Accuracy Versus Time: A Case Study with Summation Algorithms*

Philippe Langlois
Equipe de Recherche DALI
Université de Perpignan
52 Avenue Paul Alduy
66860 Perpignan, France
langlois@univ-perp.fr

Matthieu Martel
Equipe de Recherche DALI
Université de Perpignan
52 Avenue Paul Alduy
66860 Perpignan, France
matthieu.martel@univ-perp.fr

Laurent Thévenoux
Equipe de Recherche DALI
Université de Perpignan
52 Avenue Paul Alduy
66860 Perpignan, France
laurent.thevenoux@univ-perp.fr

ABSTRACT

In this article, we focus on numerical algorithms for which, in practice, parallelism and accuracy do not cohabit well. In order to increase parallelism, expressions are reparsed, implicitly using mathematical laws like associativity, and this reduces the accuracy. Our approach consists in focusing on summation algorithms and in performing an exhaustive study: we generate all the algorithms equivalent to the original one and compatible with our relaxed time constraint. Next we compute the worst errors which may arise during their evaluation, for several relevant sets of data. Our main conclusion is that relaxing very slightly the time constraints by choosing algorithms whose critical paths are a bit longer than the optimal makes it possible to strongly optimize the accuracy.

We extend these results to the case of bounded parallelism and to accurate sum algorithms that use compensation techniques.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Optimization; G.1.0 [**Mathematics of Computing**]: Numerical Analysis, Computer Arithmetic; I.2.2 [**Automatic Programming**]: Program transformation.

General Terms

Algorithms, Design, Experimentation, Performance, Reliability.

Keywords

Parallelism, Summation, Floating-point numbers, Precision.

*This work was partly supported by the project "Compil'HD" of Région Languedoc-Roussillon ("Chercheur d'Avenir" programme) and by the ANR project "EvaFlo".

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASCO 2010, 21–23 July 2010, Grenoble, France.

Copyright 2010 ACM 978-1-4503-0067-4/10/0007 ...$10.00.

1. INTRODUCTION

Symbolic-numeric algorithms have to manage the *a priori* conflicting numerical accuracy and computing time. Performances and accuracy of basic numerical algorithms for scientific computing have been widely studied, as for example the central problem of summing floating point values – see the numerous references in [5] or more recently in [20, 14, 19]. Instruction level parallelism is commonly used to speed-up these implementations during compilation steps. One could also expect that compilers improve accuracy

However as already noticed by J. Demmel [2], in practice parallelism and accuracy do not cohabit well. To exploit the parallelism within an expression, this one is reparsed implicitly using mathematical laws like associativity. The new expression is then more balanced to benefit for as much parallelism as possible. In our scope, such re-writing should yield algorithms that sum n numbers in a logarithmic time $O(\log n)$. The point is that the numerical accuracy of some algorithms is strongly sensitive to reparsing. In IEEE754 floating-point arithmetic, additions are not associative and, in general, most algebraic laws like associativity and distributivity do not hold any longer. As a consequence, while increasing the parallelism of some expression, its numerical accuracy may decrease and, conversely, improving the accuracy of some computation may reduce its parallelism. Moreover, in architectures, instruction level parallelism is bounded and it may possible to execute an algorithm less parallel than the optimum in the same (or very similar) execution time.

In this article, we address the following question: *How can we improve the accuracy of numerical summation algorithms if we relax slightly the performance constraints?* More precisely, we examine how accurate can be algorithms which are k times less efficient than the optimal one or with a constant overhead with respect to the optimal one, e.g. for the summation of n values, in $k \times \lfloor \log(n) \rfloor$ or $k + \lfloor \log(n) \rfloor$ for a constant parameter k.

For example, let us consider the sum

$$s = \sum_{i=1}^{N} a_i, \text{ with } a_i = \frac{1}{2^i}, \ 1 \le i \le N \qquad (1)$$

Two extreme algorithms compute s

$$s_1 := \Big(\big((a_1 + a_2) + a_3\big) + \dots a_{N-1} \Big) + a_N \qquad (2)$$

and, assuming $N = 2^k$,

$$s_2 := \Big(\big((a_1 + a_2) + (a_3 + a_4)\big) + \ldots + (a_{\frac{N}{2}-1} + a_{\frac{N}{2}}) \Big) +$$

$$\Big(\big((a_{\frac{N}{2}+1} + a_{\frac{N}{2}+2}) + (a_{\frac{N}{2}+3} + a_{\frac{N}{2}+4})\big) + \ldots + (a_{N-1} + a_N) \Big) \quad (3)$$

Clearly, the sum s_1 is computed sequentially while s_2 corresponds to a reduction which can be computed in logarithmic time. However, in double precision, we have, for $N = 10$:

$$s = 0.9990234375 \quad s_1 = 0.9990234375 \quad s_2 = 0.99609375$$

and it happens that s_1 is far more precise than s_2.

Our approach consists in performing an exhaustive study. First we generate all the algorithms equivalent to the original one and compatible with our relaxed time constraint. Then we compute the worst errors which may arise during their evaluation for several relevant sets of data. Our main conclusion is that relaxing very slightly the time constraints by choosing algorithms whose critical paths are a bit longer than the optimal one makes it possible to strongly optimize the accuracy. This matter of fact is illustrated using various datasets, most of them being ill-conditioned. We extend these results to the case of bounded parallelism and to compensated algorithms. For bounded parallelism we show that more accurate algorithms whose critical path is not optimal can be executed in as many cycles as optimal algorithms, e.g. on VLIW [7] architectures. Concerning compensation, we show that elaborated summation algorithms can be discovered automatically by inserting systematically compensations and then reparsing the resulting expression.

This article is organized as follows. Section 2 gives an overview of summation algorithms. It also introduce our technique to bound error terms. Section 3 presents our main results concerning the time versus precision compromise. Section 4 describes how we generate exhaustively the summation algorithms of interest and Section 5 introduces further examples involving larger sums, accuracy versus bounded parallelism and compensated sums. Finally, some perspectives and concluding remarks are given in Section 6.

2. BACKGROUND

In floating-point arithmetic accuracy is a critical matter, for scientific computing as well as for critical embedded systems [11, 12, 3, 7, 6, 4]. Famous examples alas illustrate that bad accuracy can cause human damages [13] and money loses [10]. If accuracy is critical so is parallelism but usually these two domains are considered separately. While focusing on summation, this section compares the most well-known algorithms with respect to their accuracy and parallelism characteristics.

