Operator Dependant Compensated Algorithms

Philippe Langlois Nicolas Louvet
DALI at ELIAUS Laboratory, Université de Perpignan Via Domitia
52, avenue Paul Alduy, F-66860 Perpignan, France
[langlois, nicolas.louvet]@univ-perp.fr

Abstract

Compensated algorithms improve the accuracy of a result evaluating a correcting term that compensates the finite precision of the computation. The implementation core of compensated algorithms is the computation of the rounding errors generated by the floating point operators. We focus this operator dependency discussing how to manage and to benefit from floating point arithmetic implemented through a fused multiply and add operator. We consider the compensation of dot product and polynomial evaluation with Horner iteration. In each case we provide theoretical a priori error bounds and numerical experiments to exhibit the best algorithmic choices with respect to accuracy or performance issues.

1. Introduction

Different techniques and several softwares aim to improve the accuracy of results computed in a fixed finite precision, *e.g.*, in IEEE-754 floating point arithmetic [10].

A natural way to improve the accuracy of a given computation is to increase the working precision. For this purpose, numerous multiprecision libraries are available when the computing precision is not large enough to guarantee a prescribed accuracy [4, 2, 14]. The computing-time overhead of such arbitrarily precise computation limits its use to applications for which running-time is not crucial. When twice or four times the IEEE-754 double precision is sufficient, actual and effective solutions are double-double or quad-double libraries [8, 1]. For example a double-double number is an unevaluated sum of two IEEE-754 double precision numbers and its associated arithmetic provides at least 106 bits of significand. These fixed-length expansions are currently embedded in major developments such as for example within the new extended and mixed precision BLAS [12]. Such libraries benefit from good performances in term of running-time and also from a wide applicability since they provide extended precision for classic arithmetic operators and elementary functions.

Compensating a given algorithm is a less generic process than the simple plug-in of the extended precision facilities previously mentioned. Nevertheless farther presented results will exhibit that when available, compensated algorithms run always faster than the corresponding ones with extended precision libraries. Compensated algorithms implement the computation of a correcting term that approximates the errors generated by the finite precision evaluation of the algorithm. This computation relies on error-free transformations (EFT) as named in [16]. EFT are properties that describe the final forward error such that (an approximate of) this error can be computed only using the current working precision. Such EFT for arithmetic operators, dot product and polynomial evaluation will be given hereafter. The core of the EFT computation depends on low-level arithmetic properties (which is also the case for extended precision libraries); most of them are clearly defined by the IEEE-754 standard. Nevertheless new questions raise when IEEE-754 compliant add or multiply operators are implemented from a unique fused multiply and add instruction. The fused multiply and add instruction (FMA) is available on some current processors, such as the IBM Power PC or the Intel Itanium. Given a, b and c three floating point point values, this instruction computes the expression $a \times b + c$ with only one final rounding error [13].

The FMA can be used to improve algorithms based on error-free transformations in two ways. First, it allows us to compute the EFT for the product of two floating point values in a very efficient way: algorithm TwoProd recalled hereafter computes this EFT in only two flops when a FMA is available [15, 13]. On the other hand, an algorithm that computes an EFT for the FMA has been proposed in [3]. In particular, it is proved that the EFT for the FMA is the sum of three floating point numbers. Assuming an IEEE-754 like floating point arithmetic with the round to the nearest rounding mode, algorithm ThreeFMA

Table 1. Summary of algorithms

routine	description
HornerFMA	IEEE-754 double precision with FMA (Algorithm 5)
CompHornerFMA	Compensated HornerFMA (Algorithm 7)
CompHorner	Compensated Horner (Algorithm 9)
DDHorner	Horner algorithm performed with the double-double format + FMA
DotFMA	Dot product algorithm with FMA (Algorithm 11)
CompDotFMA	Compensated DotFMA (Algorithm 12)
CompDot	Compensated Dot (Algorithm 13)
DDDot	Dot product with the double-double format + FMA

computes three floating point numbers x, y and z such that

$$a \times b + c = x + y + z \quad \text{with} \quad x = \mathsf{FMA}\,(a, b, c)\,.$$

In this paper we focus on the FMA dependency of compensated algorithms. We discuss how to manage and to benefit from this fused multiply and add operator. Notations are presented in Section 2 and then error-free transformations are introduced in Section 3.

We consider the compensation of polynomial evaluation with Horner iteration and dot product, respectively in Section 4 and Section 5. In each case we provide theoretical *a priori* error bounds and numerical experiments to exhibit the best algorithmic choices with respect to accuracy or performance issues. *A priori* error bounds prove that the FMA does not significantly improve the worst-case error – even if implementations with FMA suffer from twice less rounding errors than without. Experiments also illustrate this similar behavior in terms of accuracy for both original and compensated algorithms.

Running-time issues are different for compensated algorithms. We conclude that FMA should be avoided in the main computation (replacing it by add or multiply operators) but preferred in the compensating process (namely to compute the error generated by the multiply operator). This should motivate further research to less costly algorithms or availability of low level primitives that compute the error generated by the FMA. Hence the whole computation of compensated algorithms would benefit from the fused multiply and add instruction.

