$$

We have $\frac{1}{2^{-2p+3}} \leq \frac{32}{31}$ for all $p \geq 6$ and since $\gamma_i \leq \frac{1}{512}$ it follows that:

$$\omega_1 \leq \frac{1}{3}.$$

On the same note, $(1\pm\frac{2^{-2^i(p-4)-2}}{1-2^{-p+1}})^2 \leq \frac{3}{2}$ for all $i \leq \frac{p-4}{2}$ and $p \geq 6$, and $\frac{1-2^{-p+1}}{1-2^{-p+2}} \leq \frac{31}{30}$ for all $p \geq 6$, so:

$$\omega_2 \leq \frac{1}{3}, \text{ and } \omega_3 \leq \frac{1}{3}.$$

By the above three bounds we get

$$\frac{|x|\,(1-2^{-p+1})\varepsilon_{i+1}}{2^{-2^{i+1}(p-4)-2}} \leq 1,$$

and this completes our proof. □

Our algorithm has the feature of truncating the expansions "on-the-fly" by taking into account only the significant terms of the input expansions in order to compute the result. In total we perform q iterations, and during each ith iteration we perform

- two multiplications $M_{accurate}(2^i, 2^{i+1}, 2^{i+1})$, and
- one addition using the renormalization Algorithm 24 (see Section 3.3), $R(2^{i+1}+1, 2^{i+1})$.

In the worst case, this account for a total of $I(2^k, 2^q) = \frac{52}{3}4^q + 98 \cdot 2^q + 128q - \frac{343}{3}$ floating-point operations.

Note that, in practice, we can replace the calls to the "accurate" addition and multiplication with calls to the fast "quick-and-dirty" versions. When doing this we can still get correct results (sometimes the same), but the error bound is not guaranteed anymore. By doing this we can speed up the algorithm in order to perform only $I^{fast}(2^k, 2^q) = \frac{32}{7}8^q + \frac{16}{3}4^q + 48 \cdot 2^q - 14q - \frac{1195}{21}$ floating-point operations.

3.5.3 Newton-Raphson based division algorithm

In this setting the division of two floating-point expansions is simply performed with Algorithm 35 followed by a multiplication with the numerator expansion. This can be done using either Algorithm 31, either Algorithm 33, as shown in Algorithm 36.

Algorithm 36 – Division$(x_0, \ldots, x_{n-1}, y_0, \ldots, y_{m-1}, r)$.

1: $f[0 : r-1] \leftarrow$ Reciprocal$(y[0 : m-1], r)$ //using Alg. 35
2: $d[0 : r-1] \leftarrow$ Multiplication$(x[0 : n-1],\ f[0 : r-1],\ r)$ //using Alg. 31 or Alg. 33
3: **return** $d_0, d_1, \ldots, d_{r-1}$

If the "accurate" algorithms are employed, in the worst case, Algorithm 36 performs $D(n, m, 2^q) = \frac{143}{6}4^q + 241 \cdot 2^{q-1} + 128q - \frac{145}{3}$ floating-point operations. On the other hand, if the fast algorithms are considered it perform only $D^{fast}(n, m, 2^q) = \frac{46}{7}8^q + \frac{22}{3}4^q + 55 \cdot 2^q - 14q - \frac{1300}{21}$ floating-point operations.

3.6 Square root of floating-point expansions

The families of algorithms most commonly used are exactly the same as for division, although, in the case of square root the digit-recurrence algorithm that generalizes the "paper-and-pencil" technique is typically more complicated than for division. This is the reason why a software implementation would be tedious, so there is none available. Moreover, Newton-Raphson based algorithms offer the advantage of assuring a quadratic convergence.

3.6.1 Newton-Raphson based square root algorithm

The two algorithms presented below were presented in [41] in the context of $\mathcal{B}$-nonoverlapping and $\mathcal{P}$-nonoverlapping expansions. Here we will treat the case of ulp-nonoverlapping expansions.

Starting from the general Newton-Raphson iteration (3.34), we can compute the square root in two different ways. We can look for the zeros of the function $f(r) = r^2 - x$ that leads to the so called "Heron iteration":

$$r_{n+1} = \frac{1}{2}(r_n + \frac{x}{r_n}). \tag{3.57}$$

One can easily show that if $r_0 > 0$, then r_n goes to $\sqrt{x}$. This iteration needs a division at each step, which counts as a major drawback.

To avoid performing a division at each step we can look for the positive root of the function $f(r) = \frac{1}{r^2} - x$. From here we get the iteration

$$r_{n+1} = \frac{1}{2} r_n (3 - x r_n^2). \tag{3.58}$$

This iteration converges to $\frac{1}{\sqrt{x}}$, provided that $r_0 \in (0, \frac{\sqrt{3}}{\sqrt{x}})$. The result can be multiplied by x in order to get an approximation of $\sqrt{x}$. To obtain fast, quadratic, convergence, the first point r_0 must be a close approximation to $\frac{1}{\sqrt{x}}$. In this case we still need to perform a division (by 2), but this one is much simpler. Since dividing a floating-point number by 2 can be done by multiplying it with 0.5, this being an exact operation, we can compute the division of a floating-point expansion by 2 by simply multiplying each of the terms by 0.5, separately.

As in the case of the reciprocal, in Algorithm 37 we use an adaption of iteration (3.58), using "truncated" algorithms.

Algorithm 37 – Reciprocal_SquareRoot$(x_0, x_1, \ldots, x_{2^k-1}, 2^q)$.

1: $r_0 = \mathrm{RN}(1/\sqrt{x_0})$
2: **for** $i \leftarrow 0$ **to** $q - 1$ **do**
3: $\quad \hat{v}[0 : 2^{i+1} - 1] \leftarrow$ Multiplication_accurate$(r[0 : 2^i - 1],\ x[0 : 2^{i+1} - 1],\ 2^{i+1})$ //using Alg. 31
4: $\quad \hat{w}[0 : 2^{i+1} - 1] \leftarrow$ Multiplication_accurate$(r[0 : 2^i - 1],\ \hat{v}[0 : 2^{i+1} - 1],\ 2^{i+1})$
5: $\quad \hat{y}[0 : 2^{i+1} - 1] \leftarrow$ Renormalize$(-\hat{w}[0 : 2^{i+1} - 1],\ 3.0,\ 2^{i+1})$ //using Alg. 24
6: $\quad \hat{z}[0 : 2^{i+1} - 1] \leftarrow$ Multiplication_accurate$(r[0 : 2^i - 1],\ \hat{y}[0 : 2^{i+1} - 1],\ 2^{i+1})$
7: $\quad r[0 : 2^{i+1} - 1] \leftarrow \hat{z}[0 : 2^{i+1} - 1] * 0.5$
8: **end for**
9: **return** $r_0, r_1, \ldots, r_{2^q-1}$

The error analysis for this algorithm follows the same principle as the one for the reciprocal algorithm. The goal is to show that the relative error decreases after every loop of the algorithm, by taking into account the truncations performed after each operation. The strategy is to make the exact Newton iteration term and bound appear.

Theorem 3.6.1. *Let x be an* ulp*-nonoverlapping floating-point expansions, with $n = 2^k$ terms and $q \geq 0$, an integer parameter such that 2^q is the required number of output terms. Assume $p \geq 6$ and $q \leq \frac{p}{2} - 2$, which always holds in practice. Provided that no underflow/overflow occurs during the calculations, Algorithm 38 computes an approximation of $\frac{1}{\sqrt{x}}$ as $r = r_0 + \cdots + r_{2^q-1}$, an* ulp*-nonoverlapping floating-point expansion that satisfies:*

$$\left| r - \frac{1}{\sqrt{x}} \right| \leq \frac{2^{-2^q(p-4)-1}}{\sqrt{x}(1 - 2^{-p+1})}. \tag{3.59}$$

Proof. In Property 3.5.3 we gave and proved some properties of ulp-nonoverlapping floating-point expansions. To those we are going to add a new one:

$$\left| \frac{1}{\sqrt{u}} - \frac{1}{\sqrt{u_0}} \right| \leq \frac{1}{\sqrt{u}} \eta. \tag{3.60}$$

It can be seen that $\left|\frac{1}{\sqrt{u}} - \frac{1}{\sqrt{u_0}}\right| = \frac{1}{\sqrt{u}}\left|1 - \frac{\sqrt{u}}{\sqrt{u_0}}\right|$. By using Property 3.5.3 eq. (3.41), the fact that u and u_0 have the same sign, and the fact that the square root is an increasing function, we have: $\left|1 - \frac{\sqrt{u}}{\sqrt{u_0}}\right| \leq 1 - \sqrt{\frac{1}{1+\eta}} \leq \eta$, which proves the property.

We use the same notation as for the reciprocal: $f_i = 2^{i+1} - 1$ and $x^{(f_i)} = x_0 + x_1 + \cdots + x_{f_i}$ i.e. a "truncation" of x to $f_i + 1$ terms, with $0 \leq i$. For computing $\frac{1}{\sqrt{a}}$ we use the Newton iteration: $r_0 = \mathrm{RN}(\frac{1}{\sqrt{x_0}})$, $r_{i+1} = \frac{1}{2}r_n(3 - x^{(f_i)}r_n^2)$, $i \geq 0$ by truncating each operation involving floating-point expansions in the following way:

- let $v_i := x^{(f_i)} \cdot r_i$ be the exact product; we compute a truncation $\hat{v}_i$ with Algorithm 31, such that it has 2^{i+1} terms and from Theorem 3.4.4 and (3.41) it satisfies:

$$|v_i - \hat{v}_i| \leq \left|x^{(f_i)}r_i\right| \frac{2^{-(p-1)2^{i+1}}}{1 - 2^{-p+i+3}}, \tag{3.61}$$

  as soon as $p \geq 3$ and $i < p - 3$.

- let $w_i := \hat{v}_i \cdot r_i$ be the exact product; we compute a truncation $\hat{w}_i$ with 2^{i+1} terms that, similarly to (3.61), satisfies:

$$|w_i - \hat{w}_i| \leq |\hat{v}_i r_i| \frac{2^{-(p-1)2^{i+1}}}{1 - 2^{-p+i+3}}, \tag{3.62}$$

  as soon as $p \geq 3$ and $i < p - 3$.

- let $y_i := 3 - \hat{w}_i$ be the exact result of the subtraction; we compute a truncation $\hat{y}_i$ by directly applying the renormalization Algorithm 24, such that it has 2^{i+1} terms. This is correct by the third case of Proposition 3.2.6: $\hat{w}_i$ is ulp-nonoverlapping and 3.0 plays the role of an arbitrary number;

- let $\tau_i := \frac{1}{2}\hat{y}_i \cdot r_i$ be the exact product; we compute a truncation r_{i+1} with 2^{i+1} terms similarly to (3.61) and (3.62) we have:

$$|r_{i+1} - \tau_i| \leq \left|\frac{1}{2}\hat{y}_i r_i\right| \frac{2^{-(p-1)2^{i+1}}}{1 - 2^{-p+i+3}}, \tag{3.63}$$

  as soon as $p \geq 3$ and $i < p - 3$.

We continue by first proving a simple upper bound for the approximation error in r_0:

$$\varepsilon_0 = \left|r_0 - \frac{1}{\sqrt{x}}\right| \leq \frac{1}{\sqrt{x}}\eta(3 + \eta). \tag{3.64}$$

We denote $\alpha = \mathrm{RN}(\sqrt{x_0})$, so we have $r_0 = \mathrm{RN}(\frac{1}{\alpha})$. We know that $\left|\alpha - \sqrt{x_0}\right| \leq 2^{-p}\sqrt{x_0}$ and $\left|r_0 - \frac{1}{\alpha}\right| \leq 2^{-p}\frac{1}{\alpha}$, so we obtain: $\left|\frac{1}{\alpha} - \frac{1}{\sqrt{x_0}}\right| \leq (\frac{1}{1-2^{-p}} - 1)\frac{1}{\sqrt{x_0}}$. By (3.60) we have:

$$\begin{aligned}
\left|r_0 - \frac{1}{\sqrt{x_0}}\right| &\leq \left|r_0 - \frac{1}{\alpha}\right| + \left|\frac{1}{\alpha} - \frac{1}{\sqrt{x_0}}\right| \\
&\leq 2^{-p}\frac{1}{\alpha} + \left|\frac{1}{\alpha} - \frac{1}{\sqrt{x_0}}\right| \\
&\leq 2^{-p}\frac{1}{\sqrt{x_0}} + (2^{-p} + 1)\left|\frac{1}{\alpha} - \frac{1}{\sqrt{x_0}}\right| \\
&\leq 2\eta\frac{1}{\sqrt{x_0}}.
\end{aligned}$$

From (3.60) it follows:

$$\begin{aligned}\varepsilon_0 &\leq \left|r_0 - \frac{1}{\sqrt{x_0}}\right| + \left|\frac{1}{\sqrt{x}} - \frac{1}{\sqrt{x_0}}\right| \\ &\leq 2\eta\frac{1}{\sqrt{x_0}} + \eta\frac{1}{\sqrt{x}}.\end{aligned}$$

Because $\frac{\sqrt{x}}{\sqrt{1+\eta}} \leq \sqrt{x_0}$, then $\frac{1}{\sqrt{x_0}} \leq \frac{\sqrt{1+\eta}}{\sqrt{x}} \leq \frac{1+\eta/2}{\sqrt{x}}$. We can conclude that $\varepsilon_0 \leq (2\eta(1+\frac{\eta}{2})+\eta)\frac{1}{\sqrt{x}} \leq \eta(3+\eta)\frac{1}{\sqrt{x}}$.

Before going further, let $E_i = \varepsilon_i\sqrt{x}$, such that:

$$E_0 \leq \eta(3+\eta).$$

Next we will deduce an upper bound for the approximation error in r at step $i+1$, $\varepsilon_{i+1} = \left|r_{i+1} - \frac{1}{\sqrt{x}}\right|$. For this, we use, same as in the case of the reciprocal, a chain of triangular inequalities that make the transition from our "truncated" error, to the "untruncated" one. Let

$$\gamma_i = 2^{-(2^{i+1}-1)(p-1)}\frac{\eta}{1-\eta} = \frac{2^{-2^{i+1}(p-1)}}{1-2^{-p+2}},$$

and

$$\beta_i = \frac{2^{-(p-1)2^{i+1}}}{1-2^{-p+i+3}}.$$

From Property 3.5.3, eq. (3.38) and eq. (3.61), (3.62), and (3.63) we have:

$$|r_{i+1} - \tau_i| \leq \beta_i |\tau_i| \leq \beta_i \left|\frac{1}{2}r_i\hat{y}_i\right|, \tag{3.65}$$

$$|y_i - \hat{y}_i| \leq \gamma_i |y_i| \leq \gamma_i |3 - \hat{w}_i|, \tag{3.66}$$

$$|w_i - \hat{w}_i| \leq \beta_i |w_i| \leq \beta_i |r_i\hat{v}_i|, \tag{3.67}$$

$$|v_i - \hat{v}_i| \leq \beta_i |v_i| \leq \beta_i \left|x^{(f_i)}r_i\right|, \tag{3.68}$$

$$\left|x - x^{(f_i)}\right| \leq \gamma_i |x|. \tag{3.69}$$

From (3.65) we have:

$$\begin{aligned}\varepsilon_{i+1} &\leq |r_{i+1} - \tau_i| + \left|\tau_i - \frac{1}{\sqrt{x}}\right| \\ &\leq \beta_i \left|\frac{1}{2}r_i\hat{y}_i\right| + \left|\frac{1}{2}r_i\hat{y}_i - \frac{1}{\sqrt{x}}\right|.\end{aligned}$$

Using (3.66) and (3.67):

$$\begin{aligned}\varepsilon_{i+1} &\leq (\gamma_i(1+\beta_i)+\beta_i)\left|\frac{1}{2}r_i(3-\hat{w}_i)\right| + \left|\frac{1}{2}r_i(3-\hat{w}_i) - \frac{1}{\sqrt{x}}\right| \\ &\leq (\gamma_i(1+\beta_i)+\beta_i)\left|\frac{1}{2}r_i\right|(|3-w_i| + \beta_i|w_i|) + \beta_i\left|\frac{1}{2}r_iw_i\right| + \left|\frac{1}{2}r_i(3-w_i) - \frac{1}{\sqrt{x}}\right|.\end{aligned}$$

By (3.68) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq \left|\frac{1}{2}r_i\right|\left((\gamma_i(1+\beta_i)+\beta_i)\left|3-r_i\hat{v}_i\right|+(1+\gamma_i)(1+\beta_i)\beta_i\left|r_i\hat{v}_i\right|\right)+\left|\frac{1}{2}r_i(3-r_i\hat{v}_i)-\frac{1}{\sqrt{x}}\right| \\
&\leq \left|\frac{1}{2}r_i\right|\left((\gamma_i(1+\beta_i)+\beta_i)(\left|3-r_iv_i\right|+\beta_i\left|r_iv_i\right|)+(1+\gamma_i)(1+\beta_i)^2\beta_i\left|r_iv_i\right|\right)+ \\
&\quad \beta_i\left|\frac{1}{2}r_i^2v_i\right|+\left|\frac{1}{2}r_i(3-r_iv_i)-\frac{1}{\sqrt{x}}\right| \\
&\leq \left|\frac{1}{2}r_i\right|\left((\gamma_i(1+\beta_i)+\beta_i)\left|3-r_iv_i\right|+((\gamma_i+1)(\beta_i^2+3\beta_i+1)+\gamma_i)\beta_i\left|r_iv_i\right|\right)+ \\
&\quad \beta_i\left|\frac{1}{2}r_i^2v_i\right|+\left|\frac{1}{2}r_i(3-r_iv_i)-\frac{1}{\sqrt{x}}\right|
\end{aligned}$$

From (3.69) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq \left|\frac{1}{2}r_i\right|\left((\gamma_i(1+\beta_i)+\beta_i)\left|3-r_i^2x^{(f_i)}\right|+(\gamma_i+1)(\beta_i^2+3\beta_i+2)\beta_i\left|r_i^2x^{(f_i)}\right|\right)+ \\
&\quad +\left|\frac{1}{2}r_i(3-r_i^2x^{(f_i)})-\frac{1}{\sqrt{x}}\right| \\
&\leq (\gamma_i(1+\beta_i)+\beta_i)\left|\frac{1}{2}r_i(3-r_i^2x)\right|+\Bigg[\underbrace{(\gamma_i(\gamma_i(1+\beta_i)+\beta_i))+(\gamma_i+1)^2(\beta_i^2+3\beta_i+2)\beta_i+\gamma_i}_{(1+\gamma_i)(1+\beta_i)(\gamma_i(1+\beta_i)^2+\beta_i^2+2\beta_i)}\Bigg]\cdot \\
&\quad \cdot\left|\frac{1}{2}r_i^3x\right|+\left|\frac{1}{2}r_i(3-r_i^2x)-\frac{1}{\sqrt{x}}\right|.
\end{aligned}$$

Hence we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq (1+\gamma_i(1+\beta_i)+\beta_i)\left|r_{i+1}-\frac{1}{\sqrt{x}}\right| \\
&\quad +\left[(1+\gamma_i)(1+\beta_i)(\gamma_i(1+\beta_i)^2+\beta_i^2+2\beta_i)\right]\left|\frac{1}{2}r_i^3x\right|+ \\
&\quad +(\gamma_i(1+\beta_i)+\beta_i)\frac{1}{\sqrt{x}}.
\end{aligned} \tag{3.70}$$

By using the quadratic convergence of the sequence we can say that:

$$\left|r_{i+1}-\frac{1}{\sqrt{x}}\right|=\frac{1}{2}\sqrt{x}(r_i\sqrt{x}+2)\left|r_i-\frac{1}{\sqrt{x}}\right|^2. \tag{3.71}$$

We now prove by induction that for all $i \geq 0$ $\varepsilon_i = \left|r_i - \frac{1}{\sqrt{x}}\right|$ respects the imposed bound.
We know that $|r_i\sqrt{x}| \leq \varepsilon_i\sqrt{x}+1$ and $\left|r_i^3x\right| \leq \frac{(\varepsilon_i\sqrt{x}+1)^3}{\sqrt{x}}$ and from (3.70) we have:

$$\begin{aligned}
\varepsilon_{i+1} &\leq \frac{1}{2}(1+\gamma_i)(1+\beta_i)\sqrt{x}(\varepsilon_i\sqrt{x}+3)\varepsilon_i^2 \\
&\quad +\frac{1}{2}(1+\gamma_i)(1+\beta_i)(\gamma_i(1+\beta_i)^2+\beta_i^2+2\beta_i)\frac{(\varepsilon_i\sqrt{x}+1)^3}{\sqrt{x}} \\
&\quad +(\gamma_i(1+\beta_i)+\beta_i)\frac{1}{\sqrt{x}}.
\end{aligned} \tag{3.72}$$

Using the notation $E_i = \varepsilon_i\sqrt{x}$ we can transform (3.72) in an equation independent of x:

$$\begin{aligned} E_{i+1} \quad \leq \quad & \frac{1}{2}(1+\gamma_i)(1+\beta_i)(E_i+3)E_i^2 \\ & +\frac{1}{2}(1+\gamma_i)(1+\beta_i)(\gamma_i(1+\beta_i)^2+\beta_i^2+2\beta_i))(E_i+1)^3 \\ & +(\gamma_i(1+\beta_i)+\beta_i)). \end{aligned}$$

For the last part of the proof we denote by f a function that writes the previous inequality as: $E_{i+1} \leq f(E_i, i)$. We want to show that $\forall i \in \mathbb{N}, E_i \leq \frac{2^{-2^i(p-4)-1}}{1-2^{-p+1}}$ so we will define $\text{ind}(i) = \frac{2^{-2^i(p-4)-1}}{1-2^{-p+1}}$.