In Subsection 2.1 we recall background material on summation algorithms [5, 14] and we explain how we measure the error terms in Subsection 2.2.

2.1 Summation Algorithms

Summation in floating-point arithmetic is a very rich research domain. There are various algorithms that improve accuracy of a sum of two or more terms and similarly, there are many parallel summation algorithms.

2.1.1 Two Extreme Algorithms for Parallelism

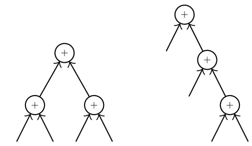


Figure 1: Graphical representation of Algorithm 1 and Algorithm 2 by dataflow graphs.

Basically, there are two extreme algorithms with respect to parallelism properties to compute the sum of $(n+1)$ terms. The first following algorithm is fully sequential whereas the second one benefits from the maximum degree of parallelism.

- **Algorithm 1** is the extreme sequential algorithm. It computes a sum in $O(n)$ operations successively summing the $n+1$ floating-point numbers (see Equation (2)).
- Pairwise summation **Algorithm 2** is the most parallel algorithm. It computes a sum in $O(\log(n))$ successive stages (see Equation (3)).

These algorithms are represented by dataflow graphs in Figure 1.

Algorithm 1 Sum: Summation of $n+1$ Floating-Point Numbers

Input: p is (a vector of) $n+1$ floating-point numbers
Output: s_n is the sum of p
 $s_0 \leftarrow p_0$
 for $i = 1$ to n **do**
  $s_i \leftarrow s_{i-1} \oplus p_i$
 end for

Algorithm 2 SumPara: Parallel Summation of $n+1$ Floating-Point Numbers

Input: $p[l:r]$ is (a vector of) $n+1$ floating-point numbers
Output: the sum of p
 $m \leftarrow \lfloor (l+r)/2 \rfloor$
 if $l = r$ **then**
  return p_l
 else
  return SumPara$(p[l:m]) \oplus$ SumPara$(p[m+1:r])$
 end if

Mixing Algorithm 1 and Algorithm 2 yields many algorithms of parallelism degrees between those two extreme ones.

2.1.2 Merging Parallelism and Accuracy

It is well known that these two extreme algorithms do not verify the same worst case error bounds [5]. Nevertheless to improve the accuracy of one computed sum, it is usual to sort the terms according to some of their characteristics (increasingly, decreasingly, negative or positive sort, etc.).

Summation accuracy varies with the order of the inputs. Increase or decrease orders of the absolute values of the operands are the first two choices for the simplest Algorithm

1. If the inputs are both negative and positive, the decrease order is better, otherwise other orders are equivalent. If all the inputs are of the same sign, the increase order is more interesting than others [5]. More dynamic inserting methods consist in sorting the inputs (in a given order), in summing the first two numbers and in inserting the result within the inputs conserving the initial order. Such sorting is more difficult to implement while conserving the parallelism level of Algorithm 2.

2.1.3 More Accuracy with Compensation

A well known and efficient techniques to improve accuracy is compensation which uses some of the following error-free transformations [14].

Algorithm 3 computes the sum of two floating-point number $x = a \oplus b$ and the absolute error y due to the IEEE754 arithmetic [1].

Algorithm 3 TwoSum, Result and Absolute Error in Summation of Two Floating-Point Numbers (Introduced by Knuth [8])

Input: a and b, two floating-point numbers
Output: $x = a \oplus b$ and y the absolute error on x
  $x \leftarrow a \oplus b$
  $z \leftarrow x \ominus a$
  $y \leftarrow (a \ominus (x \ominus z)) \oplus (b \ominus z)$

When $|a| \geq |b|$ Algorithm 4 is faster than Algorithm 3. Obviously it will be necessary to check this condition to apply it. The overcost of such practice on modern computing environments is not so clear [19, 9]. In both cases the key point is the error-free transformation $x + y = a + b$.

Algorithm 4 FastTwoSum, Result and Absolute Error in Summation of Two Floating-Point Numbers

Input: a and b two floating-point numbers such that $|a| \geq |b|$
Output: $x = a \oplus b$ and y the absolute error on x
  $x \leftarrow a \oplus b$
  $y \leftarrow (a \ominus x) \oplus b$

To improve the accuracy of Algorithm 1, VecSum Algorithm applies this error-free transformation. Algorithm 6 uses this error-free vector transformation and yields a twice more accurate summation algorithm [14]. Hence Sum2 computes every rounding error y and adds them together before compensating the classic Sum computed result. In other words, Sum Algorithm applies twice, once to the $n+1$ summand and then once to the n error terms, the compensated summation being the last addition between these two values.

Algorithm 5 VecSum, Error-Free Vector Transformation of $n+1$ Floating-Point Numbers [14]

Input: p is (a vector of) $n+1$ floating-point numbers
Output: p_n is the approximate sum of p, $p[0:n-1]$ is (a vector of) the generated errors
  for $i = 1$ to n **do**
    $[p_i, p_{i-1}] \leftarrow TwoSum(p_i, p_{i-1})$
  end for

Algorithm 6 Sum2, Compensated Summation of $n+1$ Floating-Point Numbers

Input: p is (a vector of) $n+1$ floating-point numbers
Output: s the sum of p
  $p \leftarrow VecSum(p)$
  $e \leftarrow Sum(i = 0, n-1, p[i])$
  $s \leftarrow p_n \oplus e$

These error-free transformations have been used differently within several other accurate summation algorithm. Previous Sum2 was also considered by [15]. A slight variation is the famous Kahan compensated summation: in Algorithm 7, every rounding error e is added to the next summand (the compensating step) before adding it to the previous partial sum.

It exists many other algorithms for accurate summation that use these error-free transformations, as for example Priest double-compensated summation [16] or the recursive SumK algorithms of [14] or also the very fast AccSum and PrecSum of [19]. We do not detail these any longer.