2. Notations

Throughout the paper, we assume a floating point arithmetic adhering to the IEEE-754 floating point standard [10]. We constraint all the computations to be performed in one working precision, with the "round to the nearest" rounding mode. We also assume that no overflow nor underflow occurs during the computations. Next notations are standard (see [9, chap. 2] for example). $\mathbb{F}$ is the set of all normalized floating point numbers and $\mathbf{u}$ denotes the unit roundoff, that is half the spacing between 1 and the next larger representable floating point value. For IEEE-754 double precision with rounding to the nearest, we have $\mathbf{u} = 2^{-53} \approx 1.11 \cdot 10^{-16}$.

The symbols $\oplus$, $\ominus$ and $\otimes$ represent respectively the floating point addition, subtraction and multiplication. For more complex arithmetic expressions, $fl(\cdot)$ denotes the result of a floating point computation where every operation inside the parenthesis is performed in the working precision. So we have for example, $a \oplus b = fl(a + b)$.

When no underflow nor overflow occurs, the following standard model describes the accuracy of every considered floating point computation. For two floating point numbers a and b and for $\circ$ in $\{+, -, \times\}$, the floating point evaluation $fl(a \circ b)$ of $a \circ b$ is such that

$$fl(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2), \quad \text{with} \quad |\varepsilon_1|, |\varepsilon_2| \leq \mathbf{u}. \quad (1)$$

To keep track of the $(1+\varepsilon)$ factors in next error analysis, we use the classic $(1 + \theta_k)$ and γ_k notations [9, chap. 3]. For any positive integer k, θ_k denotes a quantity bounded according to

$$|\theta_k| \leq \gamma_k = \frac{k\mathbf{u}}{1 - k\mathbf{u}}.$$

When using these notations, we always implicitly assume $k\mathbf{u} < 1$. In farther error analysis, we essentially use the following relations,

$$(1 + \theta_k)(1 + \theta_j) \leq (1 + \theta_{k+j}), \quad k\mathbf{u} \leq \gamma_k, \quad \gamma_k \leq \gamma_{k+1}.$$

3. Error Free Transformations (EFT)

First we review error free transformations (EFT) known for the elementary floating point operations $+$, $-$ and $\times$.

Let $\circ$ be an operator in $\{+, -, \times\}$, a and b be two floating point numbers, and $x = fl(a \circ b)$. Then it exists a floating point value y such that

$$a \circ b = x + y. \quad (2)$$

Table 2. Summary of *a priori* bounds and flop counts.

Algorithm	*A priori* bound for the relative accuracy	Number of flop
HornerFMA	$\gamma_n \operatorname{cond}(p,x)$	n
Horner	$\gamma_{2n} \operatorname{cond}(p,x)$	$2n$
CompHornerFMA	$\mathbf{u} + \gamma_n\gamma_{n+1} \operatorname{cond}(p,x)$	$19n$
CompHorner	$\mathbf{u} + \gamma_n\gamma_{2n+1} \operatorname{cond}(p,x)$	$10n-1$
DotFMA	$\gamma_n \operatorname{cond}(x^Ty)/2$	n
Dot	$\gamma_n \operatorname{cond}(x^Ty)/2$	$2n-1$
CompDotFMA	$\mathbf{u} + \mathbf{u}\gamma_{n+1} \operatorname{cond}(x^Ty)/2$	$19n-16$
CompDot	$\mathbf{u} + \gamma_n^2 \operatorname{cond}(x^Ty)/2$	$10n-7$

The difference y between the exact result and the computed result is the rounding error generated by the computation of x. Let us emphasize that relation (2) between four floating point values only relies on real operators and exact equality. Ogita *et al.* [16] name such a transformation an error free transformation (EFT). The practical interest of the EFT comes from next Algorithms 1 and 2 that compute the exact error term y for $\circ = +$ and $\circ = \times$.

For the EFT of the addition we use Algorithm 1, the well known TwoSum algorithm by Knuth [11] that requires 6 flop (floating point operations).

Usually, the well known algorithm TwoProd by Veltkamp and Dekker (see [5]) is used for the EFT of the product. TwoProd requires 17 floating point operations. Nevertheless, TwoProd can be rewritten very efficiently when a FMA is available. For a, b and c in $\mathbb{F}$, FMA (a,b,c) is the exact result $a \times b + c$ rounded to the nearest floating point value. Thus, $y = a \times b - a \otimes b =$ FMA $(a, b, -(a \otimes b))$, and TwoProd now only requires two flop.

The next theorem exhibits the previously announced properties of TwoSum and TwoProd.