For $i = 0$ we verify that $E_0 \leq \text{ind}(0)$ for $p \geq 3$.

For $i \geq 1$ by induction:

- we first prove, similarly to (3.54) that $\beta_i \leq \gamma_i(1 + 2^{i-p+3}$ when $p \geq 6$ and $i \leq \frac{p}{2} - 2$;
- for $i = 1$ we can prove by using computer algebra and the above inequality that $E_1 \leq \text{ind}(1)$, for $p \geq 6$;
- then, it is easily shown (by using the definition of a decreasing function and computation for example) that the function $i \mapsto \frac{f(\text{ind}(i),i)}{\text{ind}(i+1)}$ is decreasing and it's value in 1 is < 1 for $p \geq 6$ and $i \leq \frac{p}{2} - 2$. So, $\frac{f(\text{ind}(i),i)}{\text{ind}(i+1)} \leq 1$, for $i \leq \frac{p}{2} - 2$;
- suppose that $E_i \leq \text{ind}(i)$, we have $\frac{E_{i+1}}{\text{ind}(i+1)} \leq \frac{f(\text{ind}(i),i)}{\text{ind}(i+1)} \leq 1$ which concludes the induction, for $p \geq 6$ and $i \leq \frac{p}{2} - 2$.

At last we find the final inequality with $i = q$. □

In the QD library, for the square root computation, also the Newton iteration is used. Although they use the same function as we do, they use the iteration under the form: $r_{i+1} = r_i + \frac{1}{2}r_i(1 - xr_i^2)$, which from a mathematical point of view is the same, but it requires a different implementation.

During the ith iteration we perform:

- 3 multiplications $M_{accurate}(2^i, 2^{i+1}, 2^{i+1})$,
- one addition using the renormalization Algorithm 24 (see Section 3.3), $R(2^{i+1}+1, 2^{i+1})$, and
- 2^{i+1} multiplications by 0.5.

In the worst case, this account for a total of $I_{sqrt}(2^k, 2^q) = 26 \cdot 4^q + 139 \cdot 2^q + 194q - 163$ floating-point operations.

Note that, in practice, we can replace the calls to the "accurate" addition and multiplication with calls to the fast "quick-and-dirty" algorithms. When doing this we can still get correct results (sometimes the same), but the error bound is not guaranteed anymore. By doing this we can speed up the algorithm in order to perform only $I_{sqrt}^{fast}(2^k, 2^q) = \frac{48}{7}8^q 8 \cdot 4^q + 64 \cdot 2^q - 19q - \frac{538}{7}$ floating-point operations.

We obtain the square root of an expansion by simply multiplying the result obtained from Algorithm 37 by x, the input expansion, as shown in Algorithm 38. If the "accurate" algorithms are employed, in order to compute $sqrt(x)$, we perform $S(2^k, 2^q) = \frac{65}{2}4^q + 323 \cdot 2^{q-1} + 194q - 97$ floating-point operations. On the other hand, if the fast algorithms are considered we perform only $S^{fast}(2^k, 2^q) = \frac{62}{7}8^q + 10 \cdot 4^q + 71 \cdot 2^q - 19q - \frac{573}{7}$ floating-point operations.

Algorithm 38 – SquareRoot$(x_0, x_1 \ldots, x_{2^k-1}, 2^q)$.

1: $f[0 : 2^q - 1] \leftarrow$ Reciprocal_SquareRoot$(x[0 : 2^k - 1], 2^q)$ //using Alg. 37
2: $r[0 : 2^q - 1] \leftarrow$ Multiplication$(x[0 : 2^k - 1],\ f[0 : 2^q - 1],\ 2^q)$ //using Alg. 31 or Alg. 33
3: **return** $r_0, r_1, \ldots, r_{2^q-1}$

Heron iteration algorithm. The same type of proof as above can be applied for the algorithm using the "Heron iteration" (3.57) and the same type of truncations. In this case (Algorithm 39) we obtain a slightly larger error bound, given in Theorem 3.6.2.

Algorithm 39 – SquareRoot_Heron$(x_0, x_1 \ldots, x_{2^k-1}, 2^q)$.

1: $r_0 = \mathrm{RN}(\sqrt{x_0})$
2: **for** $i \leftarrow 0$ **to** $q-1$ **do**
3: $\quad \hat{v}[0:2^{i+1}-1] \leftarrow$ Division$(x[0:2^{i+1}-1],\ r[0:2^i-1],\ 2^{i+1})$ //using Alg. 35
4: $\quad \hat{w}[0:2^{i+1}-1] \leftarrow$ Addition_accurate$(r[0:2^i-1],\ \hat{v}[0:2^{i+1}-1],\ 2^{i+1})$ //using Alg. 27
5: $\quad r[0:2^{i+1}-1] \leftarrow \hat{w}[0:2^{i+1}-1] * 0.5$
6: **end for**
7: **return** $r_0, r_1, \ldots, r_{2^q-1}$

Theorem 3.6.2. *Let x be an* ulp*-nonoverlapping floating-point expansions, with $n = 2^k$ terms and $q \geq 0$, an integer parameter such that 2^q is the required number of output terms. Assume $p \geq 6$ and $q \leq \frac{p}{2} - 2$, which always holds in practice. Provided that no underflow/overflow occurs during the calculations, Algorithm 39 computes an approximation of $\frac{1}{\sqrt{x}}$ as $r = r_0 + \cdots + r_{2^q-1}$, an* ulp*-nonoverlapping floating-point expansion that satisfies:*

$$\left|r - \sqrt{x}\right| \leq 3\sqrt{x} \cdot \frac{2^{-2^q(p-4)-2}}{1 - 2^{-p+1}}. \tag{3.73}$$

During each iteration i we perform:

- one division using Algorithm 36, $D(2^{i+1}, 2^i, 2^{i+1})$.
- one addition using Algorithm 27, $A_{accurate}(2^i, 2^{i+1}, 2^{i+1})$, and
- 2^{i+1} multiplications by 0.5.

This account for a total of $S_{heron}(2^k, 2^q) = 64 \cdot q^2 + \frac{286}{9}4^q + 287 \cdot 2^q + \frac{5}{3}q - \frac{2860}{9}$ floating-point operations.

The same as before, in practice, we can replace the calls to the "accurate" algorithms with calls to the fast "quick-and-dirty" ones, by losing the guaranty of the error bound. This means we will perform only $S^{fast}_{heron}(2^k, 2^q) = -7 \cdot q^2 + \frac{368}{49}8^q + \frac{124}{9}4^q + 134 \cdot 2^q - \frac{1552}{21}q - \frac{68041}{441}$ floating-point operations.

3.7 Comparison and discussion

Choosing which arithmetic algorithm to use frequently depends on a compromise between accuracy, speed, and safety. In this section we compare the algorithms presented above, so that the user can get a flavor of how they behave in practice.

In Table 3.1 we give some approximates of the relative error bounds for each algorithm, considering underlying binary64 ($p = 53$). We present the values for input and output expansions of size $n = 4, 8, 16$, and 32.

Table 3.1 – Orders of magnitude of the the relative error bounds.

	n		4	8	16	32
Addition	Alg. 26		$1.5 \cdot 10^{-64}$	$2 \cdot 10^{-128}$	$5 \cdot 10^{-256}$	$2 \cdot 10^{-511}$
	Alg. 27		10^{-62}	$2 \cdot 10^{-125}$	$1.5 \cdot 10^{-250}$	$5 \cdot 10^{-501}$
	Alg. 29		$5 \cdot 10^{-62}$	$1.4 \cdot 10^{-124}$	$8 \cdot 10^{-250}$	$2 \cdot 10^{-500}$
Multiplication	Alg. 30		$1.5 \cdot 10^{-64}$	$2 \cdot 10^{-128}$	$5 \cdot 10^{-256}$	$2 \cdot 10^{-511}$
	Alg. 31		$2 \cdot 10^{-63}$	$5 \cdot 10^{-126}$	$3 \cdot 10^{-251}$	$1.2 \cdot 10^{-501}$
	Alg. 33		$2 \cdot 10^{-63}$	$2 \cdot 10^{-126}$	$6 \cdot 10^{-251}$	$5 \cdot 10^{-465}$
Division	Alg. 34	eq. (3.32)	0.25	$1.5 \cdot 10^{-2}$	$1.2 \cdot 10^{-4}$	$3 \cdot 10^{-9}$
		eq. (3.33)	$3 \cdot 10^{-59}$	$5 \cdot 10^{-116}$	$1.4 \cdot 10^{-227}$	$3 \cdot 10^{-446}$
Reciprocal	Alg. 35		$2 \cdot 10^{-60}$	$2 \cdot 10^{-119}$	$2 \cdot 10^{-237}$	$2 \cdot 10^{-473}$
Square root	Alg. 37		$4 \cdot 10^{-60}$	$4 \cdot 10^{-119}$	$4 \cdot 10^{-237}$	$4 \cdot 10^{-473}$
	Alg. 39		$7 \cdot 10^{-60}$	$7 \cdot 10^{-119}$	$7 \cdot 10^{-237}$	$7 \cdot 10^{-473}$

In order to assess the performance of the algorithms we first look at the operation count, followed by the real time execution performance. In Table 3.2 we give the worst case floating-point operation count for all the algorithms, using the notations already introduced. We consider that the input and output expansions have the same size n. We highlight with red, for each entry of n, the lowest operation count in each group of algorithms that perform the same operation. One can see that as the expansion size increases, the "quick-and-dirty" algorithms are not optimal from this point of view.

Table 3.2 – Effective values of the worst case floating-point operations count.

	n		3	4	6	8	12	16
Renormalization	Alg. 23	R_{priest}	40	60	100	140	220	300
	Alg. 24	R^{fast}	10	17	31	45	73	101
		R	16	26	46	66	106	146
	Alg. 25	R_{rand}	22	44	106	192	436	776
Addition	Alg. 26	A_{priest}	143	197	305	413	629	845
	Alg. 27	$A_{accurate}$	52	74	118	162	250	338
	Alg. 29	A_{quick}	82	135	227	467	991	1,707
		A^{fast}_{quick}	55	87	169	275	559	939
Multiplication	Alg. 30	M_{priest}	8,217	16,212	43,002	87,432	244,764	519,312
	Alg. 31	$M_{accurate}$	192	260	441	668	1,284	2,102
	Alg. 33	M_{quick}	117	234	654	1,402	4,266	9,594
		$M_{quick}[fma]$	115	231	649	1,395	4,255	9,579
		M^{fast}_{quick}	90	186	546	1,210	3,834	8,826
		$M^{fast}_{quick}[fma]$	88	183	541	1,203	3,823	8,811
Reciprocal	Alg. 35	I	847	847	2,259	2,259	6,619	6,619
		I^{fast}	485	485	2,967	2,967	20,745	20,745
Division	Alg. 34	D_{priest}	5,229	10,977	29,559	58,669	153,657	306,309
	Alg. 36	D	1,107	1,107	2,927	2,927	8,721	8,721
		D^{fast}	668	668	4,170	4,170	29,556	29,556
Reciprocal of square root	Alg. 37	I_{sqrt}	1,251	1,251	3,339	3,339	9,817	9,817
		I^{fast}_{sqrt}	708	708	4,401	4,401	31,006	31,006
Square root	Alg. 38	S	1,511	1,511	4,007	4,007	11,919	11,919
		S^{fast}	891	891	5,604	5,604	39,817	39,817
	Alg. 39	S_{heron}	1,646	1,646	4,743	4,743	13,818	13,818
		S^{fast}_{heron}	907	907	5,360	5,360	35,871	35,871

In what follows we present some performance measurements obtained for our underlying binary64 implementation, on the CPU and GPU[5] described in Section 1.4.1. The CPU implementation's performance is compared with that of MPFR. The CPU performance is given in Mop/s (Mega operations per second), and the GPU performance in Kop/s (Kilo operations per second).

In Table 3.3 and 3.4 we assess the addition performance on CPU and GPU, respectively. Table 3.5 and 3.6 refer to the multiplication performance. In Table 3.7 and 3.8 we present the reciprocal performance, followed by the division one in Table 3.9 and 3.10. We finish by comparing the squaring algorithms in Table 3.11 and 3.12.

In each table, for each size entry, we highlight with red the algorithm from our library that performs the best. In the CPU performance tables, in case the MPFR library performs better, we highlight that entry with green.

Table 3.3 – CPU performance in Mop/s for the addition algorithms.

	CAMPARY			MPFR
n, m, r	Alg. 27	Alg. 29	Fast Alg. 29	
3, 3, 3	34	27.3	40	31.5
3, 1, 3	75.3	75.3	75.3	22.2
4, 4, 4	19.3	16.9	23.2	28.2
4, 2, 4	33.5	19.9	28.1	20.2
4, 1, 4	59.2	59.2	59.2	20.7
8, 8, 8	8.15	4.48	7.07	24.5
8, 4, 8	11.2	5	7.36	19.7
8, 2, 8	15	6.32	10.2	19.1
16, 16, 16	3.82	1.12	1.73	17.1
16, 8, 16	5.46	1.2	2.5	15.3

Table 3.4 – GPU performance in Kop/s for the addition algorithms.

	CAMPARY		
n, m, r	Alg. 27	Alg. 29	Fast Alg. 29
3, 3, 3	396	310	765
3, 1, 3	1,244	1,244	1,244
4, 4, 4	270	189	566
4, 2, 4	365	217	752
4, 1, 4	972	972	972
8, 8, 8	135	60.8	274
8, 4, 8	170	72.2	162
8, 2, 8	205	78.9	235
16, 16, 16	55.8	17.4	35.3
16, 8, 16	81.3	22.1	46

5. The values reflect the performance obtained on GPU using only one execution thread.

Table 3.5 – CPU performance in Mop/s for the multiplication algorithms.

	CAMPARY			MPFR
n, m, r	Alg. 31	Alg. 33	Fast Alg. 33	
3, 3, 3	22.6	22	30.2	14.1
3, 1, 3	33.9	35.5	51.5	17.9
4, 4, 4	10.7	10.8	13.4	12.8
4, 2, 4	12.8	13.4	17.2	16.1
4, 1, 4	20.5	20.8	28.7	16.7
8, 8, 8	2.05	1.55	1.94	7.72
8, 4, 8	2.8	1.82	2.16	10.7
8, 2, 8	4.23	2.83	3.53	13.83
16, 16, 16	0.62	0.24	0.21	2.92
16, 8, 16	0.83	0.25	0.24	5.12

Table 3.6 – GPU performance in Kop/s for the multiplication algorithms.

	CAMPARY		
n, m, r	Alg. 31	Alg. 33	Fast Alg. 33
3, 3, 3	400	192	750
3, 1, 3	502	351	994
4, 4, 4	226	107	411
4, 2, 4	251	132	517
4, 1, 4	318	231	764
8, 8, 8	31.8	22.1	65.5
8, 4, 8	45.9	26.2	33.1
8, 2, 8	67.1	40.4	58.1
16, 16, 16	8.5	3.56	3.62
16, 8, 16	12.9	4.09	4.44

Table 3.7 – CPU performance in Mop/s for the reciprocal algorithms.

	CAMPARY		MPFR
$2^k, 2^q$	Alg. 35	Fast Alg. 35	
4, 2	$6 \cdot 10^4$	$7 \cdot 10^4$	12.8
4, 4	8.76	10.65	5.3
2, 4	4.34	5.18	10.2
1, 4	4.87	6.35	10.9
8, 8	1.04	0.83	3.9
4, 8	1.16	0.88	4.3
2, 8	1.29	1.03	8
16, 16	0.32	0.08	1.7
8, 16	0.36	0.08	2.5
4, 16	0.4	0.12	3

Table 3.8 – GPU performance in Kop/s for the reciprocal algorithms.

	CAMPARY	
$2^k, 2^q$	Alg. 35	Fast Alg. 35
4, 2	$2 \cdot 10^7$	$2 \cdot 10^7$
4, 4	129	282
2, 4	128	149
1, 4	129	202
8, 8	19	19.3
4, 8	24	20.2
2, 8	25.7	21.2
16, 16	4.4	1.83
8, 16	5.6	1.98
4, 16	6.2	2.45

Table 3.9 – CPU performance in Mop/s for the division algorithms.

	CAMPARY		MPFR
n, m, r	Alg. 36	Fast Alg. 36	
2, 4, 2	410	411	11.02
4, 4, 4	3.1	3.7	5.05
4, 2, 4	2.99	3.63	9.3
4, 1, 4	3.23	4.1	9.6
8, 8, 8	0.72	0.58	3.7
8, 4, 8	0.74	0.6	3.9
8, 2, 8	0.78	0.67	6.9
16, 16, 16	0.21	0.06	1.65
16, 8, 16	0.22	0.06	2.3
16, 4, 16	0.24	0.07	2.6

Table 3.10 – GPU performance in Kop/s for the division algorithms.

	CAMPARY	
n, m, r	Alg. 36	Fast Alg. 36
2, 4, 2	8310	7987
4, 4, 4	86.3	180
4, 2, 4	75.5	108
4, 1, 4	82.9	146
8, 8, 8	12.7	11.8
8, 4, 8	14.3	13.3
8, 2, 8	14.9	14.9
16, 16, 16	2.78	1.23
16, 8, 16	3.21	1.28
16, 4, 16	3.42	1.48

Table 3.11 – CPU performance in Mop/s for the squaring algorithms.

	CAMPARY						MPFR	
$2^k, 2^q$	Alg. 37	Fast Alg. 37	Alg. 38	Fast Alg. 38	Alg. 39	Fast Alg. 39	$1/\sqrt{}$	$\sqrt{}$
4, 2	28.9	28.9	13.3	14.3	43	163	4	6.6
4, 4	3.98	4.75	2.25	2.67	2.36	2.95	2.1	4.2
2, 4	3.04	3.4	2.93	2.97	2.44	3.18	2.2	4
1, 4	3.22	3.96	3.2	3.31	2.61	3.56	2.2	4.3
8, 8	1	0.56	0.52	0.43	0.54	0.47	1.3	2.2
4, 8	0.97	0.58	0.6	0.44	0.59	0.47	1.3	2.2
2, 8	1.08	0.63	0.68	0.54	0.63	0.53	1.3	2.2
16, 16	0.85	0.06	0.16	0.047	0.15	0.061	0.7	1.15
8, 16	0.93	0.064	0.21	0.051	0.16	0.064	0.7	1.2
4, 16	1.02	0.067	0.24	0.056	0.18	0.071	0.7	1.2

Table 3.12 – GPU performance in Kop/s for the squaring algorithms.

	CAMPARY					
$2^k, 2^q$	Alg. 37	Fast Alg. 37	Alg. 38	Fast Alg. 38	Alg. 39	Fast Alg. 39
4, 2	1,185	1,235	6,875	578	635	2,857
4, 4	70.8	131	53.2	96.4	56.5	83.9
2, 4	70.8	106	58.7	87.1	56.3	89
1, 4	76.8	137	60.1	115	62.3	93.7
8, 8	12.9	9.87	9.69	8.29	8.45	9.38
4, 8	14.9	10.4	10.2	8.89	8.98	8.14
2, 8	15.1	11.9	13.1	9.65	9.67	9.18
16, 16	2.77	1.15	2.07	0.92	2.06	1.05
8, 16	3.13	1.27	2.47	0.99	2.25	1.09
4, 16	3.5	1.46	3.07	1.24	2.51	1.19

When looking at these tables, one can observe that in general the algorithms perform in accordance with their operation count. This is why, as the expansion size increases, the "accurate" algorithms are faster. However, this is not the case for addition, where Algorithm 27 has the lowest operation count for all entries, but, in practice, for expansion size up to 4, Algorithm 29 performs better. This is due to pipeline optimization.