Algorithm 7 SumComp, Compensated Summation of n Floating-Point Numbers (Kahan [5])

Input: p is (a vector of) $n+1$ floating-point numbers
Output: s the sum of input numbers
  $s \leftarrow p_0$
  $s \leftarrow 0$
  for $i = 1$ to n **do**
    $tmp \leftarrow s$
    $y \leftarrow p_i \oplus e$
    $s \leftarrow tmp \oplus y$
    $e \leftarrow (tmp \ominus s) \oplus y$
  end for

2.2 Measuring the Error Terms

Let x and y be two real numbers approximated by floating-point numbers $\hat{x}$ and $\hat{y}$ such that $x = \hat{x} + \epsilon_x$ and $y = \hat{y} + \epsilon_y$ for some error terms $\epsilon_x \in \mathbb{R}$ and $\epsilon_y \in \mathbb{R}$. Let us consider the sum $S = x + y$. In floating-point arithmetic this sum is approximated by

$$\hat{S} = \hat{x} \oplus \hat{y}$$

where $\oplus$ denotes the floating-point addition. We write the difference ϵ_S between S and $\hat{S}$ as in [21],

$$\epsilon_S = S - \hat{S} = \epsilon_x + \epsilon_y + \epsilon_+, \quad (4)$$

where ϵ_+ denotes the round-off error introduced by the operation $\hat{x} \oplus \hat{y}$ itself.

In the rest of this article, we use intervals $\mathbf{x},\ \mathbf{y}, \ldots$ instead of floating-point numbers $\hat{x},\ \hat{y}, \ldots$ as well as for the error terms $\epsilon_x,\ \epsilon_y, \ldots$ for the next two different reasons.

(i) Our long-term objective is to perform program transformations at compile-time [12] to improve the numerical accuracy of mathematical expressions. It comes out that our transformations have to improve the accuracy of any dataset or, at least, of a wide range of datasets. So we consider inputs belonging to intervals.

(ii) The error terms are real numbers, not necessarily representable by floating-point numbers as ϵ_S in Equation

(4). We approximate them by intervals, using rounding modes towards outside. Clearly, the towards outside rouding mode correspond, in this case, at the rounding mode towards $-\infty$ for the lower bound of the interval and towards $+\infty$ for the upper bound.

An interval $\mathbf{x}$ with related interval error $\epsilon_{\mathbf{x}}$ denotes all the floating-point numbers $\hat{x} \in \mathbf{x}$ with a related error $\epsilon_x \in \epsilon_{\mathbf{x}}$. This means that the pair $(\mathbf{x}, \epsilon_{\mathbf{x}})$ represents the set X of exact results:

$$X = \{x \in \mathbb{R} \ : \ x = \hat{x} + \epsilon_x, \ \hat{x} \in \mathbf{x}, \ \epsilon_x \in \epsilon_{\mathbf{x}}\}.$$

Let $\mathbf{x}$ and $\mathbf{y}$ be two sets of floating-point numbers with error terms belonging to the intervals $\epsilon_{\mathbf{x}} \subseteq \mathbb{R}$ and $\epsilon_{\mathbf{y}} \subseteq \mathbb{R}$. We have

$$\mathbf{S} = \mathbf{x} \oplus_I \mathbf{y} \tag{5}$$

where $\oplus_I$ is the sum of intervals with the same rounding mode than $\oplus$ (generally to the nearest) and

$$\epsilon_{\mathbf{S}} = \epsilon_{\mathbf{x}} \oplus_O \epsilon_{\mathbf{y}} \oplus_O \epsilon_+ \tag{6}$$

where $\oplus_O$ denotes the sum of intervals with rounding mode towards outside. Per example:

$$\big([\underline{x}, \overline{x}]; [\underline{\epsilon_x}, \overline{\epsilon_x}]\big) + \big([\underline{y}, \overline{y}]; [\underline{\epsilon_y}, \overline{\epsilon_y}]\big) =$$

$$\big([\underline{x} +_{-\infty} \underline{y}, \overline{x} +_{+\infty} \overline{y}]; [\underline{\epsilon_x} +_{-\infty} \underline{\epsilon_y}, \overline{\epsilon_x} +_{+\infty} \overline{\epsilon_y}]\big)$$

In addition, ϵ_+ denotes the round-off error introduced by the operation $\hat{x} \oplus_I \hat{y}$. Let $ulp(x)$ denote the function which computes the unit in the last place of x [5], i.e. the weight of the least significant digit of x and let $S = [\underline{S}, \overline{S}]$. We bound ϵ_+ by the interval $[-u, u]$ by:

$$u = \frac{1}{2}\max(ulp(|\underline{S}|), ulp(|\overline{S}|)).$$

Using the notations of equations (4), (5) and (6), it follows that for all $\hat{x} \in \mathbf{x}$, $\epsilon_x \in \epsilon_{\mathbf{x}}$, $\hat{y} \in \mathbf{y}$, $\epsilon_y \in \epsilon_{\mathbf{y}}$

$$S \in \mathbf{S} \text{ and } \epsilon_S \in \epsilon_{\mathbf{S}}.$$

3. NUMERICAL ACCURACY OF NON-TIME-OPTIMAL ALGORITHMS

The aim of this section is to show how we can improve accuracy while relaxing the time constraints. In Subsection 3.1, we illustrate our approach using as an example a sum of random values. We generalize our results to some significant sets of data in Subsection 3.2.

3.1 The General Approach

In order to evaluate the algorithms to compute one sum expression, associativity and distributivity are only needed hereafter. Basically, while in exact arithmetic all the algorithms are numerically equivalent, in floating-point arithmetic the story is not the same. Indeed, many things may arise like absorption, rounding errors, overflow, etc. and then floating-point algorithms return various different results.

One mathematical expression yields a huge amount of evaluation schemes. We propose to analyse this huge set of algorithms with respect to accuracy and parallelism. First we search the most accurate algorithms among all levels of parallelism, and then we search among them the ones with the best degrees of parallelism. We aim at finding the more interesting ratio between accuracy and parallelism.

In this section, we use random data (generated using an uniform random distribution) defined as interval $[\underline{a}, \overline{a}]$. We measure the interval that represents the maximum error bound $[\overline{e}, \underline{e}]$ applying the previously described error model. Let $\mathbf{a}_i = [\underline{a_i}, \overline{a_i}], 1 \le i \le n$. This means that for all $a_1 \in \mathbf{a}_i, \ldots, a_n \in \mathbf{a}_n$, the error on $\Sigma_1^n a_i$ belongs to $[\underline{e}, \overline{e}]$. We focus the maximum error which is defined as $max(|\underline{e}|, |\overline{e}|)$. Algorithms which have the smaller maximum error are called optimal algorithms. This maximum error is a pertinant optimization criteria in the compilation domain. With this criteria we want to guarantee the maximum error which can arise during any execution of a program.