Theorem 1 ([16]). *Let a, b in $\mathbb{F}$ and $x, y \in \mathbb{F}$ such that $[x,y] =$ TwoSum(a,b) (Algorithm 1). Then, even in the presence of underflow,*

$$a+b = x+y, \quad x = a \oplus b, \quad |y| \le \mathbf{u}|x|, \quad |y| \le \mathbf{u}|a+b|.$$

Let $a, b \in \mathbb{F}$ and $x, y \in \mathbb{F}$ such that $[x,y] =$ TwoProd(a,b) (Algorithm 2). Then, if no underflow occurs,

$$a \times b = x+y, \quad x = a \otimes b, \quad |y| \le \mathbf{u}|x|, \quad |y| \le \mathbf{u}|a \times b|.$$

An algorithm that computes an EFT for the FMA has been recently given by Boldo and Muller [3]. The EFT of a FMA operation cannot be represented as a sum of two floating point numbers, as it is the case for the addition and for the product. Therefore, the following algorithm ThreeFMA produces three floating point numbers. For efficiency reasons, we slightly modify the algorithm from [3] such that ThreeFMA here performs no renormalization of the final result. Algorithm 3 requires 17 flop. It satisfies the following properties.

Theorem 2 ([3]). *Given a, b, and c three floating point values, let x, y and z be the three floating point numbers such that $[x,y,z] =$ ThreeFMA (a,b,c). Then we have*

- $a \times b + c = x + y + z$ *exactly, with* $x =$ FMA (a,b,c),
- $|y+z| \le \mathbf{u}|x|$ *and* $|y+z| \le \mathbf{u}|a \times b + c|$,
- $y = 0$ *or* $|y| > |z|$.

We notice that the algorithms presented in this section only require well optimizable floating point operations. They do not use branches nor access to the mantissa that can be time-consuming.

Two error free transformations for polynomial evaluation are introduced in next Section 4. Relation (7) exhibits the exact rounding error generated by the Horner algorithm when its inner iteration uses a FMA; Relation (10) applies when no FMA appears in the Horner algorithm.

4. Polynomial evaluation

We consider the evaluation of $p(x) = \sum_{i=0}^{n} a_i x^i$, where the data x and the polynomial coefficients a_i are floating point numbers. We study the two versions of the classic Horner algorithm (without or with the FMA) and associated compensated Horner algorithms.

We recall that the classic condition number of the evaluation of $p(x)$ is

$$\operatorname{cond}(p,x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|\sum_{i=0}^{n} a_i x^i|} = \frac{\widetilde{p}(x)}{|p(x)|}. \tag{3}$$

4.1. Horner algorithms

For any floating point value x, Horner (p,x) is the result of the floating point evaluation of the polynomial p at x using the Horner algorithm (Algorithm 4).

Algorithm 1. EFT of the sum of two floating point numbers.

function $[x, y] = \mathsf{TwoSum}\,(a, b)$
  $x = a \oplus b$
  $z = x \ominus a$
  $y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

Algorithm 2. EFT of the product of two floating point numbers with a FMA.

function $[x, y] = \mathsf{TwoProd}\,(a, b)$
  $x = a \otimes b$
  $y = \mathsf{FMA}\,(a, b, -x)$

Algorithm 3. EFT for the FMA operation.

function $[x, y, z] = \mathsf{ThreeFMA}\,(a, b, c)$
  $x = \mathsf{FMA}\,(a, b, c)$
  $(u_1, u_2) = \mathsf{TwoProd}\,(a, b)$
  $(\alpha_1, z) = \mathsf{TwoSum}\,(b, u_2)$
  $(\beta_1, \beta_2) = \mathsf{TwoSum}\,(u_1, \alpha_1)$
  $y = (\beta_1 \ominus x) \oplus \beta_2$

Algorithm 4. Horner algorithm

function $r_0 = \mathsf{Horner}\,(p, x)$
$r_n = a_n$
for $i = n - 1 : -1 : 0$
  $r_i = r_{i+1} \otimes x \oplus a_i$
end

A forward error bound for the result of Algorithm 4 is (see [9, p.95])

$$|p(x) - \mathsf{Horner}\,(p, x)\,| \leq \gamma_{2n}\,\widetilde{p}(x). \tag{4}$$

So, the accuracy of the computed evaluation is linked to the condition number of the polynomial evaluation as follows,

$$\frac{|p(x) - \mathsf{Horner}\,(p, x)\,|}{|p(x)|} \leq \gamma_{2n}\,\mathrm{cond}(p, x). \tag{5}$$

Clearly, the condition number (3) can be arbitrarily large. In particular, when $\mathrm{cond}(p, x) > \gamma_{2n}^{-1}$, we cannot guarantee that the computed result $\mathsf{Horner}\,(p, x)$ contains any correct digit.

If a FMA instruction is available on the considered architecture, then we can change the computation of $r_i = r_{i+1} \otimes x \oplus a_i$ in Algorithm 4 by $r_i = \mathsf{FMA}\,(r_{i+1}, x, a_i)$. This gives the following algorithm HornerFMA (Algorithm 5).

Algorithm 5. Horner algorithm with FMA

function $r_0 = \mathsf{HornerFMA}\,(p, x)$
$r_n = a_n$
for $i = n - 1 : -1 : 0$
  $r_i = \mathsf{FMA}\,(r_{i+1}, x, a_i)$
end

This slightly improves the error bound since we write now,

$$\frac{|p(x) - \mathsf{HornerFMA}\,(p, x)\,|}{|p(x)|} \leq \gamma_n\,\mathrm{cond}(p, x). \tag{6}$$

With the FMA, the number of floating point operations involved in the computation is also divided by two and so is the worst case error.