Another strange behavior, both on CPU and on GPU, can be observed for Algorithm 37: when we compute the reciprocal square root of an input expansion of size 2 (double-double) as an output expansion of size 4, the performance is worse or the same as for computing starting from an input expansion of size 4. We suppose that this is due to compiler optimizations an the ability of the algorithms to fill up the pipeline.

As expected the performance of the reciprocal/division and square root algorithms is in accordance to the performance of the addition and multiplication algorithms used for intermediate computations.

We ask the reader to keep in mind that these values were obtained in an ideal setting, in which we performed operations using only the tested algorithm. In real life applications the performance may depend on many factors like instruction level parallelism, pipeline depth, branch prediction, memory usage, etc. We will compare CAMPARY with other existing libraries in two real applications in Chapter 5.

CHAPTER 4
Parallel Floating-point Expansions

In this chapter we explore the possibility of directly parallelizing the arithmetic algorithms. We present new data-parallel algorithms for adding and multiplying floating-point expansions specially designed for extended precision computations on GPUs.

This is a joint work with S. Collange, [1] M. Joldes, and J.-M. Muller, that was presented in *Parallel floating-point expansions for extended-precision GPU computations* [16], published in Proceedings of the 27th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2016).

We argued in the introduction (Chapter 1) that our work also focuses on GPU implementation, but the algorithms that we presented until now were sequential. In that setting we assumed we have to deal with "embarrassingly parallel" problems with compact intermediate data for which it was certainly better not to try to parallelize the arithmetic itself. However, many applications do not provide as much parallelism. Even for those that do, locality can be a problem. Increasing the precision of sequential arithmetic operations requires a corresponding increase in the amount of intermediate data to keep. Thus, parallel arithmetic algorithms are attractive not just by the extra parallelism they provide, but also by the locality improvements they enable.

With GPU oriented implementation in mind, in this chapter we deal with what we called "parallel floating-point expansions", i.e., the expansion is stored on parallel execution threads, with one term/thread. This implies that the user has to launch as many threads as the expansion size.

In Section 4.1 and Section 4.2 we present the addition and multiplication of parallel floating-point expansions in the general context of parallel architectures. We finish by comparring performance with the algorithms presented in Chapter 2 and 3 in Section 4.3.

4.1 Data-parallel addition algorithm

The first algorithm that we developed for parallel addition of floating-point expansions is Algorithm 40 below, illustrated in Figure 4.1 for the case of expansions with 3 components. For the sake of simplicity, we only present here the "input r - output r" version of the algorithm, even though the generalized version allows for different input sizes.

In this setting, all arithmetic operations including error-free transforms are performed in parallel element-wise on r-element vectors. We assume vectors can be merged and elements inside

1. Researcher in PACAP project-team, Inria Rennes, Rennes, France.

Algorithm 40 – PAddition_safe$(\mathbf{x} = (x_0, \ldots, x_{r-1}), \mathbf{y} = (y_0, \ldots, y_{r-1}), r)$.

1: $\mathbf{a} \leftarrow (x_0, 0, 0, \ldots, 0)$
2: $\mathbf{b} \leftarrow (y_0, 0, 0, \ldots, 0)$
3: $(\mathbf{s}, \mathbf{e}) \leftarrow \text{2Sum}(\mathbf{a}, \mathbf{b})$
4: **for** $i \leftarrow 1$ to r **do**
5: $\quad \mathbf{e}' \leftarrow (x_i, e_0, e_1, \ldots, e_{r-2})$ //shift right & insert x_i
6: $\quad (\mathbf{s}, \mathbf{e}) \leftarrow \text{2Sum}(\mathbf{s}, \mathbf{e}')$
7: $\quad \mathbf{e}' \leftarrow (y_i, e_0, e_1, \ldots, e_{r-2})$ //shift right & insert y_i
8: $\quad (\mathbf{s}, \mathbf{e}) \leftarrow \text{2Sum}(\mathbf{s}, \mathbf{e}')$
9: **end for**
10: **for** $i \leftarrow 1$ to $r - 2$ **do**
11: $\quad \mathbf{e}' \leftarrow (0, e_0, e_1, \ldots, e_{r-2})$
12: $\quad (\mathbf{s}, \mathbf{e}) \leftarrow \text{2Sum}(\mathbf{s}, \mathbf{e}')$
13: **end for**
14: $\mathbf{e}' \leftarrow (0, e_0, e_1, \ldots, e_{r-2})$//shift right
15: $\mathbf{s} \leftarrow \mathbf{s} + \mathbf{e}'$
16: **return** $\mathbf{s} = (s_0, s_1, \ldots, s_{r-1})$

a vector can be shuffled. As the data-parallel computations are homogeneous, larger vectors can be assembled by concatenating several expansion vectors and processing them in parallel. This allows to fill SIMD execution units by leveraging parallelism between expansions. These assumptions make the algorithms applicable to most SIMD units, including the Intel SSE/AVX instruction set extensions [24] and all recent Nvidia GPUs [77].

The main constraint of the algorithm is the fact that it requires r parallel execution threads in order for it to work and we rely on the user to launch them.

The algorithm is based on a pipelined error propagation. We start by adding the first elements of each expansion, x_0 and y_0, on the first vector component. We continue to add the rest of the elements on the first component one by one and propagating the error upwards, to the other vector components. When we run out of elements to add we continue to propagate the errors for another $r - 1$ steps by injecting 0s on the first component. In the last step of the algorithm we can use only simple addition since we are not going to propagate the errors anymore.

By using this scheme to add the two expansions we ensure that the most significant term of the output, s_0, is the sum of the inputs rounded to nearest. Moreover, the terms of the output are arranged in terms of magnitude in decreasing order, with some constraints. We will show this by proving Theorem 4.1.1.

Theorem 4.1.1. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n and m terms, respectively. Provided that no underflow/overflow occurs during the calculations, when computing their sum using Algorithm 40, the result array s with r terms satisfies $|s_i| \leq 2^{-(p-1)i+2r-1}|s_0|$, with $0 < i < n$ and*

$$|x + y - s| < 2^{-(p-1)r}(|x_0| + |y_0|)\left(2^{2r}(1 + 2^{-p}) + \frac{1}{1 - 2^{-(p-1)}}\right).$$

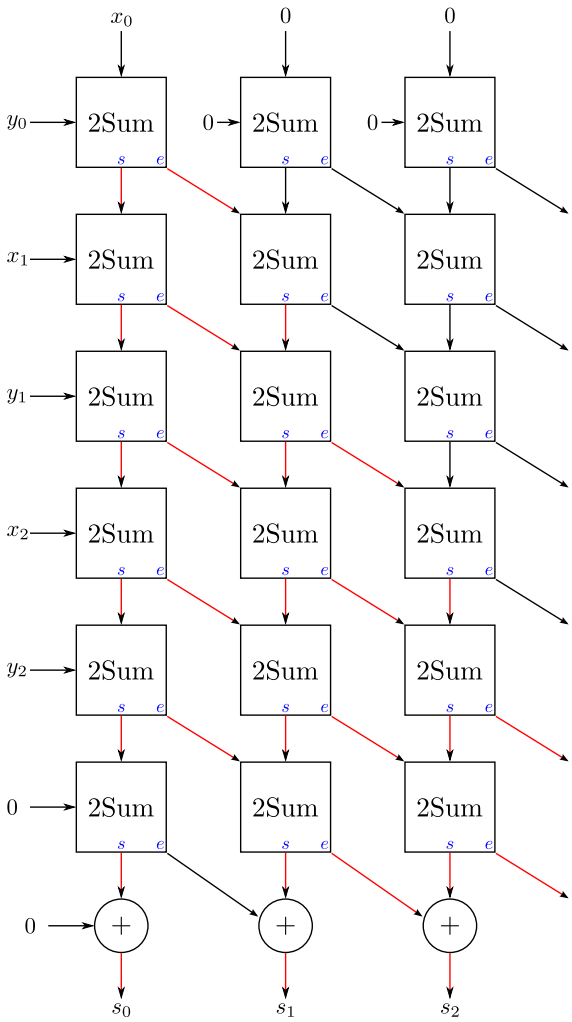


Figure 4.1 – Graphical representation of Algorithm 40 illustrated for expansions of size 3.

Before proving the above theorem we need to take a closer look to the algorithm. It is easily seen that the parallel scheme presented in Figure 4.1 can be reduced to the sequential one presented in Figure 4.2.

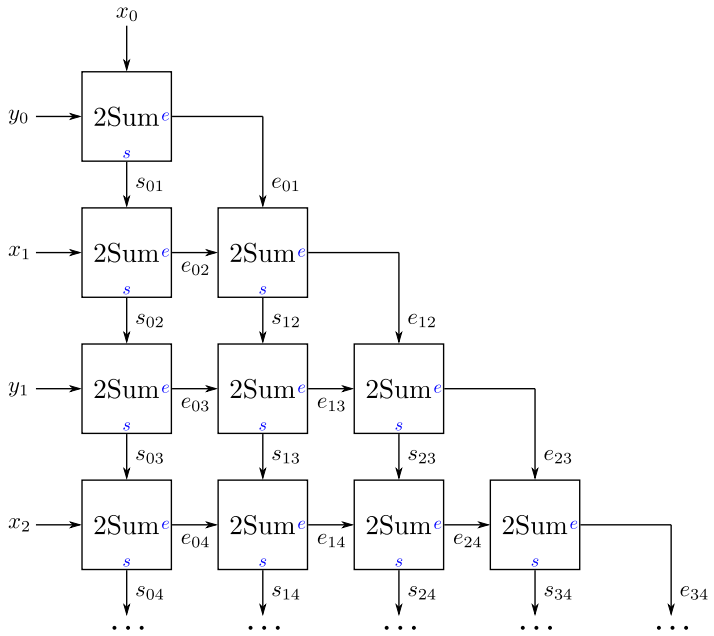


Figure 4.2 – Sequential representation of Algorithm 40.

Consider that x_0 and y_0 are of opposite signs and $\left|\frac{x_0}{2}\right| \leq |y_0| \leq |2x_0|$ (we say that there is a "Sterbenz relation" between x_0 and y_0). Then $e_{01} = 0$ and $s_{01} = x_0 + y_0$, which implies that $s_{12} = e_{02}$ and $e_{12} = 0$, and so on. In this case we end up propagating a 0 to the end of the result expansion, and we are left with the same scheme as we began with (illustrated in Figure 4.3). This means that in our analysis we can eliminate the case in which we have a "Sterbenz relation".

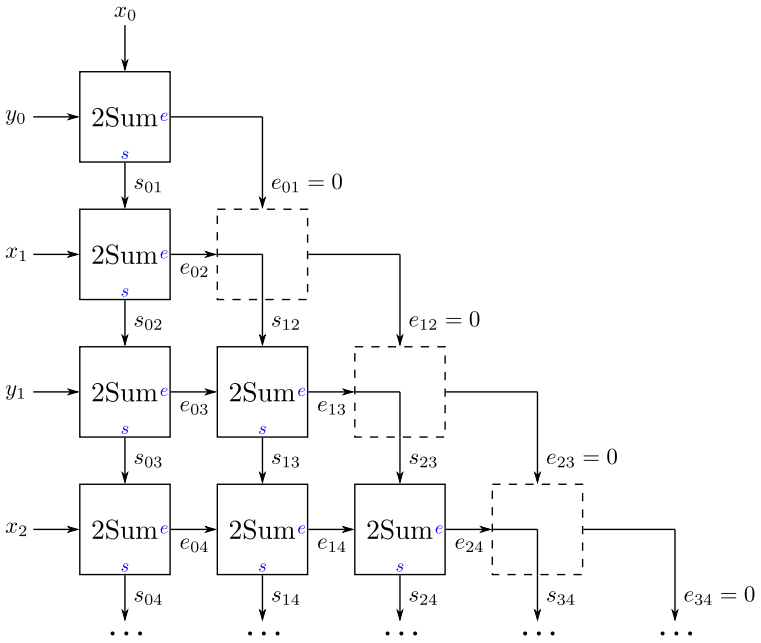


Figure 4.3 – Reduction of the sequential representation in Figure 4.2 based on the "Sterbenz relation".

We will first prove an intermediate property given in Theorem 4.1.2 that refers to only one horizontal line of the scheme, i.e., a chain of 2Sum starting from the left side and propagating the error to the right. An algorithm that performs this type of addition can be visualized as a "reverse" VecSum. For the proof we use the notations given in Figure 4.4.

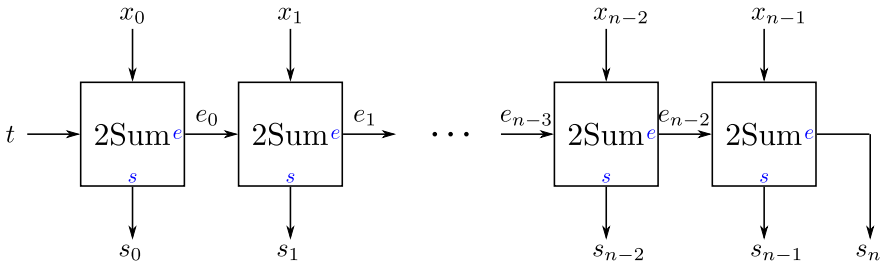


Figure 4.4 – Addition of a floating-point number to an array, starting from the left side and propagating the error. In the 2Sum calls the sum s is outputted downwards and the error e to the right.

Theorem 4.1.2. *Let $x = x_0, x_1, \dots x_{n-1}$ be an array of floating-point numbers that satisfy $|x_i| \leq 2^{-i(p-1)+\delta}|x_0|$, for some (presumably small) integer δ and $|t| \leq 2^{-p+\ell}|x_0|$, for some (presumably small) integer ℓ. If $s = (s_0, s_1, \dots, s_n)$ is the array obtained by adding t to x as shown in Figure 4.4 (from left to right, by propagating the error), then all the terms in s satisfy: $|s_i| \leq 2^{-i(p-1)+\delta+1}$, for all $0 < i \leq n$.*

Proof. From the proof of the algorithm 2Sum we know that $|e_i| \leq \frac{1}{2}\operatorname{ulp}(s_i) = 2^{-p}|s_i|$.

From

$$\begin{aligned} |s_0|(1-2^{-p}) \leq \quad |x_0+t| \quad &\leq |s_0|(1+2^{-p}) \text{ and} \\ |x_0|(1-2^{-p+\ell}) \leq \quad |x_0+t| \quad &\leq |x_0|(1+2^{-p+\ell}), \end{aligned}$$

we get

$$|s_0|\frac{1-2^{-p}}{1+2^{-p+\ell}} \leq |x_0| \leq |s_0|\frac{1+2^{-p}}{1-2^{-p+\ell}}. \tag{4.1}$$

It follows that

$$|x_1| \leq 2^{-(p-1)+\delta}|x_0| \leq 2^{-(p-1)+\delta}\frac{1+2^{-p}}{1-2^{-p+\ell}}|s_0|.$$

This gives

$$|e_0 + x_1| \leq 2^{-p}\left[1 + \frac{2^{\delta+1}(1+2^{-p})}{1-2^{-p+\ell}}\right]|s_0|\,.$$

From which we deduce

$$|s_1| \leq 2^{-p}(1+2^{-p})\left[1 + \frac{2^{\delta+1}(1+2^{-p})}{1-2^{-p+\ell}}\right]|s_0|\,. \tag{4.2}$$

We can continue, by noticing that $|e_1|$ is bounded by $2^{-p}|s_1|$, and bounding $|x_2|$ by $2^{-2(p-1)+\delta}\frac{1+2^{-p}}{1-2^{-p+\ell}}|s_0|$. This gives a bound on $|e_1 + x_2|$, and a bound on s_2 is obtained by multiplying that last bound by $(1+2^{-p})$. An easy induction finally gives:

$$|s_i| < 2^{-ip}\theta_i|s_0|, \tag{4.3}$$

with

$$\theta_i = (1+2^{-p})^i + \frac{1}{1-2^{-p+\ell}}\sum_{j=1}^{i} 2^{j+\delta}(1+2^{-p})^{i-j+2}. \tag{4.4}$$

One easily finds

$$\theta_i = (1+2^{-p})^i + \frac{2^{\delta+1}(1+2^{-p})^2}{1-2^{-p+\ell}}\left[\frac{2^i-(1+2^{-p})^i}{1-2^{-p}}\right],$$

hence,

$$\theta_i = 2^{i+\delta+1}H_i,$$

with

$$H_i = \frac{(1+2^{-p})^i}{2^{i+\delta+1}} + \frac{(1+2^{-p})^2}{1-2^{-p+\ell}}\left(\frac{1-\frac{(1+2^{-p})^i}{2^i}}{1-2^{-p}}\right).$$

We recall $u = 2^{-p}$, the roundoff error. In all practical cases $\ell \geq 2$ and $\delta \geq 0$, so that $H_i \leq G_i$, with

$$G_i = \frac{1}{2}\left(\frac{1+u}{2}\right)^i + \frac{(1+u)^2}{1-4u}\left(\frac{1-\left(\frac{1+u}{2}\right)^i}{1-u}\right).$$

We have,

$$G_i \quad = \quad 1 - \tfrac{1}{2}\left(\tfrac{1+u}{2}\right)^i + \tfrac{u(u+6)}{1-4u} - \tfrac{1}{2}\left(\tfrac{1+u}{2}\right)^i\left(\tfrac{2u(7-3u)}{(1-4u)(1-u)}\right).$$

The only positive term (after the initial "1") in that sum is $\frac{u(u+6)}{1-4u}$, which is less than $7u$ for all pertinent values of u. Hence $G_i < 1$ as soon as $2^{i+1} \leq \frac{2^p}{7}$, which occurs in all practical cases. This gives

$$|s_i| < 2^{-i(p-1)+\delta'}|s_0|,$$

with $\delta' = \delta + 1$. This concludes our proof. □

Now we are able to prove Theorem 4.1.1.

Proof. (of Theorem 4.1.1) In Algorithm 40 we use the same type of truncation as for the sequential algorithms, i.e., we take into account only the most significant r components of x and y. From Theorem 3.3.1 we know that the ignored terms satisfy, :

$$\sum_{i=r}^{n-1} x_i + \sum_{j=r}^{m-1} y_j \leq (|x_0|+|y_0|)\frac{2^{-(p-1)r}}{1-2^{-(p-1)}}.$$

From Theorem 4.1.2 we know that in the array of Figure 4.1, δ is increased by 1 at each line. For computing s we use $2r-1$ "horizontal lines", which implies that

$$|s_i| \leq 2^{-(p-1)i+2r-1}|s_0|.$$

By keeping only the first r terms of s we have an error less than

$$\underbrace{(2^{-rp+3r-1} + 2^{-(r+1)p+3r} + 2^{-(r+2)p+3r+1} + \cdots)}_{\approx 2^{-rp+3r-1} < 2^{-rp+3r}}|s_0|.$$

We can now bound the total error. We get:

$$|x+y-s| < 2^{-(p-1)r}(|x_0|+|y_0|)\left(2^{2r}(1+2^{-p}) + \frac{1}{1-2^{-(p-1)}}\right). \tag{4.5}$$

□

"Quick-and-dirty" parallel addition. We can speed up the above algorithm by using a "relaxed" version of it, that requires at most $r-1$ steps (the last step using only simple additions). The "quick-and-dirty" parallel addition, illustrated in Figure 4.5, offers a worse error bound and it does not ensure a correct result when cancellation occurs, if no renormalization algorithm is applied on the result. We advise it's use only with input floating-point expansions of the same sign and close magnitudes.

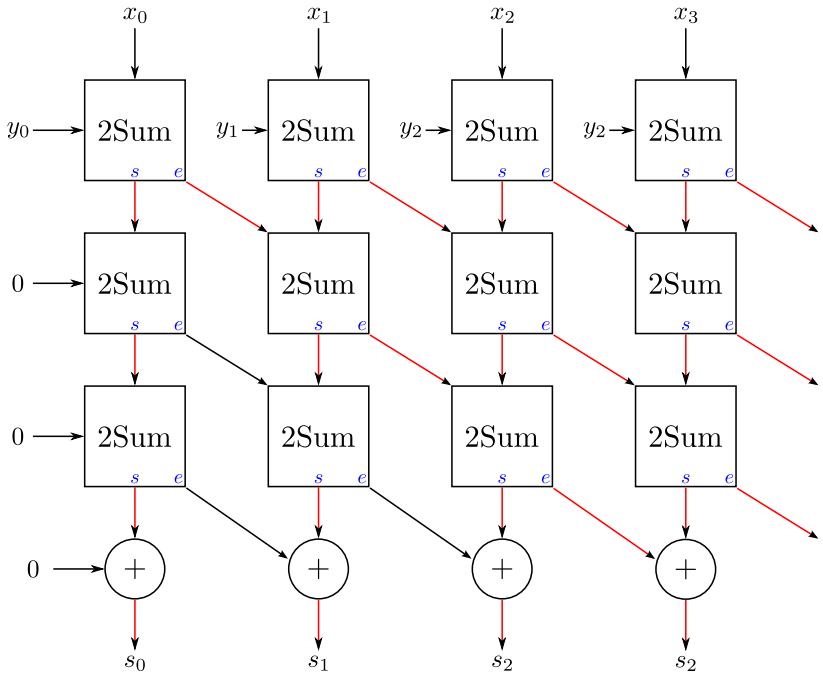


Figure 4.5 – Graphical representation of a "quick-and-dirty" version of Algorithm 40 illustrated for expansions of size 4.