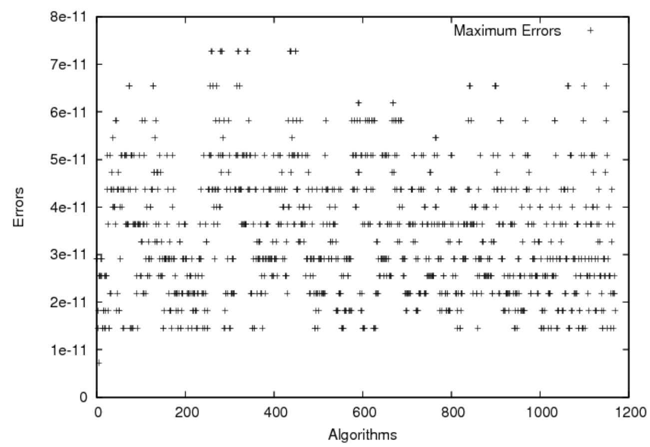


Figure 2: Maximum errors for each algorithms for a six terms summation reparsings.

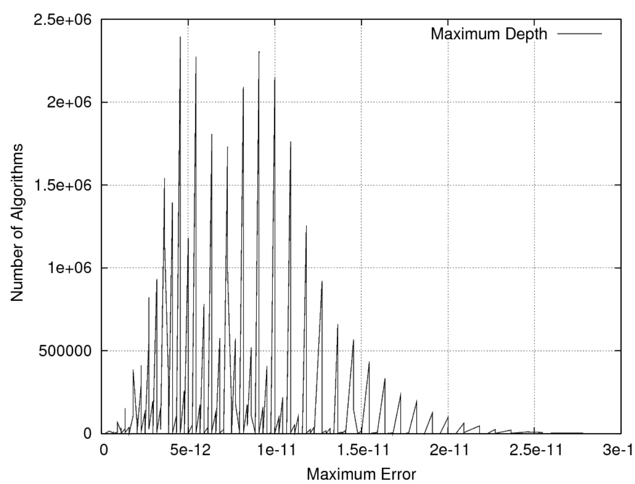


Figure 3: Error repartition when summing ten terms.

Each dot of Figure 2 shows the absolute error of every algorithms, i.e. every parsing of the summing expression with six terms. X-axis represents the algorithms numbered from 0 to 1,170 and Y-axis represents the maximal absolute error which can be encounter during the algorithm evaluation. It is not a surprise that errors are not uniformly distributed and that the errors belong to a small number of stages. Figure 3 shows the distribution of the errors for the different stages of a ten terms summation. The proportion of algorithms with very few small or very large errors is small. Most of the algorithms present an average accuracy between small and large errors. We guess that it will be difficult to find the best accurate algorithms (as well as the worst one), most having an average accuracy.

It exists 46,607,400 different algorithms for an expression of ten terms. Among this huge set, many of them are sequential or almost sequential. So we propose to restrict the search to a certain level of parallelism. Let n be the number of additions and k a constant chosen arbitrarily e.g. here $k = 2$. In the following of this article, if it is not precisely defined, we sum ten terms and k is equal to two. We restrict our search of accurate algorithms within three included sets: algorithms having a computing tree of height smaller or equal to $\lfloor \log(n) \rfloor + 1$, $\lfloor \log(n) \rfloor + k$ and $k \times \lfloor \log(n) \rfloor$. Using these restrictions, there are 27,102,600 algorithms of level $k \times \lfloor \log(n) \rfloor$, 13,041,000 algorithms of level $\lfloor \log(n) \rfloor + k$ and 2,268,000 algorithms of level $\lfloor \log(n) \rfloor + 1$.

Results are given in Figure 4 and in Table 1. We observe that the highest level of parallelism, the level $\lfloor \log(n) \rfloor + 1$, does not allow us to compute the most accurate results. Nevertheless if we use a less high but still reasonable level of parallelism, e.g. levels $O(\lfloor \log(n) \rfloor + k)$ or $O(k \times \lfloor \log(n) \rfloor)$, we can compute accurate results.

The more the level of parallelism is, the harder it is to find the more accurate algorithms among all of them. In tables 2 and 3 we observe that the level $\lfloor \log(n) \rfloor + k$ presents a better proportion of accurate algorithms (stages with small numbers) than the higher parallelism level $k \times \lfloor \log(n) \rfloor$. Moreover the most accurate algorithms within the first set are less accurate than the ones of the second set — see Figure 4.

Parallelism	Error of Optimal Algorithm	Percent
no parallelism	$2.273e^{-13}$	0.006
$\lfloor \log(n) \rfloor + 1$	$4.547e^{-13}$	0.007
$\lfloor \log(n) \rfloor + k$	$2.273e^{-13}$	0.006
$k \times \lfloor \log(n) \rfloor$	$2.273e^{-13}$	0.007

Table 1: Optimal error value and percentage of algorithms reaching this precision.

3.2 Larger Experiments

We study a more representative sets of data using various kinds of values chosen as well-known error-prone problems, i.e. ill-conditioned set of summands. The condition number for computing $s = \sum_{i=1}^{N} x_i$, is defined as following,

$$cond(s) = \frac{\sum_{i=1}^{N} |(x_i)|}{|s|}.$$

The larger this number is, the more ill-conditioned the summations are, the less the result is accurate.

Summation suffers from the two following problems:

- Absorption arises when adding a small and a large values. The smallest values are absorbed by the largest ones. In our context (IEEE-754 double precision): $10^{16} \oplus 10^{-16} = 10^{16}$. In general absorption is not so dangerous while adding values of the same sign: its condition number equals roughly one. Nevertheless a large amount of small errors accumulates in large summations — this was the case in the well known Patriot Missile failure [13].
- Cancellation arises when absorption appears within data with different sign. In this case, the condition number can be arbitrarily large. We will call such case as summation with ill-conditioned data. In our context an example is : $(10^{16} \oplus 10^{-16}) \ominus 10^{16} = 0$.

Stage	Example of expression	%
1	$(i+(f+g))+((c+d)+((h+j)+(e+(a+b))))$	0.006
2	$(i+(f+g))+(j+((c+d)+((e+h)+(a+b))))$	0.024
3	$(i+(f+g))+(j+((e+(a+h))+(b+(c+d))))$	0.001
⋮	⋮	⋮
141	$(j+((c+g)+(b+h)))+(e+(a+(d+(f+i))))$	0.001
142	$(j+(h+(g+(c+e))))+(b+(a+(d+(f+i))))$	0.005
143	$(j+(h+(e+(c+g))))+(b+(a+(d+(f+i))))$	0.002

Table 2: Repartition of the algorithms according to their precision at the parallelism level $O(\lfloor \log(n) \rfloor + k)$ on ten terms summation (stages with small numbers are the smallest errors).