4.2. Compensating HornerFMA

As previously mentioned, next EFT for the polynomial evaluation with HornerFMA exhibits the exact rounding error generated by this algorithm. Following algorithm EFTHornerFMA computes this EFT thanks to ThreeFMA (Algorithm 3).

Algorithm 6. EFT for HornerFMA

function $[u_0, p_\varepsilon, p_\varphi] = \mathsf{EFTHornerFMA}(p, x)$
$u_n = a_n$
for $i = n - 1 : -1 : 0$
  $[u_i, \varepsilon_i, \varphi_i] = \mathsf{ThreeFMA}\,(u_{i+1}, x, a_i)$
  Let ε_i be the coefficient of degree i in p_ε
  Let φ_i be the coefficient of degree i in p_φ
end

Theorem 3. *Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n with floating point coefficients, and let x be a floating point value. Algorithm 6 computes both*

- *the floating point evaluation* $\mathsf{HornerFMA}\,(p, x)$ *(Algorithm 5), and*
- *two polynomials p_ϵ and p_φ, of degree $n-1$, with floating point coefficients;*

we write

$$[\mathsf{HornerFMA}\,(p, x)\,, p_\varepsilon, p_\varphi] = \mathsf{EFTHornerFMA}\,(p, x)\,.$$

Algorithm 6 requires $17n$ floating point operations.

We have the next EFT,

$$p(x) = \mathsf{HornerFMA}\,(p, x) + (p_\varepsilon + p_\varphi)(x), \tag{7}$$

with

$$\widetilde{(p_\varepsilon + p_\varphi)}(x) \leq \gamma_n\,\widetilde{p}(x).$$

As before we have $\widetilde{(p_\varepsilon + p_\varphi)}(x) = \sum_{i=0}^{n-1} |\varepsilon_i + \varphi_i||x^i|$. Relation (7) means that EFTHornerFMA is an EFT for the polynomial evaluation with the Horner algorithm when the FMA is used. From this relation, the global forward error

affecting the floating point evaluation of p at x according to the Horner algorithm is

$$p(x) - \mathsf{HornerFMA}\,(p,x) = (p_\varepsilon + p_\varphi)(x), \qquad (8)$$

where the coefficients of the polynomials p_ε and p_φ are exactly computed by EFTHornerFMA (Algorithm 6), together with the approximate $\mathsf{HornerFMA}\,(p,x)$. Therefore, the key of the following compensated algorithm is to compute an approximate c of the global error (8) in working precision, and then to compute a corrected result

$$r = \mathsf{HornerFMA}\,(p,x) \oplus c.$$

We say that c is a correcting term for the initial result $\mathsf{HornerFMA}\,(p,x)$. The corrected result r is expected to be more accurate than $\mathsf{HornerFMA}\,(p,x)$ as proved in the sequel of the section. We compute the correcting term c by evaluating the polynomial whose coefficients are those of $p_\varepsilon + p_\varphi$ rounded to the nearest floating point value, *i.e.*, $c = \mathsf{HornerFMA}\,(p_\varepsilon + p_\sigma, x)$. We can now describe the compensated algorithm for polynomial evaluation.

Algorithm 7. Compensated HornerFMA

function $r = \mathsf{CompHornerFMA}\,(p,x)$
$[h, p_\varepsilon, p_\varphi] = \mathsf{EFTHornerFMA}\,(p,x)$
$c = \mathsf{HornerFMA}\,(p_\varepsilon \oplus p_\varphi, x)$
$r = h \oplus c$

We state hereafter that the result of a polynomial evaluation computed with Algorithm 7 is as accurate as if computed by the classic Horner algorithm using twice the working precision and then rounded to the working precision (proofs are detailed in [7]).

Theorem 4. *Given a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ of degree n with floating point coefficients, and x a floating point value. We consider the result* $\mathsf{CompHornerFMA}\,(p,x)$ *computed by Algorithm 7. Then,*

$$|\mathsf{CompHornerFMA}\,(p,x) - p(x)| \le \mathbf{u}|p(x)| + \gamma_n\gamma_{n+1}\,\widetilde{p}(x).$$

CompHornerFMA *requires $19n$ floating point operations.*

It is interesting to interpret the previous theorem with respect to the condition number of the polynomial evaluation of p at x. Combining the error bound in Theorem 4 with the condition number (3) for the polynomial evaluation gives the following relation,

$$\frac{|\mathsf{CompHornerFMA}\,(p,x) - p(x)|}{|p(x)|} \le \mathbf{u} + \gamma_n\gamma_{n+1}\,\mathrm{cond}(p,x). \qquad (9)$$

For practical purpose, just consider $\gamma_n\gamma_{n+1}$ as $\mathbf{u}^2$. In other words, the bound for the relative error of the computed result is essentially $\mathbf{u}^2$ times the condition number of the polynomial evaluation, plus the inevitable summand $\mathbf{u}$ for the final rounding of the result to the working precision. In particular, while $\mathrm{cond}(p,x) \lesssim 1/\mathbf{u}$, then the relative accuracy of the result is bounded by a constant of the order $\mathbf{u}$. This means that the compensated Horner algorithm computes an evaluation accurate to the last few bits as long as the condition number is smaller than $1/\mathbf{u}$. Besides that, Relation (9) tells us that the computed result is as accurate as if computed by the classic Horner algorithm with twice the working precision. Of course no accuracy can be expected for condition number larger than $1/\mathbf{u}^2$.