4.2 Data-parallel multiplication algorithm

Algorithm 41 computes an approximation of xy, where x and y are two parallel expansions. Here we also present just the "input r - output r" version. This algorithm has the same behavior as Multiplication_quick (Section 3.4, Algorithm 33), but we do not compute the extra error correction term π_r, since we do not apply a renormalization. A graphical representation of the parallel execution would be too difficult to read, this is why in Figure 4.6 we present just an equivalent sequential execution.

We consider two parallel floating-point expansions x and y, each with r terms and we compute the r most significant floating-point components of the product $\pi = xy$. We perform the same "on-the-fly" truncation as for the sequential multiplication algorithms (see Section 3.4), by considering only the partial products for which $0 \leq i + j \leq r - 1$.

The multiplication algorithm runs as follows: at each iteration i of the for loop (lines 3-17) we compute $p + e = xy$; we add p to the result of the same order, using 2Sum, which also generates an error, e'. After that, using the two while loops (lines $9-12$ and $13-16$) we propagate the two generated errors, e and e' to the lower order results. In the last step of the algorithm, we do not use any error-free transforsm, because the errors that are supposed to be computed are going to be of order $\mathcal{O}(\varepsilon^r \Lambda)$, and we do not need to propagate them anymore.

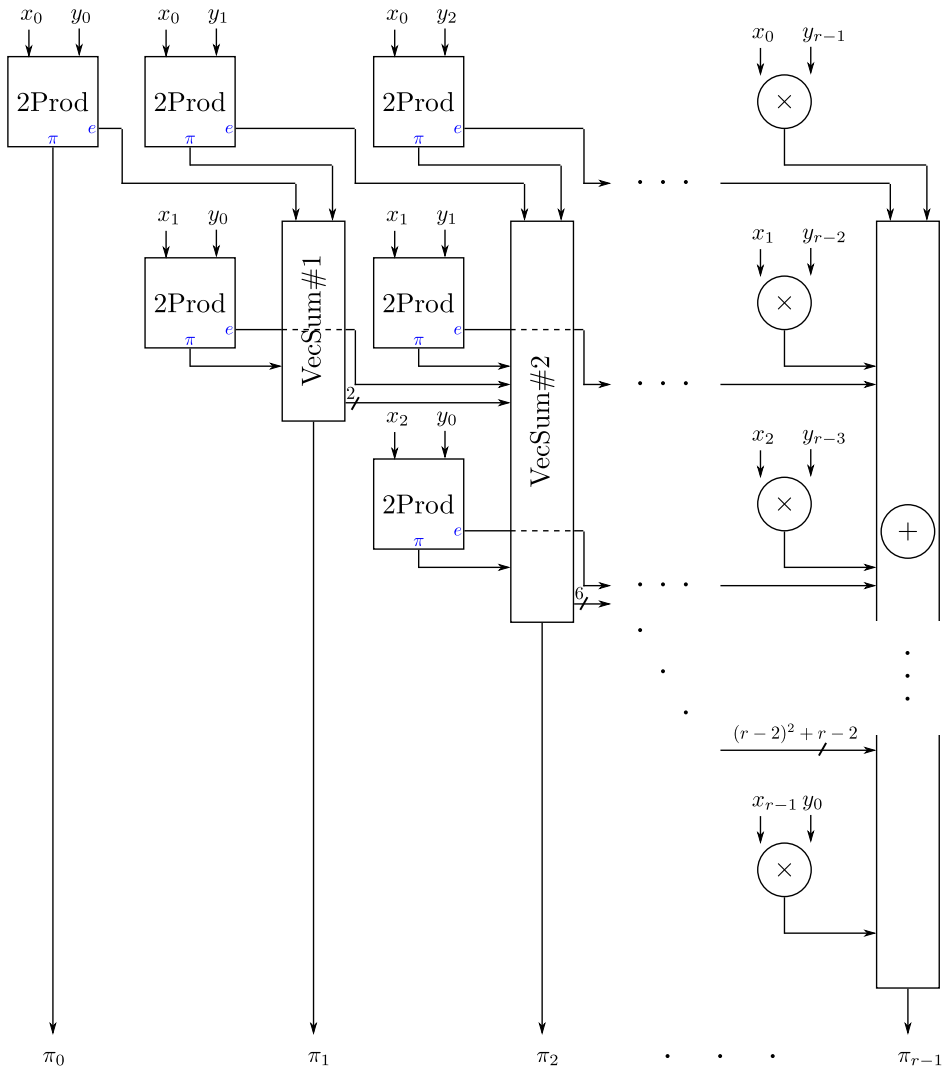


Figure 4.6 – Sequential representation of Algorithm 41.

Algorithm 41 – PMultiplication$(\mathbf{x} = (x_0, \ldots, x_{r-1}), \mathbf{y} = (y_0, \ldots, y_{r-1}), r)$.

$\mathbf{s} \leftarrow (0, \ldots, 0)$
$\pi \leftarrow (0, \ldots, 0)$
for $i \leftarrow 0$ to $r-2$ **do**
  $\mathbf{y}' \leftarrow (y_i, y_i, \ldots, y_i)$ //broadcast
  $(\mathbf{p}, \mathbf{e}) \leftarrow$ 2Prod$(\mathbf{x}, \mathbf{y}')$
  $(\mathbf{s}, \mathbf{e}') \leftarrow$ 2Sum$(\mathbf{s}, \mathbf{p})$
  $\pi_i \leftarrow s_0$ //insert into vector
  $\mathbf{s} \leftarrow (s_1, s_2, \ldots s_{r-1}, 0)$ //shift left
  while $\mathbf{e} \neq 0$ **do**
    $(\mathbf{s}, \mathbf{e}) \leftarrow$ 2Sum$(\mathbf{s}, \mathbf{e})$
    $\mathbf{e} \leftarrow (0, e_0, e_1, \ldots, e_{r-2})$ //shift right
  end while
  while $\mathbf{e}' \neq 0$ **do**
    $(\mathbf{s}, \mathbf{e}') \leftarrow$ 2Sum$(\mathbf{s}, \mathbf{e}')$
    $\mathbf{e}' \leftarrow (0, e'_0, e'_1, \ldots, e'_{r-2})$ //shift right
  end while
end for
$\mathbf{p} \leftarrow \mathbf{x} \cdot \mathbf{y}$
$\mathbf{s} \leftarrow \mathbf{s} + \mathbf{p}$
$\pi_{r-1} \leftarrow s_0$ //insert into vector
return $\pi = (\pi_0, \pi_1, \ldots, \pi_{r-1})$

Theorem 4.2.1. *Let x and y be two* ulp*-nonoverlapping floating-point expansions, with n and m terms, respectively. Assume $p \geq 8$, which always holds in practice. Provided that no underflow/overflow occurs during the calculations, when computing their product using Algorithm 41, the result array π with r terms satisfies*

$$|xy - \pi| \leq |x_0 y_0| 2^{-(p-1)r} \left(\frac{128}{127}(m+n-1) - \frac{129}{254} r + 2^{-p-r+2}(r^2 - r)(r!)^2 \right).$$

Proof. For bounding the discarded partial products we use the same reasoning as in Theorem 3.4.1 and we get:

$$\sum_{k=r}^{m+n-2} \sum_{i+j=k} x_i y_j \leq |x_0 y_0| \, 2^{-(p-1)r} \left(\underbrace{\frac{-2^{-(p-1)}}{(1 - 2^{-(p-1)})^2}}_{<0} + (m+n-r-1) \underbrace{\frac{1}{1 - 2^{-(p-1)}}}_{< \frac{128}{127} \text{ for } p \geq 8} \right).$$

For computing the error bound on the discarded errors we use the same method as in the proof of Theorem 3.4.5. Along these lines, by using simple summation for computing π_{r-1}, in the last step we neglect $r^2 - r$ terms, all less than $\Omega_r = 2^{-(r+1)p+2} \, |x_0 y_0| \, (r!)^2$.

We also have to account for the errors that occur when computing the last partial products using only simple multiplication. This means r terms less than $\frac{1}{2} \, |x_0 y_0| \, 2^{-(p-1)r}$.

When adding all these errors we get the following bound:

$$\begin{aligned} |xy - \pi| &\leq |x_0 y_0| 2^{-(p-1)r} \left(\frac{128}{127}(m+n-r-1) + 2^{-p-r+2}(r^2 - r)(r!)^2 + \frac{r}{2} \right) \\ &\leq |x_0 y_0| 2^{-(p-1)r} \left(\frac{128}{127}(m+n-1) - \frac{129}{254} r + 2^{-p-r+2}(r^2 - r)(r!)^2 \right). \end{aligned}$$

And this concludes our proof □

Unfortunately, for the multiplication algorithm we are unable to prove any constraints on the terms of the result. Even though cancellation cannot happen when multiplying two floating-point numbers, it may happen during the summation process, in which case we can get $|\pi_i| < |\pi_j|$, with $i < j$. If this happen we would have to apply a renormalization algorithm, like the ones presented in Chapter 3, Section 3.2, that would render the result ulp-nonoverlapping . However, those algorithms are highly sequential, and they would significantly decrease performance. This is why we recommend using this algorithm only if computations are known not to be cancellation-prone or if the result can be verified a-posteriori.

4.3 Comparison and discussion

In Section 1.2 we explained the implicit SIMD architecture of a GPU, that is equivalent to an explicit SIMD, which means that a GPU program can be also understood as computations on vectors from the point of view of a warp. This enables us to directly implement the data-parallel algorithms we proposed in this chapter.

An excerpt of the code of Algorithm 41 is illustrated in Figure 4.7. The code appears from a single thread's perspective, but it runs in parallel and it takes decisions based on the vector lane within an expansion (i.e., *threadIdx.x*). Although we present here a version of the code that is parameterized by only one parameter, r, our actual implementation uses different template parameters for inputs and output, meaning that we allow static generation of any input-output precision combinations.

Our implementation targets GPUs with compute capability 3.0 or above, such as Kepler and Maxwell architectures, that support *warp vote* and *shuffle* instructions. Warp shuffle instructions are used to shift vector components to propagate the errors across expansion terms, and to insert and extract scalar values inside vectors. Although the hardware only supports shuffling binary32 numbers, we implemented shuffle instructions on binary64, by shuffling each half of a number separately. Using warp vote instructions, like the `__any` function, we implement the loop exit conditions of Algorithm 41.

As the algorithms have straightforward control flow, they can be applied to larger vectors containing multiple expansions side by side. This way we exploit both the parallelism that exist between expansion terms and across different expansions. To benefit from the SIMD execution and intra-warp communication primitives, all terms in a given expansion have to be computed by threads of the same warp and, as warps have 32 threads on Nvidia architectures, the maximal supported expansion size is 32. Smaller expansions are packed together inside warps. Although this approach works with any expansion of size r between 1 and 32 using appropriate padding, we recommend using power of two sizes, which allow filling the whole warp.

To analyze the performance of the algorithms and the effect of parallelism on memory footprint, we consider two different shared memory usage scenarios: one best case that assumes the application uses no intermediate data outside of the registers used for the computation, and one worst case where the application uses 32 bytes of cuda shared memory for each term of the expansion. The performance presented here was obtained on the GPU detailed in Section 1.4.1. We measure throughput on embarrassingly-parallel computations using random generated examples, running on 1024 blocks each with 512, 256, 128 and so on, execution threads, depending on the expansion size and the required resources to run the algorithms. We compare with the sequential algorithms presented in Chapter 3. The value r represents the number of terms in both input and output expansions.

```
template<int R>
__device__ double parallelMul(double x, double y){
  int lane = threadIdx.x;  // Index within expansion
  double s = 0., r = 0., y_i, p, s, e, ep;
  for(int i=0; i<R-1; i++) {
    y_i = shfl(y, i, R);  // Broadcast y_i
    p = TwoProdFMA(x, y_i, &e);
    s = TwoSum(s, p, &ep);

    double tmp = shfl(s, 0, R);
    if(lane == i) r = tmp; // Save s_0 to r_i
    s = shfl_down(s, 1);   // Shift left
    if(lane == K-1) s = 0.;

    while(__any(e != 0.)) { // Accumulate e
      s = TwoSum(s, e, &e);
      e = shfl_up(e, 1, R); // Shift right
      if(lane == 0) e = 0.;
    }
    while(__any(ep != 0.)) { // Accumulate e'
      s = TwoSum(s, ep, &ep);
      ep = shfl_up(ep, 1, R); // Shift right
      if(lane == 0) ep = 0.;
    }
  }
  y_i = shfl(y, R-1, R);
  p = x * y_i;
  s = s + p;
  double tmp = shfl(s, 0, R); // save r_{R-1}
  if (lane == R-1) r = tmp;
  return r;
}
```

Figure 4.7 – Algorithm 41 implemented in CUDA C, simplified for the case in which the inputs and the output have the same power-of-two size, r.

In Table 4.1 we show the performance of the addition algorithm for the best case, no memory configuration, followed by the performance obtained in the memory constrained configuration, in Table 4.2. For the multiplications algorithm we assess performance in Tables 4.3 and 4.4, for the best case setting and for the memory constrained one, respectively.

Table 4.1 – GPU performance in Mop/s for the addition algorithms in the best case with no internal memory usage.

r	Alg. 40	Quick Alg. 40	Alg. 27	Alg. 29	Fast Alg. 29
2	10,131	11,500	30,134 (Alg. 9)	30,134 (Alg. 9)	51,822 (Alg. 8)
4	1,695	4,085	1,080	1,929	4,856
8	368	1,793	425	641	1,728
16	87.6	760	154	160	348
32	20.8	120	49.5	25.6	72.7

Table 4.2 – GPU performance in Mop/s for the addition algorithms in the memory constrained case with 32 B shared memory per expansion term.

r	Alg. 40	Quick Alg. 40	Alg. 27	Alg. 29	Fast Alg. 29
2	9,865	9,251	25,953 (Alg. 9)	25,953 (Alg. 9)	38,876 (Alg. 8)
4	1,672	3,330	536.3	1,210	2,272
8	364	1,340	119	249	456
16	86.9	540	28.5	41.4	61.8
32	20.7	89.5	7.54	4.83	8.85

Table 4.3 – GPU performance in Mop/s for the multiplication algorithms in the best case with no internal memory usage.

r	Alg. 41	Alg. 31	Alg. 33	Fast Alg. 33
2	2,747.2	73,061 (Alg. 14)	73,061 (Alg. 14)	73,061 (Alg. 14)
4	510	354	1,893	2,959
8	107	87.7	358	469
16	23.7	16.2	43.4	44.1
32	1.64	3.49	0.72	0.74

Table 4.4 – GPU performance in Mop/s for the multiplication algorithms in the memory constrained case with 32 B shared memory per expansion term.

r	Alg. 41	Alg. 31	Alg. 33	Fast Alg. 33
2	2,416	62,248 (Alg. 14)	62,248 (Alg. 14)	62,248 (Alg. 14)
4	456	291	977	1,444
8	94.4	49.1	94.3	116
16	20.6	5.76	8.44	9.09
32	1.39	0.64	0.23	0.28

Even in the worst-case embarrassingly-parallel setup, the performance of data-parallel algorithms is competitive with the sequential algorithms for large expansions: the parallelism comes at little cost in number of operations per expansion. For small expansions, like the double-word case the parallel algorithms suffer from parallelization overhead.

The benefits of exploiting the parallelism available within each expansion are fully realized when parallelism is constrained by internal memory usage. The performance of data-parallel algorithms remains stable in this setup, while the performance of sequential algorithm decreases sharply with memory usage. Although the sequential algorithms remain faster on expansions of size 2 (double-word), the data-parallel algorithms significantly outperform their sequential counterparts for all larger expansions, due to the distribution of memory usage over more threads. The performance gap increases with the expansion size, eventually reaching an order of magnitude for 32-term expansions.

These data-parallel algorithms can be used as a starting point into developing parallel arithmetic algorithms suitable for different parallel architectures.

Chapter 5

Applications

During this thesis, besides our efforts towards providing arithmetic algorithms for performing computations using floating-point expansions, we also looked into applications for our work. More specifically we looked into two problems. The first one comes from the dynamical systems field, the Hénon attractor, presented in Section 5.1. The second application comes from experimental mathematics, the semidefinite programing solver presented in Section 5.2, that actually has a broader range of use, form quantum chemistry to control theory.

This resulted in two publications:
– *Searching for Sinks for the Hénon Map Using a Multiple-precision GPU Arithmetic Library* [46], joint work with M. Joldes and W. Tucker,[1] published in the ACM SIGARCH Computer Architecture News - HEART '14 journal.
– *Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming* [44], joint work with M. Joldes and J.-M. Muller, which was accepted for publication and is going to be presented at the 24th IEEE Symposium on Computer Arithmetic (ARITH 2017).

5.1 Hénon attractor

CAMPARY was initially developed and tuned for long time iteration of chaotic dynamical systems in extended precision. Using a first version of the library we looked into one of the "classic" discrete dynamical systems for which several long-standing open questions remain, the Hénon map [32].

5.1.1 Mathematical background

The Hénon map is a two-parameter, invertible map

$$h(x, y) = (1 + y - ax^2, bx).$$

Depending on the two parameters a and b, the map can be:

- chaotic - trajectories belonging to the attractor are aperiodic and sensitive to initial conditions;
- regular - the attractor of the map is a stable periodic orbit;
- a combination of the above two.

1. Professor at Department of Mathematics, Uppsala University, Sweden.

It was conjectured that for the classical parameters $a = 1.4$ and $b = 0.3$, the Hénon map is chaotic and supports a strange attractor. This property has been observed numerically, but the question whether or not the Hénon attractor is indeed chaotic remains open.

It is known [8] that there exist a set of parameters (near $b = 0$) with positive Lebesgue measure for which the Hénon map has a strange (chaotic) attractor. The parameter space is believed to be densely filled with open regions, where the attractor consists of one or more stable periodic orbits (sinks). In light of this, it is probably impossible to verify that, given a specific point (a, b) in parameter space, the dynamics of the map generates a strange attractor.

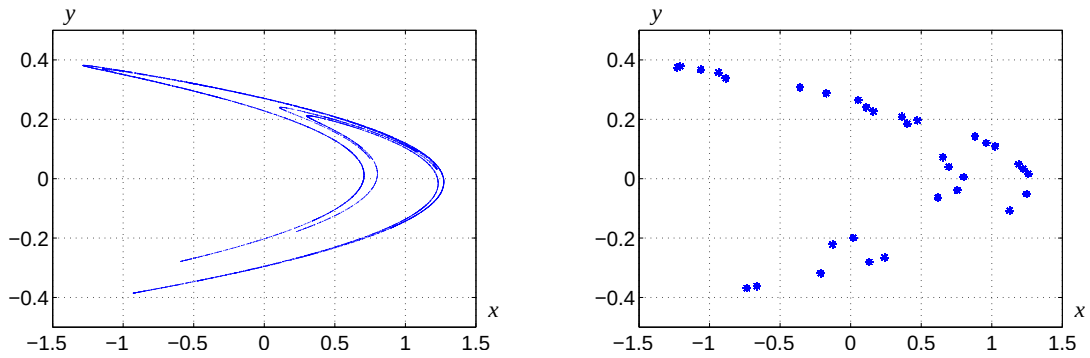
On the other hand, it was proven, using validated numerics [26], that for several parameter values close to the classical ones, what appears to be a strange attractor is actually a stable periodic orbit. Such parameters are given in Example 5.1.1.

Example 5.1.1. *When considering the two fixed parameters $a = 1.399999486944$ and $b = 0.3$, the Hénon map can be reduced to a sink with 33 points, as shown in Figure 5.1, where 10000 iterations of the map $h(x, y)$ are plotted. More specifically, the iterates appearing in part (a) of the figure start in a point (x_0', y_0') and those in part (b) in a different point (x_0'', y_0''). We chose the two points in the following way:*

(a) $5 \cdot 10^9$ iterations are performed and skipped (not plotted) before obtaining (x_0', y_0');

(b) respectively, $6 \cdot 10^9$ iterations are skipped before obtaining (x_0'', y_0'').