Stage	Example of expression	%
1	$(i+(f+g))+((c+d)+((h+j)+(e+(a+b))))$	0.008
2	$(i+(f+g))+(j+((c+d)+(h+(e+(a+b)))))$	0.039
3	$(i+(f+g))+(j+((e+(a+h))+(b+(c+d))))$	0.004
⋮	⋮	⋮
171	$(j+(g+(b+h)))+(e+(c+(a+(d+(f+i)))))$	0.007
172	$(j+(h+(e+g)))+(c+(b+(a+(d+(f+i)))))$	0.015
173	$(j+(h+(c+g)))+(e+(b+(a+(d+(f+i)))))$	0.001

Table 3: Repartition of the algorithms according to their precision at the parallelism level $O(k \times \lfloor \log(n) \rfloor)$ on ten terms summation (stages with small numbers are the smallest errors).

We introduce 9 datasets to generate different types of absorptions and cancellations. These two problems are clear with scalar values. So we first use intervals with small variations around such scalar values. Every dataset is composed of ten samples that share the same numerical characteristics. We recall that these experiments are limited to ten summands. In the following, we say that a floating-point value is a small, medium or large when it is, respectively, of the order of 10^{-16}, 1 and 10^{16}. This is justified in double

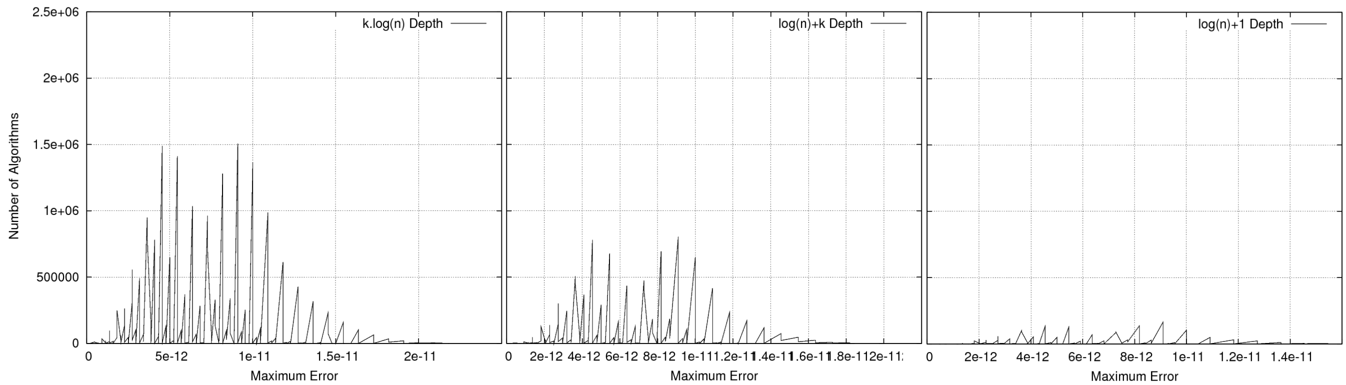


Figure 4: Error repartition with three different degrees of parallelism for ten terms summation.

precision IEEE-754 arithmetic.

- **Dataset 1**. Positive sign, 20% of large values among small values. There are absorptions and accurate algorithms should first sum the smallest terms (increasing order).
- **Dataset 2**. Negative sign, 20% of large values among small values. Results should be the same as in Dataset 1.
- **Dataset 3**. Positive sign, 20% of large values among small and medium values. The best results should be obtained with algorithms which sum in increase order.
- **Dataset 4**. Negative sign, 20% of large values among small and medium values. Results should be equivalent to the results of Dataset 3.
- **Dataset 5**. Both signs, 20% of large values that cancel, among small values. The most accurate algorithms should sum the two largest values first. In a more general case, the best algorithms should sum in decrease order of absolute values. It is a classic ill-conditioned summation.
- **Dataset 6**. Both signs, few small values and same proportion of large and medium values. Only large values cancel. The best algorithms should sum in decrease order of absolute values.
- **Dataset 7**. Both signs, few small values and same proportion of large and medium values. Large and medium values are ill-conditioned. Results should be the same than in Dataset 6.
- **Dataset 8**. Both signs, few small values and same proportion of large and medium values. Only medium values cancel. Results should be the same than in Dataset 6.
- **Dataset 9**. In order to simulate data encounted in embedded systems, this dataset is composed of intervals defined by $[0.4, 1.6]$. This is representative of values sent by a a sensor to an accumulator. This dataset is well-conditioned.

Example of data generated for Dataset 1:
$a = [2.667032062476577e^{16}, 3.332967937523422e^{16}]$
$b = [1.778021374984385e^{-16}, 2.221978625015614e^{-16}]$
$c = \ldots$ etc.

Figure 5 shows the proportion of optimal algorithms, i.e. the ones which return the smallest error with each dataset for the corresponding level of parallelism. Each proportion is the average value for the ten samples within each dataset. Parallelism degrees are $O(\lfloor \log(n) \rfloor + 1)$, $O(\lfloor \log(n) \rfloor + k)$, $O(k \times \lfloor \log(n) \rfloor)$, and $O(n)$ which describe all the algorithms of all levels of parallelism, as defined in Subsection 3.1.

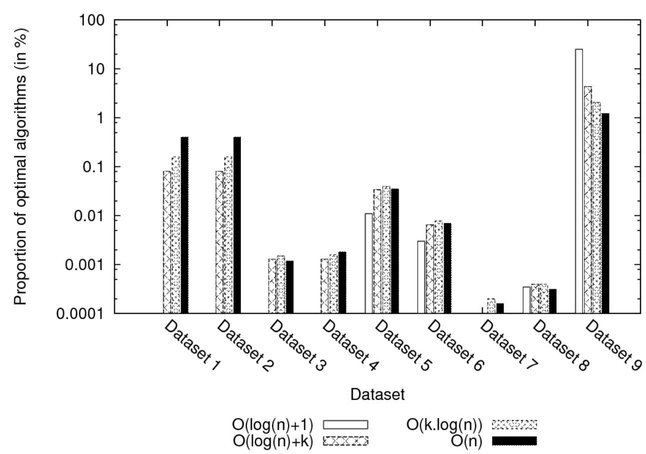


Figure 5: Proportion of the optimal algorithm on ten terms summation (average on 10 datasets).