4.3. Compensating Horner

The principle of next algorithm CompHorner is the same as CompHornerFMA. Nevertheless we need an EFT which computes the rounding error generated by Horner, that is for polynomial evaluation without using the FMA. Next results provide this EFT.

Algorithm 8. EFT for Horner.

function $[q_0, p_\pi, p_\sigma] = \mathsf{EFTHorner}(p,x)$
$q_n = a_n$
for $i = n-1 : -1 : 0$
  $[p_i, \pi_i] = \mathsf{TwoProd}\,(q_{i+1}, x)$
  $[q_i, \sigma_i] = \mathsf{TwoSum}\,(p_i, a_i)$
  Let π_i be the coefficient of degree i in p_π
  Let σ_i be the coefficient of degree i in p_σ
end

Theorem 5. *Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be a polynomial of degree n with floating point coefficients, and let x be a floating point value. Then following Algorithm 8 computes both*

- *the floating point value* $\mathsf{Horner}\,(p,x)$ *(Algorithm 4), and*
- *two polynomials p_π and p_σ, of degree $n-1$, with floating point coefficients;*

we write

$$[\mathsf{Horner}\,(p,x), p_\pi, p_\sigma] = \mathsf{EFTHorner}\,(p,x).$$

Algorithm 8 requires $8n$ flops.

We have the next EFT,

$$p(x) = \mathsf{Horner}\,(p,x) + (p_\pi + p_\sigma)(x), \qquad (10)$$

with

$$(\widetilde{p_\pi} + \widetilde{p_\sigma})(x) \le \gamma_{2n}\,\widetilde{p}(x).$$

We deduce another compensated evaluation algorithm based on the previous EFT for Horner.

Algorithm 9. Compensated Horner algorithm.

function $r = \mathsf{CompHorner}\,(p, x)$
$[h, p_\pi, p_\sigma] = \mathsf{EFTHorner}\,(p, x)$
$c = \mathsf{HornerFMA}\,(p_\pi \oplus p_\sigma, x)$
$r = h \oplus c$

Theorem 6. *Given a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ of degree n with floating point coefficients, and x a floating point value. We consider the result* $\mathsf{CompHorner}\,(p, x)$ *computed by Algorithm 9. Then,*

$$|\mathsf{CompHorner}\,(p, x) - p(x)| \leq \mathbf{u}|p(x)| + \gamma_n\gamma_{2n+1}\,\widetilde{p}(x).$$

Algorithm 9 requires $10n - 1$ floating point operations.

Again, combining the error bound in Theorem 6 with the condition number (3) for polynomial evaluation leads to

$$\frac{|\mathsf{CompHorner}\,(p, x) - p(x)|}{|p(x)|} \leq \mathbf{u} + \gamma_n\gamma_{2n+1}\,\mathrm{cond}(p, x). \tag{11}$$

Since $\gamma_n\gamma_{2n+1} \approx \mathbf{u}^2$, the previous remarks about error bound (9) also apply to the previous one. While CompHornerFMA needs almost two times more flop than CompHorner, we notice that the error bounds (9) and (11) are similar.

Actually, our next experimental results confirm that CompHorner is more efficient than CompHornerFMA in terms of computing time while being similarly accurate.

4.4. Experimental scheme

All our experiments are performed using IEEE-754 double precision. Since the double-doubles [8, 12] are usually considered as the most efficient portable library to double the IEEE-754 double precision, we consider it as a reference in the following comparisons. For our purpose, it suffices to know that a double-double number a is the pair (a_h, a_l) of IEEE-754 floating point numbers with $a = a_h + a_l$ and $|a_l| \leq \mathbf{u}|a_h|$. This property implies a renormalization step after each arithmetic operation. We denote by DDHorner our implementation of the Horner algorithm with the double-double format, derived from the implementation proposed by the authors of [12]. We notice that the double-double arithmetic naturally benefits from the availability of a FMA instruction: DDHorner uses TwoProd in the inner loop of the Horner algorithm. DDHorner requires $20n$ floating point operations. Using the double-double library proposed in [8], we can slightly reduce this flop count, but it has almost no impact on the measured computing times.