One can observe that, what looks like the Hénon strange attractor in Figure 5.1(a), proves to be just the periodic orbit showed in Figure 5.1(b). This means that what we observe in computer simulations is actually a transient behavior to the periodic steady state that we are interested in.



(a) (b)

Figure 5.1 – Hénon map with parameters $a = 1.399999486944$, $b = 0.3$; 10000 iterates are plotted after skipping (a) $5 \cdot 10^9$ and (b) $6 \cdot 10^9$ iterations.

Proving the existence of such a stable periodic orbit involves a finite (yet challenging) amount of computations and we should theoretically be able to find them using high performance computing.

5.1.2 Computational approach

We adapted the method in [26] where a CPU architecture is used. In brief:

(i) for each considered point (a, b) in parameter space, we perform a large amount of iterations of the Hénon map h, for many different initial points. The hope is that at least one of these trajectories will, after some initial transient behavior, be attracted to what appears to be a periodic orbit;

(ii) we use rigorous numerics to validate/falsify the existence of any sink found at the previous step.

With more detail, given a fixed (a, b) together with a single initial point (x_0, y_0), the subsequent computations are governed by two integers N_t and p_{max}.

- First, we perform N_t iterations of the map h which now depends on (a, b) :

$$h(x_0, y_0), h(h(x_0, y_0)), \ldots, h^{N_t}(x_0, y_0).$$

- These are all discarded, except the final iterate $h^{N_t}(x_0, y_0)$.
- Starting from $h^{N_t}(x_0, y_0)$ we continue to follow for another p_{max} iterates.
- At this stage, we examine the piece of orbit

$$h^{N_t+1}(x_0, y_0), \ldots, h^{N_t+p_{max}}(x_0, y_0)$$

for any close returns.

In other words, we attempt to find an integer $1 < k < p_{max}$ such that

$$\max_{i=1}^{k} \left| h^{N_t+i}(x_0, y_0) - h^{N_t+i+k}(x_0, y_0) \right|$$

is small. If this succeeds, we may have found a period-k sink, which we later attempt to verify using rigorous numerics.

The number N_t of transient iterations which are discarded is usually chosen by trial-and-error since it depends on hidden intrinsic properties of the dynamics of the Hénon map. In practice, $N_t \sim 10^9$. In our search, we have used $p_{max} = 5000$. For each parameter we use $N_i \sim 10^3$ different initial points. Finally, we repeat the entire procedure for $N_p \sim 10^6$ parameters near $(1.4, 0.3)$.

If at the end of this search process we identify some "numerical periodic orbits", in a second step we rigorously prove their existence using methods from interval analysis [67, 75]. This part can be checked "off-line" on a CPU architecture, and we use the procedure described in [26], which is based on an interval Newton operator. This step is not detailed further here, since it is only the first part that is computationally expensive. Its complexity depends on two main factors: the precision used for computations, and the capability of exploiting the inherent parallelism available in the parameter space and the initial points considered for each parameter.

Parallelization approach. In order to tackle the conjecture we need to analyze a very wide parameter space, so the computation can be viewed as a SIMD parallel problem, for which a GPU implementation is suitable.

The main idea for the parallelization on GPU is that each thread computes the iterates of the map h starting with one fixed initial point and fixed parameter (a, b) (these iterations are inherently sequential). The initial points are generated in a suitable region close to the attractor by a single thread on the GPU. They are stored in a shared memory array that gives access to all the other threads in the same block. Each thread writes in a shared memory array the period (if any) of the orbit, and one point of the orbit in the affirmative case. Each block is bi-dimensional and corresponds to one parameter (a, b). We grid the parameter space near $(1.4, 0.3)$ and apply the above process to each grid point. We also implemented some variants where several blocks correspond to the same parameter (a, b) in order to be able to iterate on more initial points. Without this ability, we are limited by the size of the block's shared memory to ca 10^3 initial points/parameter, depending on the used GPUs architecture and model.

Remark 5.1.2. *Before going further we would like to stress here that this research was performed in 2013-2014, when we did not have access to the server presented in Section 1.4.1. This is why we present our results based on a previous architecture available at the time:*

- *an Nvidia GeForce Tesla C2075 GPU with 448 cores, 1.15GHz and,*
- *for the CPU code, an Intel(R) Core(TM) i7 CPU 3820, 3.6GHz, 4 cores, 8 threads computer.*

As a first step in our research we implemented this GPU-oriented method using the standard binary64 format, to compare the performance with respect to the CPU implementation in [26] and we re-checked the same orbits already found there, given in Table 5.1. For example, for 10^6 grid points for $a \in [1.3999, 1.4001]$, $b = 0.3$ fixed, 1024 orbits/parameter and 10^6 iterations/orbit we found 57 parameters which present stable periodic orbits in 2.94 hours on 2 GPUs. A 21.5x speedup is obtained by our CUDA C implementation vs, a C implementation with OpenMP; for other Intel(R) platforms like Xeon E5 series the speed-ups are similar.

Table 5.1 – Known Hénon map orbits from [26]. The parameter $b = 0.3$ is fixed, P is the period, d is the distance to the point $(1.4, 0.3)$, r is the minimum immediate basin radius, and λ_1 is the largest Lyapunov exponent.

a	P	d	r	λ_1
1.399922051	25	$5.522e{-12}$	$2.473e{-12}$	-0.00132
1.39997174948	30	$1.354e{-11}$	$3.561e{-12}$	-0.01887
1.3999769102	18	$3.207e{-09}$	$1.014e{-09}$	-0.05306
1.39998083519	24	$1.703e{-11}$	$7.384e{-12}$	-0.02819
1.399984477	20	$8.875e{-10}$	$4.076e{-10}$	-0.05099
1.39999492185	22	$3.686e{-11}$	$1.531e{-11}$	-0.09600
1.3999964733062	39	$2.784e{-13}$	$1.115e{-13}$	-0.03547
1.399999486944	33	$1.110e{-12}$	$6.901e{-13}$	-0.01843
1.40000929916	25	$1.118e{-11}$	$5.128e{-12}$	-0.08379
1.4000227433	21	$2.262e{-10}$	$7.901e{-11}$	-0.05612
1.40002931695	27	$5.782e{-11}$	$2.646e{-11}$	-0.01140
1.40006377472	27	$8.692e{-11}$	$3.810e{-11}$	-0.05636
1.40006667358	24	$6.278e{-11}$	$2.646e{-11}$	-0.01112
1.4000843045	27	$9.400e{-11}$	$4.572e{-11}$	-0.06870
1.40009110518	22	$3.493e{-11}$	$1.531e{-11}$	-0.02157
1.4000967515	26	$2.463e{-10}$	$1.365e{-10}$	-0.13233

But binary64 precision does not suffice if we want to obtain sinks for parameters closer to the classical ones, so we had to increase the precision. Using the "quick-and-dirty" level of our library we were able to adapt the code written for binary64 Hénon map iterations using only minor changes. This was possible due to the operators that we overload. An excerpt of the code using quad-double precision is:

```
#define prec 4
/*device fct to be run using prec*doubles precision*/
__host__ __device__ void henon_iterate(double x0, double y0,
                                       double a, double b, long int ITER){
  /*init multi_prec template vars*/
  multi_prec<prec> x_i(x0);
  multi_prec<prec> y_i(y0);
  multi_prec<prec> x_old;
  for (long int i=1; i <= ITER; i++) {
    /*Compute iterates*/
    x_old = x_i;
    x_i = y_i + 1.0 - a*x_i*x_i;
    y_i = b*x_old;
  }
}
```

5.1.3 Numerical results and performance

A performance comparison between CAMPARY versus GQD library computations on GPU for Hénon iterations is given in Table 5.2. In the same table we also compare performance versus binary64 computations. In what follows, we denote by 2D double-double, 3D triple-double, 4D quad-double, 5D quintuple-double, and so on.

Table 5.2 – Peak number of Hénon map orbits/second for double vs. CAMPARY vs. QD library on Tesla GPU[C2075] using 10^6 iterations/orbit. *precision not supported

Precision	CAMPARY	QD
binary64	102398	
2D	7608	4539
3D	5200	*
4D	1788	618
5D	758	*
6D	374	*
7D	205	*
8D	122	*

For this problem we also intended to compare performance with GARPREC and CUMP libraries, but we were not able to straightforwardly adapt our code because they were both tuned for big array operations where the data is generated on the host, and only the operations are performed on the device. In our case, each thread needs to generate and allocate multiple-precision data on the device. However, in [61] it is stated that GQD should be faster that GARPREC for double-double and quad-double computations. Moreover, it is also known [34, 61] that multiple-term operations are usually faster than multiple-digit ones for precisions in the range of up to several hundreds of bits, which was also confirmed in our case.

More precisely, for the same benchmark Hénon map code we also compare the performances on a CPU implementations parallelized with OpenMP using CAMPARY versus MPFR in Table 5.3. This comparison is not entirely fair seeing that the multiple-digit format is not equivalent to the

multiple-terms format. Indeed, we do not guarantee the correct rounding for each basic operation, but we present this comparison from a prospective point of view.

Table 5.3 – Peak number of Hénon map orbits/second for CAMPARY vs. MPFR library (both parallelized with OpenMP on 8 threads) on Intel i7-3820 @3.60GHz using 10^6 iterations/orbit.

Precision	CAMPARY	MPFR
2D (106 bits)	227	11.8
3D (159 bits)	76	10.6
4D (212 bits)	37	10.1
6D (318 bits)	15	8.9
8D (424 bits)	8	7.9

It is clear from a mathematical point of view that only a very small amount of sinks can be found using double-precision. At the same time, any sink can be resolved using a sufficiently high precision.

In [99] Galias and Tucker present a very extensive study of the problem. Two of the orbits presented there, given in Table 5.4, were obtained using our GPU implementation.

Table 5.4 – Hénon map sinks found using CAMPARY on GPU [99]. P is the period, d is the distance to the point $(1.4, 0.3)$, r is the minimum immediate basin radius.

P	d	r
41	$4.73e{-}10$	$3.31e{-}17$
47	$1.47e{-}10$	$1.17e{-}20$

In the same paper, [99], the authors report that the closest to the classical parameters periodic window that they found, with $d = 6.335e{-}22$, is given by parameters:

$$a = 1.39999999999999999999968839903277301984563091568983, \text{ and}$$

$$b = 0.29999999999999999999944845519288458244332946957783,$$

for which a period-115 sink was confirmed. They concluded their search by stating that since the periodic windows are very narrow and the transient time to corresponding sinks can be extremely long, "it is practically impossible to observe such sinks in simulations".

This work provided us with better means for observing the behavior of dynamical systems, which would be impossible using only standard machine precision numbers, due to the numerical instability of the problem. These first results offered numerical support for the belief that the parameter space close to the classical ones $(1.4, 0.3)$ is densely filled with open regions, where the attractor can be reduced to a periodic sink.

5.2 SDP solver

A known class of problems that can benefit from increased precision is the class of semidefinite optimization problems (SDP) which come to solving in a very accurate way, numerically sensitive

(and sometimes large-scale). Examples include problems from experimental mathematics, like the high-accuracy computation of *kissing numbers*, i.e., the maximal number of non-overlapping unit spheres that simultaneously can touch a central unit sphere [63]; bounds from binary codes [21]; control theory and structural design optimization (e.g., the wing of Airbus A380) [21]; quantum information and physics [89].

Given the wide range of possible applications and the recent increased interest in the subject, as a second application for CAMPARY, we also took interest in higher-precision and high-performance SDP solvers.

Developing such a solver comes with multiple challenges. Firstly, one has to establish the core mathematical algorithm for numerically solving (say, in "real numbers") the SDP problem: most nowadays solvers employ primal-dual path-following interior-point method (PDIPM) [65]. Secondly, the underlying multiple-precision arithmetic operations have to be treated. Some high-precisions solvers were developed, and though they are more accurate, they are also more computationally expensive [72]. Finally, since most problems are large-scale, parallelization is also very important. Having (at least partially) computations done on highly parallel architectures like GPUs is of interest.

5.2.1 Mathematical background

Semidefinite programing (SDP) is a convex optimization problem, which can be seen as a natural generalization of linear programming to the cone of symmetric matrices with non-negative eigenvalues, i.e., positive semidefinite matrices. While linear programming optimizes a linear functional subject to linear inequality constraints, SDP optimizes a linear functional subject to linear matrix inequalities (LMIs). Many optimization problems in automatic control or signal processing can be formulated using LMIs.

Denote by $\mathbb{R}^{n\times n}$ the space of size $n \times n$ real matrices, by $\mathbb{S}^n \subseteq \mathbb{R}^{n\times n}$ the subspace of real symmetric matrices, equipped with the inner product $\langle A, B\rangle_{\mathbb{S}^n} = \mathrm{tr}(A^T B)$, where $\mathrm{tr}(A)$ denotes the trace of the matrix A. Also, denote by $A \succcurlyeq O$ the fact that A is positive semidefinite (and respectively $A \succ O$ for positive definite). A typical SDP program is expressed in its primal-dual form as follows:

$$\text{(P)} \quad \begin{aligned} p^* = &\sup_{X\in\mathbb{S}^n} \langle C, X\rangle_{\mathbb{S}^n} \\ &\text{s.t. } \langle A_i, X\rangle_{\mathbb{S}^n} = b_i, i = 1, \ldots, m, \\ &\quad X \succcurlyeq O, \end{aligned}$$

$$\text{(D)} \quad \begin{aligned} d^* = &\inf_{y\in\mathbb{R}^m} b^T y \\ &\text{s.t. } Y := \sum_{i=1}^{m} y_i A_i - C \succcurlyeq O, \end{aligned}$$

where $C, A_i \in \mathbb{S}^{n\times n}, i = 1, \ldots m$ and $b \in \mathbb{R}^m$ are given.

However, in general, it is difficult to obtain an accurate optimum for a SDP problem. On the one hand, strong duality does not always hold, unlike for linear programs: weak duality is always satisfied, i.e., $p^* \leq d^*$, but sometimes, p^* is strictly less than d^*. Simpler instances are those where strong duality holds and this happens when the feasible set contains a positive definite matrix [56, Theorem 1.3].

More than often the method of choice for SDP solving is based on the interior-point algorithm, which relies on the existence of interior feasible solutions for problems (P) and (D). In such cases, these two problems are simultaneously solved in polynomial time in the size of parameters of the input problem using the well-established primal-dual path-following interior-point method (PDIPM) [65]. This algorithm is considered in the literature as theoretically mature, is widely

accepted and implemented in most state-of-the-art SDP solvers. The method relies on the fact that when an interior feasible solution exits one has the necessary and sufficient condition for X^*, Y^* and y^* to be an optimal solution:

$$X^*Y^* = 0,\ X^* \succ O,\ Y^* \succ O, \tag{5.1}$$

$$Y^* = \sum_{i=1}^{m} y_i^* A_i - C, \tag{5.2}$$

$$\langle A_i, X\rangle_{\mathbb{S}^n} = b_i, i = 1, \ldots, m, \tag{5.3}$$

The complementary slackness condition (5.1) is replaced by a perturbed one: $X_\mu Y_\mu = \mu I$. It is known that this perturbed system has a unique solution and that the central path, that is the set $\mathcal{C} = \{(X_\mu, Y_\mu, y_\mu) : \mu > 0\}$ forms a smooth curve converging to (X^*, Y^*, y^*) as $\mu \to 0$. So, the main idea of the method is to numerically trace the central path $\mathcal{C}$.

In Algorithm 42 we give a sketch of the algorithm adapted from [72]. Step 1 describes the procedure for computing the search direction based on Mehrotra type predictor-corrector [96]. The stopping criteria (Step 4) depends on several quantities: primal and dual feasibility error are defined as the maximum absolute error appearing in (5.3) and (5.2), respectively. The duality gap depends on the absolute or relative difference between the objectives of (P) and (D).

Algorithm 42 – PDIPM Algorithm, adapted from SDPA implementation [72].

Step 0: Choose an initial point $X^0 \succ O$, y^0 and $Y^0 \succ O$. Set $h = 0$ and choose the parameter $\gamma \in (0, 1)$.

Step 1: Compute a search direction:
Evaluate the Shur Complement Matrix $B \in \mathbb{S}^n$ by the formula $B_{ij} = \langle (Y^h)^{-1} A_i X^h, A_j\rangle$.
Solve the linear equation $Bdy = r$. Using the solution dy, compute dX, dY and obtain the search direction (dy, dX, dY).

Step 2: Compute max step length α to keep the positive semidefiniteness: $\alpha = \max\{\alpha \in [0, 1] : Y^h + \alpha dY \succ O, X^h + \alpha dX \succ O\}$.

Step 3: Update the current point: $(y^{h+1}, X^{h+1}, Y^{h+1}) = (y^h, X^h, Y^h) + \gamma\alpha(dy, dX, dY)$.

Step 4: If $(y^{h+1}, X^{h+1}, Y^{h+1})$ satisfies the stopping criteria, output it as a solution. Otherwise, set $h = h + 1$ and return to Step 1.

However, problems which do not have an interior feasible point [2] induce numerical instability and may result in inaccurate calculations or non-convergences. Even for problems which have interior feasible solutions, numerical inaccuracies may appear when solving with finite precision due to large condition numbers (higher than 10^{16}, for example) which appear when solving linear equations. This happens, as explained in [72], when approaching optimal solutions: suppose there exist $X^* \succ O$, $Y^* \succ O$ and y^* which satisfy all the constraints in (P) and (D); or better said, when $\mu \to 0$ on the central path. Then, $X^*Y^* = 0$. From this it follows that $\text{rank}(X^*) + \text{rank}(Y^*) \leq n$, which implies that these matrices are usually singular in practice.

In the later case, having an efficient underlying multiple-precision arithmetic is crucial to detect (at least numerically) whether the convergence issue came simply from numerical errors due to lack of precision.

2. Recently, SPECTRA package [33] proposes to solve such problems with exact rational arithmetic, but the instances treated are small and this package does not aim to be a concurrent of general numerical solvers.

5.2.2 Existing mathematical software

The PDIPM algorithm is considered in the literature as theoretically mature. This is why most state-of-the-art SDP solvers implement it. The most commonly used are SDPA [96], CSDP [11], SeDuMi [91] and SDPT3 [93]. We took main interest in the SDPA solver, since it provides multiple-precision versions, using the GMP and the QD libraries: SDPA-GMP, SDPA-DD, SDPA-QD.

Starting with version 6.0, SDPA incorporated LAPACK [4] for dense matrix computations, but also exploits the sparsity of data matrices and solves large scale SDPs [97]. More recently, MPACK [71] was developed and integrated with SDPA. This is a multiple-precision linear algebra package which is based on BLAS and LAPACK [4]. For this package, the major change is, as in our case, the underlying arithmetic format, such that users can easily switch from a double-precision BLAS/LAPACK code to a multiple-precision one, in order to obtain better accuracy. MPACK supports various multiple-precision arithmetic libraries like GMP, MPFR, and QD, as well as IEEE 754 binary128 (via `gcc`'s extension `__float128`).

Moreover, MPACK also provided a GPU tuned implementation in double-double arithmetic of the *Rgemm* routine: this is the multiple-precision *Real* version of *Dgemm*, the general double matrix multiplication [73]. This routine is central for other linear algebra operations such as solving linear equations, singular value decomposition or eigenvalue problems. For this, MPACK authors re-implemented parts of QD library for cuda-compliant code.

This routine's implementation is reported in [73] with best practical performance and was intensively tuned for GPU-based parallelism: classical blocking algorithm is employed, and for each element of a block a thread is created; a specific number of threads is allocated per block also. More specifically, in [73], for the NVIDIA(R) Tesla(TM) C2050 GPU, best performance of 16.4 GFlops is obtained for $A \times B$, the product of two matrices A and B with blocks of size: 16×16 for A and 16×64 for B; 256 threads are allocated per block. Shared memory is used for each block. Also, reading is done from texture memory.

5.2.3 SDPA-CAMPARY package

We built this package starting from the SDPA-QD/DD package, where the QD/GQD library was replaced with CAMPARY (using underlying binary64) at the compilation step of SDPA. This can be done efficiently since both SDPA and CAMPARY are written in C/C++.