First, we can observe that the proportion of optimal algorithms is tiny: the average of optimal algorithms with respect to the best accuracy is less than one percent except for the well-conditioned Dataset 9. Results in Table 1 match those displayed in Figure 5. In most cases, among all the levels of parallelism, the highest degree in $O(\lfloor \log(n) \rfloor + 1)$ is not able to keep the most accurate algorithms, particularly when there is absorption (percentage equals zero and no bar is plotted). We observe that the more the level of parallelism is, the harder it is to find a good algorithm. But if we relax the time constraint, i.e. the parallelism, it is easier to get an optimal algorithm.

For example, results of Dataset 1 show that if we limit the algorithms to all the algorithms of complexity $O(\lfloor \log(n) \rfloor + 1)$ there are no algorithm with the best error. If the level of parallelism is not so good, for example $O(\lfloor \log(n) \rfloor + k)$ or $O(k \times \lfloor \log(n) \rfloor)$ there are algorithms with the best errors.

Results in Figure 5 show that for Dataset 9, the proportion of optimal algorithms with the highest degree of parallelism is larger than the ones with less parallelism. In this case of well-conditioned summation, it reflects that whereas there are less algorithms of this parallelism level, these ones do not particularly suffer from inaccuracy. For well-conditioned summation, it seems that it is easier and easier to find an optimal algorithm as parallelism increases.

4. GENERATION OF THE ALGORITHMS

In this section, we describe how our tool generates all the algorithms. Our program, written in C++, builds all the reparsing of an expression. In the case of summation, the combinatory is huge, so it is very important to reduce the reparsing to the minimum.

The combinatory of summation is important, this was often studied but no general solution exists. For example, see CGPE [17] which computes equivalent polynomial expressions.

Intuitively, to generate all the expressions for a sum of n terms we process as follows.

- Step 1 : Generate all the parsings using the associativity of summation $((a+b)+c = a+(b+c))$. The number of parsings is given by the Catalan Number C_n [18]:

$$C_n = \frac{(2n)!}{n!\,(n+1)!}$$

- Step 2 : Generate all the permutations for all the expressions found in Step 1 using the commutativity of summation $(a+b = b+a)$. There is $n!$ ways to permute n terms in a sum.

So, the total number of equivalent expressions for a n terms summation is

$$C_n \cdot n! \tag{7}$$

Figure 6 shows this first combinatory result.

Our tool finds all the equivalent expressions of an original expression but it only generates the different equivalent expressions. For example, $a+(b+c)$ is equivalent to $a+(c+b)$ but it is not different because it corresponds to the same algorithm: these expressions correspond to the same sequence of operation. In Subsection 4.1, we present how we generate the structurally different trees and in Subsection 4.2, how we generate the permutations.

Table 4 and Figure 6 represent the number of algorithms generated for n terms as n grows.

The summation is a complex case, for example CGPE [17] generate equivalent polynomial expressions using heuristics to find a result as fast as possible. We want to do a study on exhaustive expressions reparsings, so because the combinatory is huge, we use ten terms during this work.

Terms	All expressions	Different expressions
5	1680	120
10	$1.76432e^{+10}$	$4.66074e^{+07}$
15	$3.4973e^{+18}$	$3.16028e^{+14}$
20	$4.29958e^{+27}$	$1.37333e^{+22}$

Table 4: Number of terms and expressions.

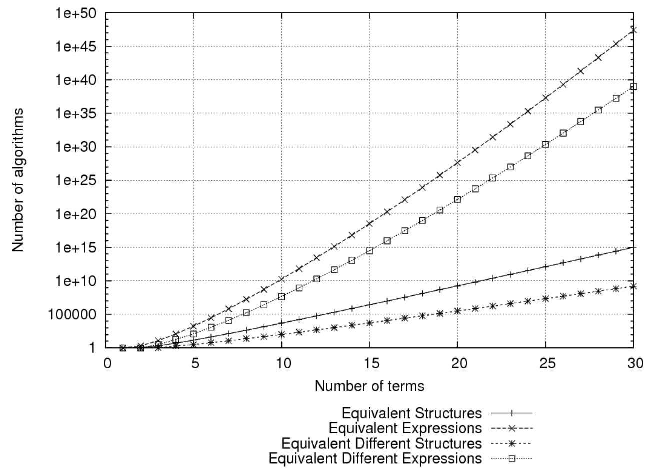


Figure 6: Number of trees when summing n terms.

4.1 Exhaustive Generation of Structurally Different Trees

We represent one algorithm with one binary tree. Nodes are sum operators and leaves are values. We describe how to generate all the structurally different trees. It is a recursive method defined as follows.

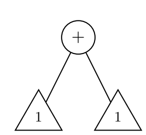
- We know that the number of terms is $n \geq 1$. An expression is composed of one term at least.

- A leaf x has only one representation, it is a tree of one term represented like this: △1.

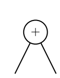
  x

  Then the number of structures for one term trivially reduces to one.

- Expression $x_1 + x_2$ is a tree of two terms △2. It has the following structural representation.



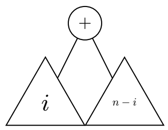
  With two terms we can create only one tree. So again the number of structures for two terms equals 1.



$x_1 \quad x_2$

- Recursively, we apply the same rules. For a tree of n terms, we generate all the different structural trees for all the possible combinations of sub-trees, i.e. for all $i \in [1, n-1]$, two sub-trees with, respectively, i and $(n-i)$ terms. Because summation is commutative, it is sufficient to generate these $(i; n-i)$-sub-trees for all $i \in [1, \lfloor \frac{n}{2} \rfloor]$. This is represented as follows:

$\forall i \in [1, \lfloor \frac{n}{2} \rfloor]$,



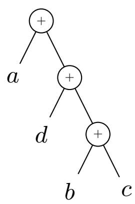
- So, for n terms, we generate the following numbers of structurally different trees,

$$S_{truct}(1) = S_{truct}(2) = 1, \qquad (8)$$

$$S_{truct}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} S_{truct}(n-i) \cdot S_{truct}(i). \qquad (9)$$

4.2 Generation of Permutations

To generate only different permutations, the leaves are related to the tree structure. For example, we do not wish to have the following two permutations, $a + (d + (b + c))$ and $a + ((c + b) + d)$.