4.5. Accuracy tests

We test the expanded form of the polynomial $p_n(x) = (x-1)^n$. Accuracy of the evaluation is not guaranteed in the neighborhood of the real root 1 of p_n. Indeed the condition number is

$$\mathrm{cond}(p_n, x) = \frac{\widetilde{p_n}(x)}{|p_n(x)|} = \left|\frac{|x| + 1}{x - 1}\right|^n,$$

and $\mathrm{cond}(p_n, x)$ grows exponentially with respect to n. In the experiments reported on Figure 1, we have chosen $x = fl(1.333)$ to provide a binary floating point value with many non-zero bits in its mantissa. The value of $\mathrm{cond}(p_n, x)$ varies from 10^2 to 10^{40}, that corresponds to degrees n range 3 to 42. These huge condition numbers have a sense since the coefficients of p and the value x are floating point numbers.

We experiment both HornerFMA, CompHornerFMA, CompHorner and DDHorner (see Table 1). For every polynomial p_n, the exact value $p_n(x)$ is approximated with high accuracy thanks to the MPFR library [14]. Figure 1 presents the relative accuracy $|r - p_n(x)|/|p_n(x)|$ of the evaluation r computed by each algorithm. We set to the value one relative errors greater than one, which means that almost no useful information is available in the computed result. We also display the *a priori* error estimates (6) and (9) – no difference between (9) and (11) appears on this figure. We observe that our compensated algorithms exhibit the expected behavior: compensated results are roughly as if the Horner algorithm is computed with twice more bits. We identify no significant difference between the accuracy provided by double-double implementation compared to compensated ones. The full precision solution is computed as long as the condition number is smaller than $\mathbf{u}^{-1} \approx 10^{16}$. Then, for condition numbers between $\mathbf{u}^{-1}$ and $\mathbf{u}^{-2} \approx 10^{32}$, the relative error degrades (linearly in the log scale) to no accuracy at all as it was expected from the *a priori* error bounds (9) and (11).

As usual, these *a priori* bounds are definitely pessimistic especially when the condition number term becomes predominant in (9) and (11). More realistic bounds are provided by a dynamic analysis we describe in [7]. This latter reference also presents experiments with a more generic choice of polynomials; results are still similar to those displayed on Figure 1.

4.6. Running-time tests

All the algorithms are implemented in a C-code. We use the same programming techniques for the implementations of the three routines CompHornerFMA, CompHorner and DDHornerFMA. The experimental environments are listed in Table 3. Our measures are performed with polynomials whose degrees vary from 5 to 200 by step of 5. We randomly choose the values of the coefficients and the arguments. For each degree, the routines are tested on the same polynomial with the same argument. Table 4 displays the time overhead of the algorithms with respect to

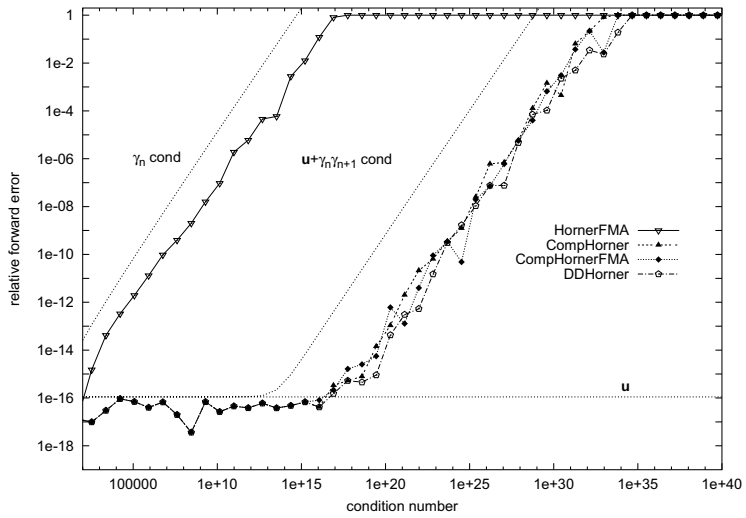

Figure 1. Accuracy of the polynomial evaluations.

HornerFMA. We have reported the minimum, the mean and the maximum of these ratios. The theoretical overheads (resulting from the number of floating point operations involved by each algorithm) are also reported.

Our compensated algorithms CompHornerFMA and CompHorner are both significantly faster than DDHorner. Algorithm CompHorner seems to be the most efficient alternative to improve the accuracy of the Horner algorithm. It runs about 1.8 times faster than CompHornerFMA and more than two times faster than DDHorner that uses the double-double library. We also notice that the measured overheads are always significantly smaller than theoretically expected. This issue will be explained in last Section 5.5.

5. Dot product

The purpose is now to compare the classic dot product algorithm without and with the use of FMA and corresponding compensated algorithms.

5.1. Classic dot product algorithm

Let $x = (x_1, \ldots, x_n)^T$ and $y = (x_1, \ldots, x_n)^T$ be n-vectors with floating point elements. The classic algorithm to compute a dot product is the following.

Algorithm 10. Dot product

function $s_n = \mathsf{Dot}(x, y)$
  $s_1 = x_1 y_1$
  for $i = 2 : n$
    $s_i = x_i y_i + s_{i-1}$
  end

Algorithm 10 requires $2n-1$ flops. The computed result satisfies [9]

$$|\mathsf{Dot}(x, y) - x^T y| \leq \gamma_n |x^T||y|. \tag{12}$$

FMA is suitable for dot product algorithm. We now look at the dot product algorithm where we use the FMA instead of the classic multiplication and addition.