We also integrated CAMPARY with MPACK, in order to take advantage of the parallelized matrix multiplication for which we used our GPU version of CAMPARY. In our implementation, we use a similar algorithm, except that reading is done from global memory instead of texture memory. We were forced to do this since a texture memory element (texel) size is limited to *int4* (i.e., 128 bits) and our implementation is generic for n-term expansions. In what follows, we denote by 2D double-double, 3D triple-double, 4D quad-double, 5D quintuple-double, and so on.

***Rgemm* performance.** As a first step in our research we compared our implementation of *Rgemm* with the implementation in [73], using a GPU card similar to theirs. Specifically, we used an NVIDIA(R) Tesla(TM) C2075 card, [3] that is part of the same Fermi architecture. The difference is that our GPU has 6 GB of global memory, compared to 3 GB of their NVIDIA(R) Tesla(TM) C2050. However, this has little importance for the performance results on kernel execution once the global memory has been loaded.

In their (re-)implementation of the QD library the authors of [73] used Algorithms 9 and 13 for addition and multiplication of double-double numbers. For a fair comparison we did the same.

3. NVIDIA(R) Tesla(TM) C2075 with 448 cores, 1.15 GHz, 32 KB of register, 64 KB shared memory / L1 cache set by default to 48 KB for shared memory and 16 KB for L1 cache

On top of that, for our arbitrary precision with n-term expansions we used the "quick-and-dirty" level of our library, implemented in the *multi_prec.h* file (see Section 1.4), that uses the fast versions of Algorithms 29 and 33.

Remark 5.2.1. *We recall that the theoretical peak performance is obtained as follows:*

(i) first, consider that in Rgemm operations are mainly multiply-add type, so the theoretical peak for multiply-add is of 1.15 *GHz* $\times 14$ *SM* $\times 32$ *cuda cores* $\times (2$ *Flops*$/2$ *cycle* $) = 515$ *GFlops;*

(ii) second, we compute the theoretical peak for multiple-precision Rgemm by dividing the above peak performance for standard floating-point operations by the operation count for addition plus multiplication with n*-term expansions.*

In Figure 5.2 we compare our implementation of 2D *Rgemm*, with the one in [73]. Our implementation proved to be slower by $\sim 10\%$ than the implementation in [73], which can be explained by the generality of our code. Although we tested our implementation also using texture memory, we observed no speedup. Maximum performance was 14.8 GFlops for CAMPARY and 16.4 GFlops for [73].

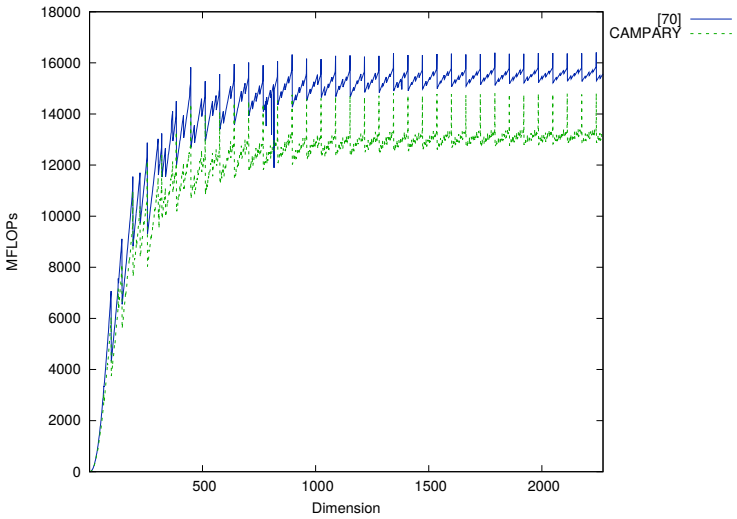


Figure 5.2 – Performance of *Rgemm* with CAMPARY vs [73] using double-double precision on GPU.

On the other hand, in our case, higher precision *Rgemm* is straightforward. Performance results for n-double *Rgemm* are shown in Figure 5.3: one observes that the decrease of performance when the precision is increased fits the increase in the number of standard operations performed for additions and multiplications with n-term expansions. In Table 5.5 we compare the theoretical peak vs. the maximum performance that we obtained for n-double expansions. We also recall the worst case operation count for the algorithms that we employed.

Note that cuBLAS, the NVIDIA GPU linear algebra package does not support precisions higher than binary64 and it is not open source, so we consider it difficult to extend it in the context of multiple-precision linear algebra for GPUs. For Fermi architecture GPUs like C2050 or C2075, the peak performance of *Dgemm* is reported in [92] to be 302 GFlops with cuBlas and 362 GFlops with further optimizations which is 58% and 70% of the theoretical peak performance, so, based in the values reported in Table 5.5, the *Rgemm* implementation we have is quite efficient.

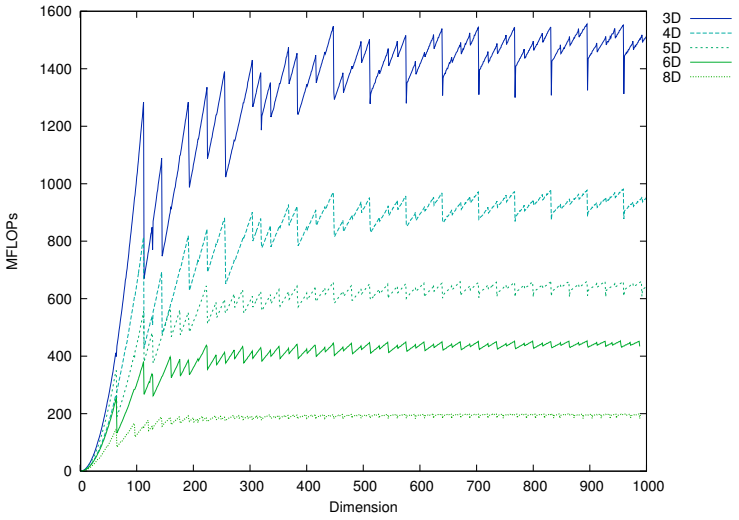


Figure 5.3 – Performance of *Rgemm* with CAMPARY for n-double on Fermi architecture GPUs.

Table 5.5 – Theoretical peak vs. kernel peak for *Rgemm* with CAMPARY for n-double on GPU.

Format	# of Flops		Theoretical peak	Kernel peak	
	+	*			
2D	20	9	17.8 GFlops	14.8 GFlops	83%
3D	55	88	3.6 GFlops	1.6 GFlops	44%
4D	87	183	1,900 MFlops	976 MFlops	51%
5D	125	330	1,130 MFlops	660 MFlops	58%
6D	169	541	725 MFlops	453 MFlops	62%
8D	275	1203	348 MFlops	200 MFlops	57%

5.2.4 Numerical results and performance

In order to verify the SDPA-CAMPARY solver and assess its performance we look at the results obtained for some standard SDP problems both on CPU and GPU. In particular, on GPU we use the *Rgemm* routine, with the implementation explained above. All the tests presented in what follows were performed on the server detailed in Section 1.4.1, which we recall that is not the same as the one used for the *Rgemm* performance assessment. The results obtained on that server are published in [44].

Table 5.6 shows the results and performance obtained for five well-known problems from the SDPLIB package [12]. We compare both SDPA-DD and SDPA-QD with the 2D, 3D, and 4D formats of SDPA-CAMPARY, on both CPU and GPU. One can observe that our 2D and 4D implementations outperform the ones of QD. The 3D format proves that it can be a good alternative for problems for which 2D does not suffice, but for which 4D is too expensive. When looking at the performance obtained with the GPU version versus the CPU one, one can observe that for 3D and 4D precision the GPU enhanced version performs better, while for 2D precision no gain is obtained. This is the case for both SDPA-CAMPARY and SDPA-DD. This can be explained by the overhead given by the memory transfer and GPU-CPU communication. Accuracy wise, SDPA-CAMPARY performs as expected: the results obtained are more and more accurate as precision is increased.

Table 5.6 – The optimal value, relative gaps, primal/dual feasible errors, iterations and time for solving some problems from SDPLIB by SDPA-DD, -QD, and -CAMPARY.

Problem	SDPA-DD		SDPA-QD	SDPA-CAMPARY					
				2D		3D		4D	
	CPU	GPU		CPU	GPU	CPU	GPU	CPU	GPU
`gpp124-1`	optimal: -7.3430762652465384								
relative gap	$7e-04$	$7e-04$	$6e-13$	$7e-04$	$7e-04$	$8e-12$	$6e-12$	$1e-18$	$8e-18$
p.feas.error	$1e-19$	$5e-20$	$3e-45$	$1e-19$	$8e-20$	$1e-30$	$1e-30$	$2e-41$	$4e-42$
d.feas.error	$5e-14$	$3e-14$	$7e-31$	$2e-14$	$3e-14$	$1e-16$	$1e-16$	$2e-21$	$2e-22$
iteration	24	24	40	24	24	38	39	58	49
time (s)	1.32	1.96	23.4	1.12	1.62	14.2	12.14	32.6	29.5
`gpp250-1`	optimal: $-1.5444916882934070e+01$								
relative gap	$5e-04$	$5e-04$	$4e-13$	$5e-04$	$5e-04$	$4e-12$	$8e-12$	$5e-18$	$5e-19$
p.feas.error	$2e-20$	$2e-20$	$3e-45$	$2e-20$	$5e-20$	$1e-31$	$1e-30$	$1e-41$	$6e-42$
d.feas.error	$1e-13$	$7e-14$	$1e-30$	$1e-13$	$1e-13$	$1e-16$	$9e-16$	$4e-21$	$7e-21$
iteration	25	25	41	25	25	41	40	56	66
time (s)	7.86	9.43	151.7	6.46	7.42	80.5	71.7	200.7	213.5
`gpp500-1`	optimal: $-2.5320543879075792e+01$								
relative gap	$1e-03$	$1e-03$	$4e-13$	$1e-03$	$1e-03$	$9e-12$	$9e-12$	$2e-17$	$2e-17$
p.feas.error	$1e-20$	$1e-20$	$2e-45$	$6e-21$	$1e-20$	$7e-31$	$1e-30$	$2e-42$	$1e-42$
d.feas.error	$8e-14$	$7e-14$	$3e-30$	$7e-14$	$6e-14$	$1e-16$	$3e-16$	$7e-22$	$8e-22$
iteration	25	25	42	25	25	40	41	56	53
time (s)	50.4	54.2	1,053	43.11	42.3	520.5	466.9	1,356	1,112
`qap10`	optimal: $-1.0926074684462390e+03$								
relative gap	$1e-04$	$3e-05$	$1e-14$	$6e-05$	$4e-05$	$1e-09$	$2e-10$	$6e-15$	$2e-14$
p.feas.error	$3e-21$	$2e-21$	$3e-46$	$2e-20$	$5e-21$	$9e-35$	$6e-34$	$4e-47$	$2e-47$
d.feas.error	$3e-13$	$2e-14$	$3e-30$	$1e-13$	$7e-14$	$2e-21$	$1e-22$	$5e-31$	$2e-30$
iteration	19	20	37	59	20	27	29	37	36
time (s)	12.87	16.4	288.9	74.2	12.4	114	116.9	284.7	261.7
`theta5`	optimal: $5.7232307282180003e+01$								
relative gap	$6e-25$	$1e-23$	$2e-46$	$1e-25$	$1e-24$	$2e-31$	$2e-31$	$2e-31$	$2e-31$
p.feas.error	$8e-31$	$4e-31$	$1e-63$	$7e-31$	$7e-31$	$1e-45$	$1e-44$	$5e-61$	$2e-61$
d.feas.error	$1e-27$	$2e-25$	$9e-49$	$1e-26$	$9e-26$	$4e-39$	$6e-39$	$1e-55$	$2e-55$
iteration	70	54	58	65	54	43	43	43	43
time (s)	470.2	401.25	4,687	312.3	319.6	1,837	1,533	3,373	2,685

While the QD library offers only two extended precisions, GMP offers arbitrary precision, so we compare our library against it in Table 5.7. Note that GMP has yet to been ported on GPU. We consider three problems from the SDPLIB with corresponding precisions of $106, 159, 212, 318$ and 424 bits.

One can see that our library outperforms GMP by far for all tested precisions, even without GPU support. However this was not the case in the tests presented in [44] (where we used a different server configuration), where GMP performed better for precisions higher than 4D. This shows that the underlying architecture plays a very important role in the performance of these libraries and that our library takes better advantage of the newer architectures, like the ones used for these tests.

Table 5.7 – The optimal value, iterations and time for solving some problems from SDPLIB by SDPA-GMP, -CAMPARY.

Problem	SDPA-CAMPARY		SDPA-GMP
	CPU	GPU	
gpp124-1	optimal: -7.3430762652465384		
precision	2D		106 bits
iteration	24	24	38
time (s)	1.12	1.62	88.6
precision	3D		159 bits
iteration	38	39	48
time (s)	14.2	12.14	117.8
precision	4D		212 bits
iteration	58	49	59
time (s)	32.6	29.5	159.1
precision	6D		318 bits
iteration	77	77	77
time (s)	109.2	103	225.5
precision	8D		424 bits
iteration	77	77	77
time (s)	223.4	271.7	280.7
gpp250-1	optimal: $-1.5444916882934070e+01$		
precision	2D		106 bits
iteration	25	25	39
time (s)	6.46	7.42	715
precision	3D		159 bits
iteration	41	40	46
time (s)	80.5	71.7	889.9
precision	4D		212 bits
iteration	56	66	64
time (s)	200.7	213.5	1,359
precision	6D		318 bits
iteration	73	73	73
time (s)	648.5	578.5	1,683
precision	8D		424 bits
iteration	73	73	73
time (s)	1,449	1,253	2,103
theta5	optimal: $5.7232307282180003e+01$		
precision	2D		106 bits
iteration	65	43	46
time (s)	312	1,837	28,072
precision	3D		159 bits
iteration	43	43	43
time (s)	1,837	1,533	27,601
precision	4D		212 bits
iteration	43	43	43
time (s)	3,373	2,685	31,377
precision	6D		318 bits
iteration	43	43	43
time (s)	8,538	6,476	33,857
precision	8D		424 bits
iteration	43	43	43
time (s)	17,392	13,248	41,206

In Table 5.8 we compare the performance obtained by SDPA-CAMPARY on CPU versus GPU, when varying precision from four to eight doubles, for five problems from the SDPLIB.

Table 5.8 – The optimal value, iterations and time for solving some problems from SDPLIB by SDPA–CAMPARY, when varying precision from 4D to 8D.

			`gpp500-1`	`theta5`	`theta6`	`equalG51`	`mcp500-1`
		opt.	$-2.532054e+01$	$5.723230e+01$	$6.347708e+01$	$4.005601e+03$	$5.981485e+02$
4D	CPU	it.	56	43	44	45	43
		(s)	1,356	3,373	8,291	7,856	793
	GPU	it.	53	43	44	45	43
		(s)	1,112	2,685	6,052	6,654	701
5D	CPU	it.	63	43	44	45	43
		(s)	2,563	5,745	14,157	13,217	1,339
	GPU	it.	63	43	44	45	43
		(s)	2,194	4,410	9,921	11,092	1,159
6D	CPU	it.	82	43	44	45	43
		(s)	4,969	8,538	21,153	19,712	2,007
	GPU	it.	82	43	44	45	43
		(s)	4,282	6,476	14,543	16,466	1,760
7D	CPU	it.	82	43	44	45	43
		(s)	7,337	12,380	30,749	29,384	2,967
	GPU	it.	82	43	44	45	43
		(s)	6,225	9,220	20,624	24,292	2,534
8D	CPU	it.	82	43	44	45	43
		(s)	10,324	17,392	42,971	41,092	4,231
	GPU	it.	82	43	44	45	43
		(s)	8,985	13,248	29,854	34,925	3,694

For testing not only the performance, but also the accuracy of our library, we considered several examples from Sotirov's collection [21], which are badly conditioned numerically, and cannot be tackled using only binary64. A classical problem in coding theory is finding the largest set of binary words with l letters, such that the Hamming distance between two words is at least some given value d. This is reformulated as a maximum stable set problem, which is solved with SDP, according to the seminal work of Schrijver [87], followed by Laurent [55].

In Table 5.9 we show the performance obtained for the Schrijver and Laurent instances from [21]. The comparison is done between SDPA-DD, SDPA-GMP (run with 106 bits of precision) and SDPA-CAMPARY with 2D and 3D. Some instances do not converge when 2D precision is used, this is why we also include the 3D results.

A strange behavior that we obtained on these instances is that the GPU versions of both SDPA-DD and SDPA-CAMPARY perform worse than their CPU counterparts. This is probably due to the nature of the problems treated and the way they are formulated. On the CPU side of these tests one can observe once again that our library outperforms its counterparts and even more, our 3D precision version is faster than GMP with 106 bits.

Table 5.9 – The optimal value, iterations and time for solving some ill-posed problems for binary codes by SDPA-DD, -GMP and -CAMPARY. $*$ problems that converge to more than two digits only with 4D precision. $**$ problems that converge to more than two digits with precision higher than 4D. The digits written with blue were obtained only when 3D precision was employed.

Problem	SDPA-DD		SDPA-CAMPARY				SDPA-GMP
			2D		3D		106 bits
	CPU	GPU	CPU	GPU	CPU	GPU	
Laurent_A(19,6)	optimal: $-2.4414745686616550e-03$						
iteration	94	89	92	87	71	71	73
time (s)	22.14	77.9	18.98	77.49	34.7	99.77	45.63
Laurent_A(26,10)	optimal: $-1.3215201241629400e-05$						
iteration	79	79	77	77	121	138	125
time (s)	33	105	27.67	95	155.32	357.8	270.2
Laurent_A(28,8)	optimal: $-1.1977477306795422e-04$						
iteration	90	101	95	98	76	76	113
time (s)	88.3	337.8	74.31	311	274.4	565.9	837.9
Laurent_A(48,15)	optimal: $-2.229e-09$						
iteration	133	133	133	132	164	164	145
time (s)	2,777	4,469	2,101	3,845	13,889	17,792	34,969
Laurent_A(50,15)	optimal: $-1.9712e-09$						
iteration	143	161	144	151	175	172	154
time (s)	4,143	11,393	3,127	5,841	20,534	25,053	56,364
Laurent_A(50,23)	optimal: $-2.5985e-13$						
iteration	126	125	123	126	155	156	140
time (s)	533	814.6	361	671.5	2,915	3,515	5,385
Schriver_A(19,6)	optimal: $-1.279036270018091000e+03$						
iteration	41	41	40	40	66	66	95
time (s)	8.5	36	8.24	33.1	28	86.8	50.3
Schriver_A(26,10)	optimal: $-8.8585714285713880e+02$						
iteration	54	54	54	54	124	118	108
time (s)	21.2	66.7	17.7	67.2	143.4	290	209.3
Schriver_A(28,8)	optimal: $-3.2150795825792913e+04$						
iteration	45	45	45	45	70	72	97
time (s)	40.6	143.2	34.5	142.5	226.4	502	656
Schriver_A(37,15)	optimal: $-1.400699999999999886e+03$						
iteration	58	59	58	59	141	163	116
time (s)	104.5	225	78.53	214.3	970.9	1,654	1,532
Schriver_A(40,15)*	optimal: $-1.9e+04$						
iteration	23	23	23	23	23	23	23
time (s)	84	168.4	62.34	153.7	349.7	494	742.2
Schriver_A(48,15)*	optimal: $-2.56e+06$						
iteration	27	27	27	27	27	27	27
time (s)	526.8	865.3	400	768	2,095	2,687	6,172
Schriver_A(50,15)**	optimal: $-7.6e+06$						
iteration	29	29	29	29	29	29	29
time (s)	792.1	1,245	596.8	1,103	3,142	3,937	10,224
Schriver_A(50,23)**	optimal: $-5e+03$						
iteration	29	29	29	29	29	29	29
time (s)	119.8	188.7	86.74	158.5	523.7	631.8	1,069

6 CHAPTER 6
Conclusions

Nowadays, very efficient arithmetic operations with double-precision floating-point numbers compliant with the IEEE-754 standard are available on most recent computers. However, when more than double-precision/binary64 (53 bits) is required, especially in the HPC context, fewer multiple-precision arithmetic libraries exist, and the trade-off between performance versus reliability is still a challenge.

To address this challenge, in this work we propose to represent multiple-precision numbers as unevaluated sums of standard machine precision floating-point numbers, so-called **floating-point expansions**. This approach allows us to directly benefit from the available and efficient hardware implementation of the IEEE-754 standard.

Although several works exist in this area, many algorithms have been published without a proof, or with error bounds that are sometimes loose, sometimes fuzzy (the error is "less than a small integer times u^2"), and sometimes unsure. Thus we started this study by first "cleaning up" the literature.