Indeed, these expressions have the same accuracy and the same degree of parallelism.

In order to generate all the permutations, we use a similar algorithm as in the previous subsection.

- Firstly, we know that for an expression of one term, we may generate only one permutation. $P_{erm}(1) = 1$.
- Using our permutation restriction, it is sufficient to generate one permutation for an expression of two terms; so, again, $P_{erm}(2) = 1$.
- Permutations are related to the tree structures and we count them with the following recursive relation,

$$P_{erm}(1) = P_{erm}(2) = 1, \qquad (10)$$

$$P_{erm}(n) = \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \binom{i}{n} \cdot P_{erm}(n-i) \cdot P_{erm}(i). \qquad (11)$$

Equations $S_{truct}(n)$ and $P_{erm}(n)$ are asymptotically exponential.

5. FURTHER EXAMPLES

In this section, we present results for larger or more sophisticated examples. Subsection 5.1 introduces a sum of twenty terms, Subsection 5.2 focuses on compensation and we discuss about bounded parallelism in Subsection 5.3

5.1 An Example Over More Terms

We now consider a sum of 20 terms. We chose a dataset where all the values belong to the interval $[0.4, 1.6]$. Again this is representative, for example, of what may happen in an embedded system when accumulating values provided by a sensor, like a sinusoidal signal.

Critical path	Average of optimal algorithms (%)	
$\lfloor \log(n) \rfloor + 1$	54.08	
	$k = 2$	$k = 3$
$\lfloor \log(n) \rfloor + k$	19.75	12.41
$k \times \lfloor \log(n) \rfloor$	5.26	4.15
n	4.13	

Table 5: Proportion of optimal algorithms.

We can see that the results in Table 5 are similar to the results of Dataset 9. We obtain the same repartition of optimal algorithms with ten or twenty terms. This confirms that the sum length does not govern the accuracy – at least while overflow does not appear.

In this case, we show that for a sum of identical intervals, the more parallelism, the easier to find an algorithm which preserves the maximum accuracy.

5.2 Compensated Summation

Now we present an example to illustrate one of the core motivation of this work. The question is the following: Starting from the simplest sum algorithm, are we able to automatically generate a compensated summation algorithm that improves the evaluation ? Here we describe how to introduce one level of compensation as in the algorithms presented in Section 2 (TwoSum, Sum2, SumComp).

To improve the accuracy of expression E, we compute an expression E_{cmp}.

For values X and Y, we introduce the function $C(X, Y)$ which computes the compensation of $X \oplus Y$ (like in Algorithm TwoSum, see section 2.1).

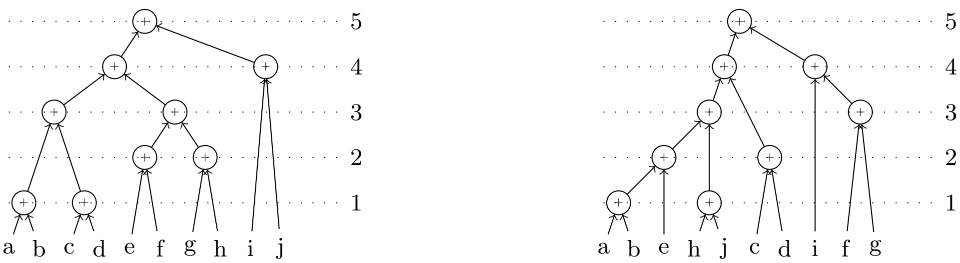
For example, for three terms we have:
$E = (X \oplus Y) \oplus Z$
$E_{cmp} = \big[((X \oplus Y) \oplus C(X, Y)) \oplus Z\big] \oplus C(X + Y, Z)$

E_{cmp} is the expression we obtain automatically by systematically compensating the original sums. It could be generated by a compiler. To illustrate this, we present one example with a summation of five terms $((((a + b) + c) + d) + e)$. Terms are defined as follows,

$$a = -9.5212224350e^{-18}$$
$$b = -2.4091577979e^{-17}$$
$$c = 3.6620086288e^{+03}$$
$$d = -4.9241247828e^{+16}$$
$$e = 1.4245601293e^{+04}$$

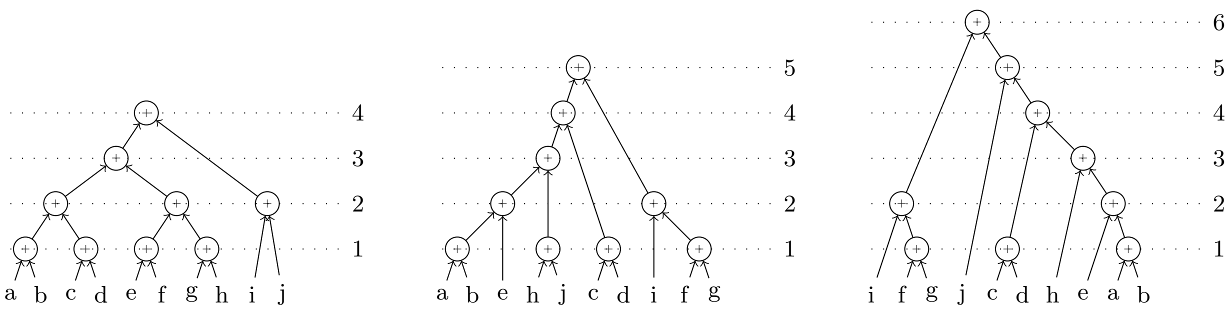
As before we can identify the two followings cases. The maximal accuracy which can be obtained, among all the reparsing of this five terms expression, is given by the following algorithm: $(((a+b)+c)+e)+d$. It generates the absolute error $\Delta = 4.0000000000020472513$. We observe that this algorithm is Algorithm 1 at Section 2 with increase order.

The maximal accuracy given by the maximal level of parallelism is obtained by the algorithm $((a + c) + (b + e)) + d$.



a) Fully Parallel and Non-Fully Accurate Algorithm. b) Fully Accurate and Non-Fully Parallel Algorithm.

Figure 7: Dataflow Graphs of Algorithms in Bounded Parallelism on 2sums/cycle architecture.



a) Fully Parallel and Non-Fully Accurate Algorithm.

b) Fully Accurate and $\log(n) + k$ Parallelism Degree Algorithm.

c) Fully Accurate and $k \times \log(n)$ Parallelism Degree Algorithm.

Figure 8: Dataflow Graphs of Algorithms in Bounded Parallelism on 4sums/cycle architecture.