Algorithm 11. Dot product with FMA.

function $s_n = \mathsf{DotFMA}(x, y)$
  $s_1 = x_1 y_1$
  for $i = 2 : n$
    $s_i = \mathsf{FMA}(x_i, y_i, s_{i-1})$
  end

As we can see, the number of floating point operations is divided by two when the FMA is used: algorithm DotFMA only requires n floating point operations. Nevertheless, the FMA does not improve the worst case accuracy of the computed dot product. Indeed, the two previous algorithms share the same error bound (12). Let us remark that the γ_n factor does not describe the number of floating point operations but the length of the largest path from the data to the result in the data flow graph.

5.2. Compensated DotFMA

Again $x = (x_1, \ldots, x_n)^T$ and $y = (x_1, \ldots, x_n)^T$ are two n-vectors with floating point elements. We consider the following compensated version of DotFMA. The rounding errors generated by every FMA are computed thanks to ThreeFMA (Algorithm 3).

Algorithm 12. Compensated DotFMA.

Table 3. Experimental environments

environment	description
I	Intel Itanium I, 733MHz, GNU Compiler Collection 2.96
II	Intel Itanium II, 1.5GHz, GNU Compiler Collection 3.4.6
III	Intel Itanium I, 733 MHz (16KB L1, 96KB L2 cache), Intel C++ Compiler v9.0.
IV	Intel Itanium II, 1.6 GHz (32KB L1, 256KB L2 cache), Intel C++ Compiler v9.0.

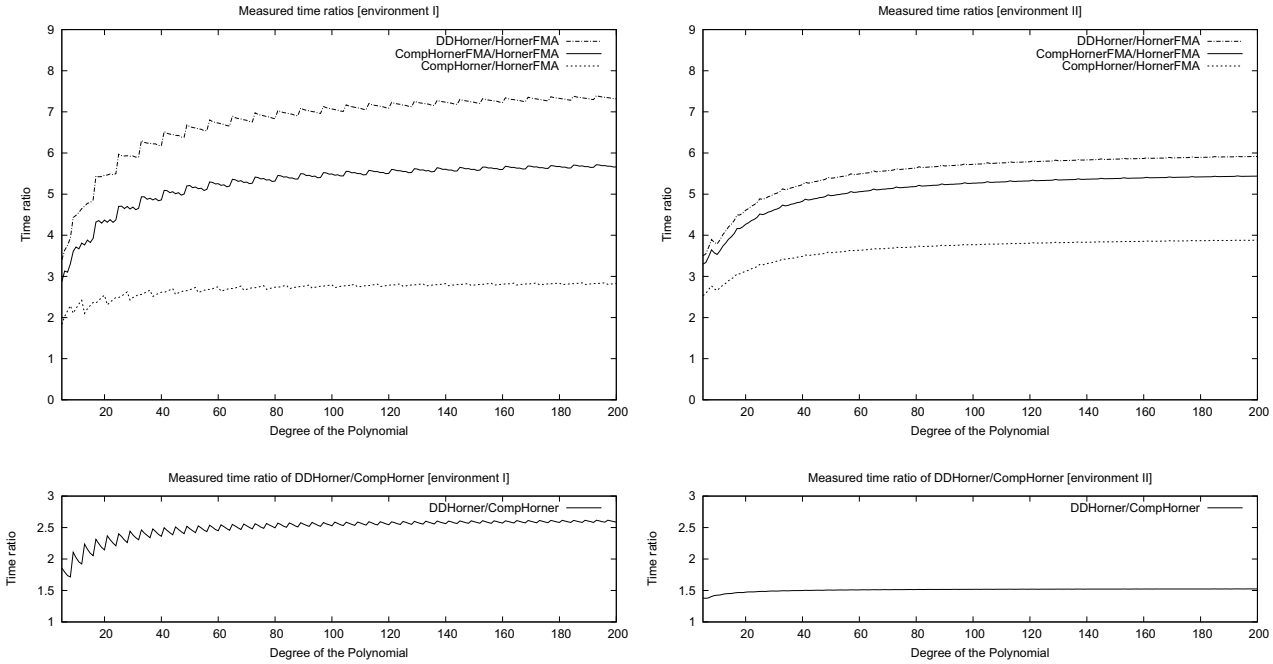

Figure 2. Measured overhead for CompHornerFMA**,** CompHorner **and** DDHorner **with respect to the polynomial degree (environment I on the left, and II on the right).**