This allowed us to improve or design several new algorithms for performing basic arithmetic operations using this extended format. For all the algorithms that we present we give rigorous correctness and error bound proofs. For all the obtained theoretical results we provide an implementation in our mupltiple-precision arithmetic library, CAMPARY. This library targets both CPUs and GPUs, provides flexible and user friendly routines and it allows one to easily reprogram a problem that uses standard precision, into a program that uses extended precision by providing overloaded operators for all the basic arithmetic operations.

Specifically, we provide:

- specialized algorithms for addition, multiplication and division using double-word numbers, that allow for very efficient computations with "double" of the available precision;
- algorithms for all basic operations ($+/-, \times, /, \sqrt{}$) using arbitrary precision, in the range of a few hundred bits;
- parallel algorithms for arbitrary precision addition and multiplication, tuned for highly optimized GPU performance.

Our work focused not only on the arithmetic details and technicalities of achieving a multiple-precision arithmetic, but also on its utility. We applied our theoretical results in two applications:

- in the context of dynamical systems, we search for periodic orbits in the Hénon map, which is a very numerically sensitive problem, using an apporach based on extensive long-term numerical iterations of the map;

- for semidefinite programming (SDP) solvers for numerically sensitive problems we integrated CAMPARY with the already existing SDPA solver, and we provide arbitrary precision not only for the CPU routines, but also for matrix multiplication with GPU support.

Despite our best efforts, and our reluctance to admit it, work is still to be done.

Perspectives and future work.

– Firstly, seeing the performance obtained when using specialized algorithms for double-word operations, a logical step would be to also design specialized algorithms for triple-word arithmetic. Triple-word arithmetic procedures are also useful for implementing correctly rounded elementary functions in CRLIBM.

– Also on the theoretical part of our work, we intend to continue the development of our library by also providing rigorous algorithms for elementary functions.

– One can remark that the proofs that come with this type of algorithms are very tedious and not easy to follow. A formal proof would inspire even more confidence in our algorithms. We already started a collaboration on this topic.

– On the application side of our work, as a first step we plan on continuing to develop the SDPA-CAMPARY package by also integrating the parallel algorithms into the matrix multiplication with GPU support routine. This would allow us to better test those algorithms and to take advantage of the data locality and the extra parallelization layer that they provide.

– Apart from the endless applications that could benefit from the use of the SDPA-CAMPARY package, we would be interested to tackle the kissing numbers problem.

Bibliography

[1] A. Abad, R. Barrio, and Á. Dena. Computing periodic orbits with arbitrary precision. *Phys. Rev. E*, 84:016701, Jul 2011.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754–1985, 1985.

[3] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Radix Independent Floating-Point Arithmetic*. ANSI/IEEE Standard 854–1987, 1987.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[5] D. H. Bailey and J. M. Borwein. High-precision arithmetic in mathematical physics. *Mathematics*, 3(2):337–367, 2015.

[6] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: an arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory, 2002. Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf`.

[7] D.H. Bailey, R. Barrio, and J.M. Borwein. High-precision computation: Mathematical physics and dynamics. *Applied Mathematics and Computation*, 218(20):10106 – 10121, 2012.

[8] M. Benedicks and L. Carleson. The dynamics of the Hénon map. *Annals of Mathematics*, 133(1):pp. 73–169, 1991.

[9] S. Boldo. Pitfalls of a full floating-point proof: example on the formal proof of the Veltkamp/Dekker algorithms. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 52–66, 2006.

[10] S. Boldo, M. Joldes, J.-M. Muller, and V. Popescu. Formal Verification of a Floating-Point Expansion Renormalization Algorithm. working paper or preprint, April 2017.

[11] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization methods and Software*, 11(1-4):613–623, 1999.

[12] B. Borchers. SDPLIB 1.2, a library of semidefinite programming test problems. *Optimization Methods and Software*, 11(1-4):683–690, 1999.

[13] K. Briggs. The `doubledouble` library, 1998. Available at `http://www.boutell.com/fracster-src/doubledouble/doubledouble.html`.

[14] N. Brisebarre, J.-M. Muller, and S.-K. Raina. Accelerating correctly rounded floating-point division when the divisor is known in advance. *IEEE Transactions on Computers*, 53(8):1069–1072, August 2004.

[15] P. E. Ceruzzi. The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262, 1981.

[16] S. Collange, M. Joldes, J.-M. Muller, and V. Popescu. Parallel floating-point expansions for extended-precision gpu computations. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 139–146, July 2016.

[17] M. Cornea, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton–Raphson-based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[18] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium®-based Systems*. Intel Press, Hillsboro, OR, 2002.

[19] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller. CR-LIBM, a library of correctly-rounded elementary functions in double-precision. Technical report, LIP Laboratory, Arenaire team, Available at https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf, December 2006.

[20] M. Daumas and C. Finot. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science*, 5(6):323–338, jun 1999.

[21] E. de Klerk and R. Sotirov. A new library of structured semidefinite programming instances. *Optimization Methods and Software*, 24(6):959–971, 2009.

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

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

[24] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. Technical report, Intel white paper, 2008.

[25] 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*, 33(2), 2007. available at http://www.mpfr.org/.

[26] Z. Galias and W. Tucker. Combination of exhaustive search and continuation method for the study of sinks in the Hénon map. In *Proc. IEEE Int. Symposium on Circuits and Systems, ISCAS'13*, pages 2571–2574, Beijing, May 2013.

[27] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[28] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991. An edited reprint is available at http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at http://www.validlab.com/goldberg/addendum.html.

[29] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2016. http://gmplib.org/.

[30] J. Harrison. A machine-checked theory of floating-point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, Berlin, September 1999.

[31] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2):139–174, 1996.

[32] M. Hénon. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50:69–77, 1976. 10.1007/BF01608556.

[33] D. Henrion, S. Naldi, and M. Safey El Din. SPECTRA - a Maple library for solving linear matrix inequalities in exact arithmetic. arXiv:1611.01947, 2016.

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

[35] Y. Hida, X.S. Li, and D.H. Bailey. C++/fortran-90 double-double and quad-double package, release 2.3.17. Accessible electronically at `http://crd-legacy.lbl.gov/~dhbailey/mpdist/`, March 2012.

[36] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 2nd edition, 2002.

[37] E. Hokenek, R. K. Montoye, and P. W. Cook. Second-generation RISC floating point with multiply-add fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, October 1990.

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

[39] C.-P. Jeannerod and S. M. Rump. On relative errors of floating-point operations: optimal bounds and applications. https://hal.inria.fr/hal-00934443, 2016.

[40] Claude-Pierre Jeannerod and Siegfried M. Rump. Improved error bounds for inner products in floating-point arithmetic. *SIAM Journal on Matrix Analysis and Applications*, 34(2):338–344, April 2013.

[41] M. Joldes, O. Marty, J.-M. Muller, and V. Popescu. Arithmetic algorithms for extended precision using floating-point expansions. *IEEE Transactions on Computers*, 65(4):1197–1210, 2016.

[42] M. Joldes, J.-M. Muller, and V. Popescu. On the computation of the reciprocal of floating point expansions using an adapted newton-raphson iteration. In *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 63–67, June 2014.

[43] M. Joldes, J.-M. Muller, and V. Popescu. Tight and Rigorous Error Bounds for Basic Building Blocks of Double-Word Arithmetic. working paper or preprint, July 2016.

[44] M. Joldes, J.-M. Muller, and V. Popescu. Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming. In *Arith24*, London, United Kingdom, July 2017.

[45] M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker. CAMPARY: cuda multiple precision arithmetic library and applications. In *Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, pages 232–240, 2016.

[46] M. Joldes, V. Popescu, and W. Tucker. Searching for sinks for the hénon map using a multiple-precision gpu arithmetic library. *ACM SIGARCH Computer Architecture News*, 42(4):63–68, December 2014.

[47] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

[48] W. Kahan. Lecture notes on the status of IEEE-754. PDF file accessible at `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`, 1996.

[49] W. Kahan. IEEE 754: An interview with William Kahan. *Computer*, 31(3):114–115, March 1998.

[50] W. Kahan. A logarithm too clever by half. Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT, 2004.

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

[52] I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[53] P. Kornerup, V. Lefevre, N. Louvet, and J.-M. Muller. On the computation of correctly rounded sums. *IEEE Transactions on Computers*, 61(3):289–298, 2012.

[54] J. Laskar and M. Gastineau. Existence of collisional trajectories of Mercury, Mars and Venus with the Earth. *Nature*, 459(7248):817–819, June 2009.

[55] M. Laurent. Strengthened semidefinite programming bounds for codes. *Mathematical programming*, 109(2-3):239–261, 2007.

[56] M. Laurent. Sums of squares, moment matrices and optimization over polynomials. In *Emerging applications of algebraic geometry*, pages 157–270. Springer, 2009.

[57] X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. Technical Report 45991, Lawrence Berkeley National Laboratory, 2000. http://crd.lbl.gov/~xiaoye/XBLAS.

[58] X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung, and D. J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, 2002.

[59] C. Lichtenau, S. Carlough, and S. M. Mueller. Quad precision floating point on the ibm z13. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 87–94, July 2016.

[60] S. Linnainmaa. Software for doubled-precision floating-point computations. *ACM Transactions on Mathematical Software*, 7(3):272–283, 1981.

[61] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[62] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[63] H. D. Mittelmann and F. Vallentin. High-accuracy semidefinite programming bounds for kissing numbers. *Experimental Mathematics*, 19(2):175–179, 2010.

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

[65] R. DC Monteiro. Primal–dual path-following algorithms for semidefinite programming. *SIAM Journal on Optimization*, 7(3):663–678, 1997.

[66] R. K. Montoye, E. Hokonek, and S. L. Runyan. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):59–70, 1990.

[67] R.E. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[68] 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.

[69] J.-M. Muller, V. Popescu, and P. T. P. Tang. A new multiplication algorithm for extended precision using floating-point expansions. *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, 00:39–46, 2016.

[70] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Springer, 2016.

[71] K. Nakata. The MPACK (MBLAS/MLAPACK) a multiple precision arithmetic version of BLAS and LAPACK. http://mplapack.sourceforge.net/, 2008–2012.

[72] M. Nakata. A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP,-QD and -DD. In *2010 IEEE International Symposium on Computer-Aided Control System Design*, pages 29–34. IEEE, 2010.

[73] M. Nakata, Y. Takao, S. Noda, and R. Himeno. A fast implementation of matrix-matrix product in double-double precision on nvidia c2050 and application to semidefinite programming. In *2012 Third International Conference on Networking and Computing*, pages 68–75, Dec 2012.

[74] T. Nakayama and D. Takahashi. Implementation of multiple-precision floating-point arithmetic library for GPU computing. In *Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS 2011, pages 343–349, December 2011.

[75] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[76] Y. Nievergelt. Analysis and applications of priest's distillation. *ACM Transactions on Mathematical Software*, 30(4):402–433, December 2004.

[77] NVIDIA. Kepler GK110 architecture. Technical report, NVIDIA Whitepaper, 2012.

[78] NVIDIA. *NVIDIA CUDA Programming Guide 8.0.61*. 2017.

[79] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[80] M. L. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, Philadelphia, PA, 2001.

[81] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144. IEEE Computer Society Press, Los Alamitos, CA, June 1991.

[82] D. M. Priest. *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, 1992.

[83] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11:1–16, 2005.

[84] S. M. Rump. Ultimately Fast Accurate Summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, January 2009.

[85] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[86] Boldo S. and G. Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.

[87] A. Schrijver. New code upper bounds from the terwilliger algebra and semidefinite programming. *IEEE Transactions on Information Theory*, 51(8):2859–2866, 2005.

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

[89] D. Simmons-Duffin. A Semidefinite Program Solver for the Conformal Bootstrap. *JHEP*, 06:174, 2015.

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

[91] J. F Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization methods and software*, 11(1-4):625–653, 1999.

[92] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 35. ACM, 2011.

[93] K.C. Toh, M.J. Todd, and R.H. Tutuncu. SDPT3 — a Matlab software package for semidefinite programming. *Optimization methods and software*, 11(1-4):545–581, 1999.

[94] G. W. Veltkamp. ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Technical Report 22, RC-Informatie, Technishe Hogeschool Eindhoven, 1968.

[95] G. W. Veltkamp. ALGOL procedures voor het rekenen in dubbele lengte. Technical Report 21, RC-Informatie, Technishe Hogeschool Eindhoven, 1969.

[96] M. Yamashita, K. Fujisawa, M. Fukuda, K. Kobayashi, K. Nakata, and M. Nakata. Latest developments in the SDPA family for solving large-scale SDPs. In *Handbook on semidefinite, conic and polynomial optimization*, pages 687–713. Springer, 2012.

[97] M. Yamashita, K. Fujisawa, and M. Kojima. Implementation and evaluation of sdpa 6.0 (semidefinite programming algorithm 6.0). *Optimization Methods and Software*, 18(4):491–505, 2003.

[98] T. J. Ypma. Historical development of the Newton-Raphson method. *SIAM Rev.*, 37(4):531–551, December 1995.

[99] G. Zbigniew and T. Warwick. Is the hénon attractor chaotic? *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(3):033102, 2015.

APPENDIX A

CAMPARY - Implementation details

As already mentioned, one of the main goals of this thesis was to develop a multiple-precision arithmetic software library. To this end we developed CAMPARY (CudA Multiple Precision ARithmetic LibrarY), an open source library distributed under the GNU General Public License as published by the Free Software Foundation, that is freely available at http://homepages.laas.fr/mmjoldes/campary/.

The library targets both CPU applications and applications deployed on NVIDIA GPU platforms (compute capability 2.0 or greater), this is why we provide two different versions:

- a C++ version, compilable with G++, that can be used on the CPU;
- a CUDA C version, compilable with NVCC, that can be used both on GPU [1] and CPU, except for the parallel algorithms (see Chapter 4).

For each of these versions we offer support for extending both the binary64 and the binary32 formats, even though the latter one is mostly useful for GPU use, since some architectures offer optimizations only for binary32. The constraint on the expansion size is given by the exponent range of the underlying floating-point format:

- for binary64 (exponent range $[-1022, 1023]$) the maximum expansion size is 39;
- for binary32 (exponent range $[-126, 127]$) the maximum expansion size is 12.

Since all these different versions are equivalent from a design and algorithmic point of view we will continue by detailing the binary64 GPU version.

The extended precision using classical sequential floating-point expansions (Chapter 3) is offered as a class called *multi_prec*; the parallel floating-point expansions (Chapter 4) are dealt with in a separate class that we are going to detail later on. One of the most important design decisions that we had to take was how to define the precision, i.e., the expansion size. We decided on using templates, because it offers flexibility to the implementation. The exact definition is:

```
template <int prec>
class multi_prec{
  private:
    double data[prec];
  public:
    ...
}
```

1. Compute capability 2.0 or greater for the sequential algorithms or at least 3.0 for the parallel ones.

For allocating (and initializing) a *multi_prec* object the user has the choice between six different constructors. Depending on ones choice the objects can be initialized with one or more values, a string or another *multi_prec* object:

```
__host__ __device__ multi_prec(){};
__host__ __device__ multi_prec(const double x);
__host__ __device__ multi_prec(const double *datap, const int precP);
__host__ __device__ multi_prec(const char *s);
template<int precP>
__host__ __device__ multi_prec(const multi_prec<precP> *mp);
__host__ __device__ multi_prec(const multi_prec<prec> *mp);
```

Member functions for random initialization with an ulp-nonoverlapping or $\mathcal{P}$-nonoverlapping expansion are also available along with "visualization" functions like pretty print.

In order to achieve robustness we encapsulate the private data array (that contains the floating-point expansion) with getters and setters that allow the user to manipulate it. The *prec* value is given as a template, so it cannot be changed throughout the execution; this is why only a getter is required for it. The declarations are as follows:

```
__host__ __device__ int getPrec() const;
__host__ __device__ const double* getData() const;
__host__ __device__ void setData(const double *datap, const int precp);
__host__ __device__ void setData(const double *datap);
__host__ __device__ void setElement(const double datap, const int index);
```

The code is also very flexible due to all the overloaded operators. The ones that we overload are:

- assignment operator, $=$, that can receive a binary64 value, another multi_prec object or a string of characters;
- unary $-$, that negates the data array;
- arithmetic operators, $+, -, *, /$, applied between two *multi_prec* objects or one *multi_prec* and one binary64;
- compound assignment operators $+=, -=, *=, /=$, applied as the arithmetic ones;
- relational/comparison operators, $==, !=, >, >=, <, <=$ (along with the *min* and *max* functions), applied as the arithmetic ones;
- subscript [] that gives access to one specific element of the data array;

The functions that implements the arithmetic operations are declared as templated friend functions[2], in order to keep the code of the class "clean". For each function that receives as input *multi_prec* objects we also include the equivalent with a *multi_prec* object and a binary64 value as input.

For some of the algorithms presented in Chapter 3 we mention that in practice we can make them faster by not considering the corner cases. Since fully certified algorithms usually come with a performance cost, we allow trade-off between proven output accuracy in the worst case versus highly efficient average case by offering two levels of algorithms. The library includes two separate files, that define the same class with the same functionalities, and the user has the choice of which one to use, depending on the time and accuracy constraints of one's problem.

2. Only the signature is included in the class, while the full body function is external.

1. The *multi_prec_certif.h* file implements all the algorithms as presented in Chapter 3, ensuring that the result is always an ulp-nonoverlapping expansion that conforms to the given bound. The declared functions are:

- *certifAddExpans*, implements Algorithm 27 without truncation on the input;
- *truncAddExpans*, implements Algorithm 27;
- *QD_LikeAddExpans*, implements Algorithm 29;
- *certifMulExpans*, implements Algorithm 31 without truncation of the partial products;
- *truncMulExpans*, implements Algorithm 31;
- *QD_LikeMulExpans*, implements Algorithm 33;
- *invExpans*, implements Algorithm 35;
- *divExpans*, implements division with Algorithm 35 followed by Algorithm 31;
- *invSqrtExpans*, implements Algorithm 38 for computing the reciprocal of the square root;
- *sqrtNewtonExpans*, implements square root with Algorithm 38 followed by Algorithm 31;
- *sqrtHeronExpans*, implements square root with Algorithm 39.

The arithmetic operators use the truncated versions of the algorithms.

For the special case of double-double numbers we use specialized templates in which we implemented the algorithms in Chapter 2 in oder of error bound tightness on the result.

2. The *multi_prec.h* file implements the "quick-and-dirty" level of the library. It uses algorithms which are faster, but do not consider accuracy issues for corner cases. In most cases the result is going to be the same as obtained when computing with the certified level, even the nonoverlapping condition can be achieved. The uncertainty appears if cancellation happens during intermediate computations, since this can generate intermediate 0s or even non-monotonic expansions in the result.

Even though some of the functions are implemented the same as in the certified level, the ones that are optimized are now used by the arithmetic operators. They are:

- *QD_LikeAddExpans*, implements Algorithm 29 with a faster renormalization step that uses Algorithm 24 with Fast2Sum;
- *QD_LikeMulExpans*, implements Algorithm 33, with the changed renormalization;
- *invExpans*, implements Algorithm 35 using the fast "quick-and-dirty" algorithms for intermediate computations;
- the same changes as above for *invSqrtExpans*, *sqrtNewtonExpans* and *sqrtHeronExpans*.

This level also comes with a code generation module, that allows for the user to code generate the algorithms function of the needed expansion size. This module provides increased performance by custom unrolling some complex loops (which are usually not optimized by GCC or NVCC compilers).

We recommend the use of this level if the performance requirements are strong, especially if there is the possibility of a-posteriori verification of the correctness of the numerical result.

Parallel expansions. As we mentioned, the parallel expansions are dealt with separately. In a separate file, *gpu_mprec.h*, we declare the simple (not templated) class:

```
class gpu_mprec{
  private:
```

```
    double val;
  public:
    ...
}
```

which appears from the point of the view of a single execution thread, this is why it stores only one value.

For an easy "switch" between the two classes we implemented load and store functions that can distribute a *multi_prec* object across threads and reform it, respectively. In more detail:

```
template <int prec>
__device__ gpu_mprec loadExpans(multi_prec<prec> const &mp){
  if (threadIdx.x < prec)
    return gpu_mprec(mp.getData()[threadIdx.x]);
  else
    return gpu_mprec(0.);
}

template <int prec>
__device__ void storeExpans(gpu_mprec gmp, multi_prec<prec> &mp){
  mp.setElement(gmp.getVal(), threadIdx.x);
}
```

Alongside two constructors , a getter and a setter, the class implements the three parallel algorithms presented in Chapter 4. These are parallelized using the x dimension of the thread block, allowing the user to also parallelize at a higher level, depending on the problem, using dimensions y and z. At a first glance this may seam difficult, but users that are familiar with CUDA programing can take advantage of these algorithms with minimum effort.