In this case, the absolute error is

$$\delta_{nocomp} = 4.0000000000029558578.$$

When applying compensation on this algorithm, we obtain the following algorithm :

$$(f + (g + (h + i))) + (d + ((b + e) + (a + c))),$$

with :

$$\begin{aligned} f = C(a, c) &= -9.5212224350000e^{-18} \\ g = C(b, e) &= -2.4091577978999e^{-17} \\ h = C(f, g) &= -1.8189894035458e^{-12} \\ i = C(h, d) &= 3.6099218000017 \end{aligned}$$

Now, we measure the improved absolute error $\delta_{comp} = 4.0000000000000008881$. It happens that this algorithm found with the application of compensation is actually the Sum2 algorithm –Algorithm 6 at Section 2. This results illustrates that we can automatically find algorithms existing in the bibliography and that the transformation improves the accuracy.

5.3 Bounded Instruction Level Parallelism

Section 3 showed that in the case of maximum parallelism, maximum accuracy is not possible (or very difficult) to have. The fastest algorithms ($O(\lfloor \log(n) \rfloor + 1)$ are rarely the most accurate but by relaxing the time constraint, it becomes possible to find an optimally accurate algorithm.

This subsection is motivated by the following fact. In processor architectures, parallelism is bounded. So it is possible to execute an algorithm less parallel in the same execution time, or in a very closed time, as the fastest parallel one. We show here two examples to illustrate this. Firstly we use a processor which executes two sums per cycle and secondly one which executes four sums per cycle.

For an expression of ten terms :

- **2 sums/cycle**:

  The execution of the fastest algorithm ($\lfloor \log(n) \rfloor + 1$) for the expression does not provide the maximum accuracy. It takes five cycles to compute the expression as shown in a), in Figure 7. Now we take another algorithm, with less parallelism but that provides the maximum accuracy (See Line 1, Table 2, Subsection 3.1). In bounded parallelism this algorithm takes the same time than the more parallel one as shown in b), in Figure 7.

- **4 sums/cycle**:

  Again, execution of the fastest algorithm ($\lfloor \log(n) \rfloor + 1$) of this expression, do not have the maximum accuracy. It takes four cycles to compute the expression (See a), in Figure 8). We take two other algorithms, both with less parallelism but with the maximum accuracy. The first algorithm is described at Line 1, Table 2, Subsection 3.1. It takes one more cycle than the most parallel one (See b), in Figure 8). The second algorithm is in $k \times \lfloor \log(n) \rfloor$; it corresponds to Line 2, Table 3, Subsection 3.1. This one takes two more cycles than the most parallel one (See c), in Figure 8).

This confirms our claim that in current architectures, we can improve accuracy without lowering too much the execution.

6. CONCLUSION AND PERSPECTIVES

We have presented our first steps towards the development of a tool that aims at automatically improving the accuracy of numerical expressions evaluated in floating-point arithmetic. Since we target to embed such tool within compiler, introducing more accuracy should not jeopardize the improvement of running-time performances provided by the optimization steps. This motivates to study the simultaneous improvement of accuracy and timing. Of course we exhibit that a trade-off is necessary to generate optimal transformed algorithms. We validated the presented tool with summation algorithms; these are simple but significant problems in our application scope. We have shown that this trade-off can be automatically reached, and the corresponding algorithm generated, for data belonging to intervals – and not only scalars. These intervals included ill-conditioned summations. In the last section, we have shown that we can automatically generates more accurate algorithms that use compensation techniques. Compared to the fastest algorithms, the overcost of these automatically generated more accurate algorithms may be reasonable in practice. Our main conclusion is that relaxing very slightly the time constraints by choosing algorithms whose critical paths are a bit longer than the optimal makes it possible to strongly optimize the accuracy.

Next step needs to increase the complexity of the case study both performing more operations and different ones. One of the main problem to tackle is the combinatory of the possible transformations. Brute force transformation should be replaced using heuristics or more sophisticated transformations as, e.g. the error-free ones we introduced to recover the compensated algorithms. Another point to explore is how to develop significant datasets corresponding to any data intervals provided by the user of the expression to transform. A further step will be to transform any expression up to a prescribed accuracy and to formally certified it. Such facility is for example necessary to apply such tool for symbolic-numeric algorithms. In this scope, this project plans to use static analysis and abstract interpretation as in [12].

7. REFERENCES

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, revision of std 754-1985 edition, 2008.

[2] James Demmel. Trading off parallelism and numerical stability, 1992.

[3] Stef Graillat and Philippe Langlois. Real and complex pseudozero sets for polynomials with applications. *Theoretical Informatics and Applications*, 41(1):45–56, 2007.

[4] Nicholas J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14:783–799, 1993.

[5] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[6] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. Faster floating-point square root for integer processors. In *IEEE Symposium on Industrial Embedded Systems (SIES'07)*, Lisbon, Portugal, July 2007.

[7] Claude-Pierre Jeannerod and Guillaume Revy. Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores. In *Proceedings of the 43rd Asilomar Conference on Signals, Systems, and Computers (Asilomar'09)*, Pacific Grove, CA, USA, november 2009.

[8] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] Philippe Langlois. Compensated algorithms in floating point arithmetic. In *12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics, Duisburg, Germany*, September 2006. (Invited plenary speaker).

[10] Jean Louis Lions, Remy Hergott (CNES), Bernard Humbert (Aerospatiale), and Eric Lefort (ESA). Ariane 5, flight 501 failure, report by the inquiry board. european space agency, 1996.

[11] Matthieu Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19:7–30, 2006.

[12] Matthieu Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Journal of Formal Methods in System Design*, 35:265–278, 2009.

[13] United States General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Report GAO/IMTEC-92-26, Information Management and Technology Division, Washington, D.C., February 1992.

[14] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26, 2005.

[15] Michèle Pichat. Correction d'une somme en arithmétique à virgule flottante. *Applied Numerical Mathematics*, 19:400–406, 1972.

[16] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David W. Matula, editors, *Proc. 10th IEEE Symposium on Computer Arithmetic*, pages 132–143. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991.

[17] Guillaume Revy. *Implementation of binary floating-point arithmeric on embedded integer processors*. PhD thesis, École Normale Supérieure de Lyon, 2009.

[18] Fred S. Roberts. *Applied combinatorics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[19] Siegfried M. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502, 2009.

[20] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. Accurate floating-point summation –part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[21] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, second edition, 1993.