APPENDIX B

CAMPARY - Class code

B.1 *multi_prec* class code

```
#ifndef _multi_prec_h
#define _multi_prec_h

/**forward declarations**/
template <int prec> class multi_prec ;

/**template friends**/
template <int prec>
__host__ __device__ multi_prec<prec> abs(const multi_prec<prec> &mp);
template <int prec>
__host__ __device__ multi_prec<prec> sqrt(const multi_prec<prec> &mp);
template <int prec>
__host__ __device__ multi_prec<prec> invSqrt(const multi_prec<prec> &mp);

template <int prec>
__host__ __device__ void renorm(multi_prec<prec> &mp);
template <int prec>
__host__ __device__ void renorm_rand(multi_prec<prec> &mp);
template <int prec>
__host__ __device__ void renorm_2ndL(multi_prec<prec> &mp);

//******************** ADD ********************//
template <int pR, int p1, int p2>
__host__ __device__ void    certifAddExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void  certifAddExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const double val );
template <int pR, int p1, int p2>
__host__ __device__ void     truncAddExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void   truncAddExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const double val );
template <int pR, int p1, int p2>
__host__ __device__ void   QD_LikeAddExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void QD_LikeAddExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const double val );
```

```
//********************* MUL *********************//
template <int pR, int p1, int p2>
__host__ __device__ void    certifMulExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void  certifMulExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const double val );
template <int pR, int p1, int p2>
__host__ __device__ void     truncMulExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void   truncMulExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const double val );
template <int pR, int p1, int p2>
__host__ __device__ void   QD_LikeMulExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void QD_LikeMulExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                             const double val );

//********************* DIV *********************//
template <int pR, int p1>
__host__ __device__ void   invExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ void invExpans_d( multi_prec<pR> &res, const double val );
template <int pR, int p1, int p2>
__host__ __device__ void   divExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                      const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ void divExpans_d( multi_prec<pR> &res, const multi_prec<p1> &mp1,
                                      const double val );
template <int pR, int p1>
__host__ __device__ void divExpans_d( multi_prec<pR> &res, const double val,
                                      const multi_prec<p1> &mp1 );

//********************* SQRT *********************//
template <int pR, int p1>
__host__ __device__ void      invSqrtExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ void    invSqrtExpans_d( multi_prec<pR> &res, const double val );
template <int pR, int p1>
__host__ __device__ void   sqrtNewtonExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ void sqrtNewtonExpans_d( multi_prec<pR> &res, const double val );
template <int pR, int p1>
__host__ __device__ void    sqrtHeronExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ void  sqrtHeronExpans_d( multi_prec<pR> &res, const double val );

template <int prec>
class multi_prec{
private:
double data[prec];
public:
//---------constructors--------------
__host__ __device__ multi_prec(){};
__host__ __device__ multi_prec(const double x);
__host__ __device__ multi_prec(const double *datap, const int precp);
__host__ __device__ multi_prec(const char *s);
template<int precP>__host__ __device__ multi_prec(const multi_prec<precP> *mp);
__host__ __device__ multi_prec(const multi_prec<prec> *mp);
```

```
//----------geters & setters--------------
__host__ __device__ int getPrec() const;
__host__ __device__ const double* getData() const;
__host__ __device__ void setData(const double *datap, const int precp);
__host__ __device__ void setData(const double *datap);
__host__ __device__ void setElement(const double datap, const int index);

/**pretty print**/
__host__ __device__ void prettyPrint(){
  printf("Prec = %d\n", prec);
  for(int i=0; i<prec; i++) printf("   Data[%d] = %e\n",i,data[i]);
}
__host__ __device__ void prettyPrintBin(){
  printf("Prec = %d\n", prec);
  for(int i=0; i<prec; i++) printf("   Data[%d] = %a;\n",i,data[i]);
}
__host__ __device__ void prettyPrintBin_UnevalSum(){
  for(int i=0; i<prec-1; i++) printf("%a + ", data[i]);
  printf("%a;", data[prec-1]);
}

__host__ __device__ char* prettyPrintBF(){
  size_t needed = snprintf(NULL, 0, "%e", data[0]);
  if (prec>1)
    for(int i=1; i<prec; i++) needed += snprintf(NULL, 0, "%e ", data[i]);

  char *ch;
  ch =(char *) malloc((2*needed++)*sizeof(char));

  sprintf(ch, "%e ", data[0]);
  if (prec>1)
    for(int i=1; i<prec; i++) sprintf(&ch[strlen(ch)], "%e ", data[i]);
  sprintf(&ch[strlen(ch)], "%c", '\0');
  return ch;
}

//----------operators--------------
/** operator [] overloading  **/
__host__ __device__ double operator [](const int i)const {return data[i];}

/** Puts the value other in data[0] of the current multi_prec variable  **/
__host__ __device__ multi_prec& operator =(const double other){
  setData(&other, 1);
  return *this;
}
/** Transfers the multi_prec parameter other in the current multi_prec variable **/
__host__ __device__ multi_prec<prec>& operator =(const multi_prec<prec>& other){
  if(this != &other) setData(other.getData(), prec); /* check against self assingment */
  return *this;
}
template <int pS>
__host__ __device__ multi_prec& operator =(const multi_prec<pS>& other){
  setData(other.getData(),pS);
  return *this;
}
__host__ __device__ multi_prec& operator =(const char *s){
  if (read(s, *this)){
    printf("(qd_real::operator=): INPUT ERROR.");
    *this = 0.0;
  }
```

```
  return *this;
}

/** Equality overloading**/
template <int pS> __host__ __device__ bool operator ==(const multi_prec<pS> &mp2) const;
__host__ __device__ bool operator ==(const multi_prec<prec> &mp2) const;
__host__ __device__ bool operator ==(const double &mp2) const;
template <int pS> __host__ __device__ bool operator !=(const multi_prec<pS> &mp2) const;
__host__ __device__ bool operator !=(const multi_prec<prec> &mp2) const;
__host__ __device__ bool operator !=(const double &mp2) const;
template <int pS> __host__ __device__ bool operator < (const multi_prec<pS> &mp2) const;
__host__ __device__ bool operator < (const multi_prec<prec> &mp2) const;
__host__ __device__ bool operator < (const double &mp2) const;
template <int pS> __host__ __device__ bool operator <=(const multi_prec<pS> &mp2) const;
__host__ __device__ bool operator <=(const multi_prec<prec> &mp2) const;
__host__ __device__ bool operator <=(const double &mp2) const;
template <int pS> __host__ __device__ bool operator > (const multi_prec<pS> &mp2) const;
__host__ __device__ bool operator > (const multi_prec<prec> &mp2) const;
__host__ __device__ bool operator > (const double &mp2) const;
template <int pS> __host__ __device__ bool operator >=(const multi_prec<pS> &mp2) const;
__host__ __device__ bool operator >=(const multi_prec<prec> &mp2) const;
__host__ __device__ bool operator >=(const double &mp2) const;

/**Random expansion generation **/
__host__ __device__ void randomInit_P(){ genExpans_P<prec>(data);}
__host__ __device__ void randomInit_ulp(){ genExpans_ulp<prec>(data);}
__host__ __device__ void randomInit_ulp(int order){ genExpans_ulp<prec>(data, order);}
__host__ __device__ void randomInit_ulp(double first){ genExpans_ulp<prec>(data, first);}

/**Unary - overloading**/
__host__ __device__ friend multi_prec operator -(const multi_prec<prec> &mp1){
  multi_prec<prec> res;
  for(int i=0; i<prec; i++) res.data[i] = -mp1.getData()[i];
    return res;
}

/** operator += overloading  **/
template <int pS>
__host__ __device__ multi_prec<prec>& operator +=(const multi_prec<pS> &mp){
  QD_LikeAddExpans<prec,prec,pS>( *this, *this, mp );
  return *this;
}
__host__ __device__ multi_prec<prec>& operator +=(const double val){
  QD_LikeAddExpans_d<prec,prec>( *this, *this, val );
  return *this;
}

/** operator -= overloading  **/
template <int pS>
__host__ __device__ multi_prec<prec>& operator -=(const multi_prec<pS> &mp){
  QD_LikeAddExpans<prec,prec,pS>( *this, *this, -mp );
  return *this;
}
__host__ __device__ multi_prec<prec>& operator -=(const double val){
  QD_LikeAddExpans_d<prec,prec>( *this, *this, -val );
  return *this;
}

/** operator *= overloading  **/
template <int pS>
__host__ __device__ multi_prec<prec> operator *=(const multi_prec<pS> &mp){
```

```
  QD_LikeMulExpans<prec,prec,pS>( *this, *this, mp );
  return *this;
}
__host__ __device__ multi_prec<prec>& operator *=(const double val){
  QD_LikeMulExpans_d<prec,prec>( *this, *this, val );
  return *this;
}

/** operator /= overloading  **/
template <int pS>
__host__ __device__ multi_prec<prec>& operator /=(const multi_prec<pS> &mp){
  divExpans<prec,prec,pS>( *this, *this, mp );
  return *this;
}
__host__ __device__ multi_prec<prec>& operator /=(const double val){
  divExpans_d<prec,prec>( *this, *this, val );
  return *this;
}

/**absolute value **/
__host__ __device__ friend multi_prec<prec> abs<>(const multi_prec<prec> &mp);
__host__ __device__ friend multi_prec<prec> sqrt<>(const multi_prec<prec> &mp);
__host__ __device__ friend multi_prec<prec> invSqrt<>(const multi_prec<prec> &mp);

template<int pS> __host__ __device__ friend int read(const char *s, multi_prec<pS> &mp);

template <int pR, int p1, int p2>
__host__ __device__ friend multi_prec<pR> max( const multi_prec<p1> &mp1,
                                               const multi_prec<p2> &mp2 );
template <int pR, int p1, int p2>
__host__ __device__ friend multi_prec<pR> min( const multi_prec<p1> &mp1,
                                               const multi_prec<p2> &mp2 );

__host__ __device__ friend void renorm<>(multi_prec<prec> &mp);
__host__ __device__ friend void renorm_rand<>(multi_prec<prec> &mp);
__host__ __device__ friend void renorm_2ndL<>(multi_prec<prec> &mp);

/** operator + overloading  **/
template <int pS>
__host__ __device__ friend multi_prec operator +(const multi_prec<prec> &mp1,
                                                 const multi_prec<pS> &mp2){
  multi_prec<(prec>pS)?prec:pS> res(mp1.getData(), prec);
  return res += mp2;
}
__host__ __device__ friend multi_prec operator +(const multi_prec<prec> &mp1,
                                                 const double val){
  multi_prec<prec> res(mp1.getData(), prec);
  return res += val;
}
__host__ __device__ friend multi_prec operator +(const double val,
                                                 const multi_prec<prec> &mp1){
  multi_prec<prec> res(mp1.getData(), prec);
  return res += val;
}

/** operator - overloading  **/
template <int pS>
__host__ __device__ friend multi_prec operator -(const multi_prec<prec> &mp1,
                                                 const multi_prec<pS> &mp2){
  multi_prec<(prec>pS)?prec:pS> res(mp1.getData(), prec);
  return res -= mp2;
```

```
}
__host__ __device__ friend multi_prec operator -(const multi_prec<prec> &mp1, const double val){
  multi_prec<prec> res(mp1.getData(), prec);
  return res -= val;
}
__host__ __device__ friend multi_prec operator -(const double val, const multi_prec<prec> &mp1){
  multi_prec<prec> res(val);
  return res -= mp1;
}

/** operator * overloading  **/
template <int pS>
__host__ __device__ friend multi_prec operator *(const multi_prec<prec> &mp1,
                                                 const multi_prec<pS> &mp2){
  multi_prec<(prec>pS)?prec:pS> res(mp1.getData(), prec);
  return res *= mp2;
}
__host__ __device__ friend multi_prec operator *(const multi_prec<prec> &mp1, const double val){
  multi_prec<prec> res(mp1.getData(), prec);
  return res *= val;
}
__host__ __device__ friend multi_prec operator *(const double val, const multi_prec<prec> &mp1){
  multi_prec<prec> res(mp1.getData(), prec);
  return res *= val;
}

/** operator / overloading  **/
template <int pS>
__host__ __device__ friend multi_prec operator /(const multi_prec<prec> &mp1,
                                                 const multi_prec<pS> &mp2){
  multi_prec<(prec>pS)?prec:pS> res(mp1.getData(), prec);
  return res /= mp2;
}
__host__ __device__ friend multi_prec operator /(const multi_prec<prec> &mp1, const double val){
  multi_prec<prec> res(mp1.getData(), prec);
  return res /= val;
}
__host__ __device__ friend multi_prec operator /(const double val, const multi_prec<prec> &mp1){
  multi_prec<prec> res(val);
  return res /= mp1;
}

//******************** ADD ********************//
template <int pR, int p1, int p2>
__host__ __device__ friend void    certifAddExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void  certifAddExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );
template <int pR, int p1, int p2>
__host__ __device__ friend void     truncAddExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void   truncAddExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );
template <int pR, int p1, int p2>
__host__ __device__ friend void   QD_LikeAddExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void QD_LikeAddExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );
```

```
//********************* MUL *********************//
template <int pR, int p1, int p2>
__host__ __device__ friend void    certifMulExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void  certifMulExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );
template <int pR, int p1, int p2>
__host__ __device__ friend void    truncMulExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void   truncMulExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );
template <int pR, int p1, int p2>
__host__ __device__ friend void   QD_LikeMulExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void QD_LikeMulExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );

//********************* DIV *********************//
template <int pR, int p1>
__host__ __device__ friend void   invExpans( multi_prec<pR> &res, const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ friend void invExpans_d( multi_prec<pR> &res, const double val );
template <int pR, int p1, int p2>
__host__ __device__ friend void   divExpans( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const multi_prec<p2> &mp2 );
template <int pR, int p1>
__host__ __device__ friend void divExpans_d( multi_prec<pR> &res,
                                    const multi_prec<p1> &mp1, const double val );
template <int pR, int p1>
__host__ __device__ friend void divExpans_d( multi_prec<pR> &res,
                                    const double val, const multi_prec<p1> &mp1 );

//********************* SQRT *********************//
template <int pR, int p1>
__host__ __device__ friend void      invSqrtExpans( multi_prec<pR> &res,
                                                    const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ friend void    invSqrtExpans_d( multi_prec<pR> &res,  const double val );
template <int pR, int p1>
__host__ __device__ friend void   sqrtNewtonExpans( multi_prec<pR> &res,
                                                    const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ friend void sqrtNewtonExpans_d( multi_prec<pR> &res, const double val );
template <int pR, int p1>
__host__ __device__ friend void    sqrtHeronExpans( multi_prec<pR> &res,
                                                    const multi_prec<p1> &mp1 );
template <int pR>
__host__ __device__ friend void  sqrtHeronExpans_d( multi_prec<pR> &res, const double val );
};

#endif
```

B.2 *gpu_mprec* class code

```
#ifndef _gpu_mprec_h
#define _gpu_mprec_h

class gpu_mprec{
private:
  double val;
public:
//---------constructors--------------
__device__ gpu_mprec(){}
__device__ gpu_mprec(const double newVal):val(newVal){}

//----------geters & setters--------------
__device__ double getVal(){ return val; }
__device__ void setVal(const double newVal){ val = newVal; }

template <int prec>
__device__ friend gpu_mprec loadExpans(multi_prec<prec> const &mp);
template <int prec>
__device__ friend void storeExpans(gpu_mprec gmp, multi_prec<prec> &mp);

//----------friend functions--------------
template <int L, int K, int R>
__device__ friend gpu_mprec P_addExpans_safe(gpu_mprec x, gpu_mprec y);
template <int R>
__device__ friend gpu_mprec P_addExpans_quick(gpu_mprec x, gpu_mprec y);

template <int K, int L, int R>
__device__ friend gpu_mprec P_mulExpans(gpu_mprec x, gpu_mprec y);
};

#endif
```

APPENDIX C

Server configuration details

As the CPU we used an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz based on the Haswell architecture, which offers improved AVX[1] 2.0 instruction with floating-point FMA3 with up to twice the Flops per core (16 Flops/clock). The main features are given in Table C.1 and an excerpt of the result obtained using the command *more /proc/cpuinfo* is given in Figure C.1.

On the GPU side we had two NVIDIA Tesla K20Xm cards with Kepler GK110 architecture. Some important features are given in Tables C.2 and C.3 and an excerpt of the result obtained using the command *nvidia-smi -i 0 -q* is given in Figure C.2.

The software configuration was as follows:

- Debian 4.9.2-10 GNU/Linux 8.2 operating system with 3.16.0-4-amd64 kernel;
- compilers GCC and G++ 4.9.2;
- CUDA 7.5 toolkit with NVCC V7.5.17.

1. Advanced Vector Extensions.

Table C.1 – Main features of the Intel(R) Xeon(R) CPU E5-2695 v3. [2]

# of cores	14
# of threads	28
Processor Base Frequency	2.3 GHz
Max Turbo Frequency	3.3 GHz
L3 Cache	35 MB SmartCache
Max Memory Size	768 GB
Memory Types	DDR4 1600/1866/2133

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz
stepping        : 2
microcode       : 0x29
cpu MHz         : 2300.000
cache size      : 35840 KB
physical id     : 0
siblings        : 14
core id         : 0
cpu cores       : 14
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic
movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm ida arat epb xsaveopt
pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2
erms invpcid
bogomips        : 4600.23
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
```

Figure C.1 – Excerpt of the result obtained using the command *more /proc/cpuinfo*.

2. https://ark.intel.com/products/81057/Intel-Xeon-Processor-E5-2695-v3-35M-Cache-2_30-GHz

Table C.2 – Main features of the NVIDIA Tesla K20Xm card. [3]

# of SMX Units	14
# of CUDA Cores	2688
GPU Base Clock	732 MHz
GPU Boost Support	Limited
GPU Boost Clocks	758 MHz 784 MHz
Peak Single Precision	3.95 TFlops
Peak Double Precision	1.32 TFlops
Onboard GDDR5 Memory	6 GB
Memory Bandwidth	250 GB/s
Memory Clock	2600 MHz

Table C.3 – Execution features of the NVIDIA Tesla K20Xm card.

Compute Capability	3.5
Threads per Warp	32
Max Warps per SM	64
Max Threads per SM	2048
Max Thread Blocks per SM	16
32-bit Registers per SM	64 K
Max Registers per Thread Block	64 K
Max Registers per Thread	255
Max Threads per Thread Block	1024
Shared Memory an L1 Cache	64 KB total
Max Shared Memory per Thread Block	48 KB

3. https://www.microway.com/knowledge-center-articles/in-depth-comparison-of-nvidia-tesla\-kepler-gpu-accelerators/#tabs-3-2

```
Driver Version                      : 361.45.18
Attached GPUs                       : 2
GPU 0000:03:00.0
    Product Name                    : Tesla K20Xm
    Accounting Mode Buffer Size     : 1920
    Serial Number                   : 0324114085169
    GPU UUID                        : GPU-e35294d2-1252-6a68-aae9-72b8f039b90b
    VBIOS Version                   : 80.10.39.00.13
    MultiGPU Board                  : No
    Board ID                        : 0x300
    GPU Part Number                 : 900-22081-0130-000
    GPU Operation Mode
        Current                     : Compute
        Pending                     : Compute
    Performance State               : P0
    Clocks Throttle Reasons
        Idle                        : Not Active
        Applications Clocks Setting : Active
        SW Power Cap                : Not Active
        HW Slowdown                 : Not Active
        Sync Boost                  : Not Active
        Unknown                     : Not Active
    FB Memory Usage
        Total                       : 5759 MiB
        Used                        : 12 MiB
        Free                        : 5747 MiB
    Power Readings
        Power Management            : Supported
        Power Draw                  : 59.07 W
        Power Limit                 : 235.00 W
        Default Power Limit         : 235.00 W
        Enforced Power Limit        : 235.00 W
        Min Power Limit             : 150.00 W
        Max Power Limit             : 235.00 W
    Clocks
        Graphics                    : 732 MHz
        SM                          : 732 MHz
        Memory                      : 2600 MHz
        Video                       : 540 MHz
    Applications Clocks
        Graphics                    : 732 MHz
        Memory                      : 2600 MHz
    Default Applications Clocks
        Graphics                    : 732 MHz
        Memory                      : 2600 MHz
    Max Clocks
        Graphics                    : 784 MHz
        SM                          : 784 MHz
        Memory                      : 2600 MHz
        Video                       : 540 MHz
```

Figure C.2 – Excerpt of the result obtained using the command *nvidia-smi -i 0 -q*.