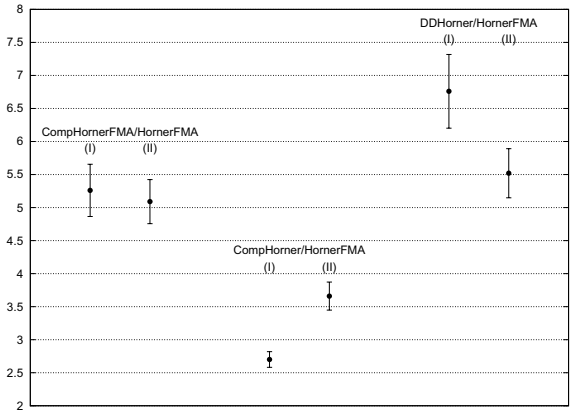

Figure 3. Overhead of CompHornerFMA**,** CompHorner **and** DDHorner **compared to** HornerFMA**: mean values are reported together with mean absolute deviation (obtained from Figure 2).**

Table 4. Measured running-time overhead compared to theoretical values for polynomial evaluation.

environment	CompHornerFMA/HornerFMA				CompHorner/HornerFMA				DDHorner/HornerFMA			
	min.	mean	max.	theo.	min.	mean	max.	theo.	min.	mean	max.	theo.
I	2.9	5.3	5.7	19	1.8	2.7	2.8	10	3.4	6.8	7.4	20
II	3.3	5.1	5.4	19	2.5	3.7	3.9	10	3.5	5.5	5.9	20

function $r = \mathsf{CompDotFMA}\,(x, y)$
  $[s_1, c_1] = \mathsf{TwoProd}\,(x_1, y_1)$
  for $i = 2 : n$
    $[s_i, \alpha_i, \beta_i] = \mathsf{ThreeFMA}\,(x_i, y_i, s_{i-1})$
    $c_i = c_{i-1} \oplus (\alpha_i \oplus \beta_i)$
  end
  $r = s_n \oplus c_n$

Proposition 7. *The result computed by previous Algorithm 12 satisfies*

$$|\mathsf{CompDotFMA}\,(x, y) - x^T y| \leq \mathbf{u}|x^T y| + \mathbf{u}\gamma_{n+1}|x|^T|y|. \quad (13)$$

CompDotFMA *requires* $19n-16$ *floating point operations.*

Proofs are detailed in [6].

5.3. Compensating Dot

The following algorithm for dot product computation is due to Ogita, Rump and Oishi [16].

Algorithm 13. Compensated Dot.

function $r = \mathsf{CompDot}\,(x, y)$
  $[s_1, c_1] = \mathsf{TwoProd}\,(x_1, y_1)$
  for $i = 2 : n$
    $[p_i, \pi_i] = \mathsf{TwoProd}\,(x_i, y_i)$
    $[s_i, \sigma_i] = \mathsf{TwoSum}\,(p_i, s_{i-1})$
    $c_i = c_{i-1} \oplus (\pi_i \oplus \sigma_i)$
  end
  $r = s_n \oplus c_n$

The following proposition sums up the properties of this algorithm.

Proposition 8 ([16]). *If no underflow occurs, the result computed by Algorithm 13 satisfies*

$$|\mathsf{CompDot}\,(x, y) - x^T y| \leq \mathbf{u}|x^T y| + \gamma_n^2|x^T||y|, \quad (14)$$

CompDot *algorithm requires* $10n - 7$ *flops when the* FMA *is available.*

Let us note again that this proposition means that the compensated algorithm returns a computed dot product as accurate as if computed in twice the working precision.

Experimental scheme for the following testing of accuracy and running-time issues is similar to the one we have described in Section 4.4.

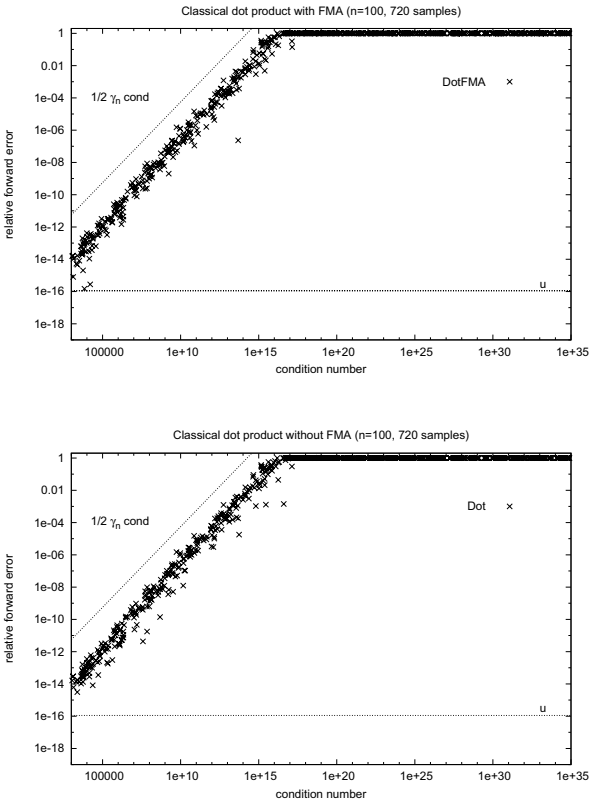

Figure 4. Accuracy of classic dot product algorithm with and without FMA

5.4. Accuracy Tests

For testing the actual accuracy reached by the various dot product algorithms previously presented, we need to generate dot products with condition number up to about 10^{32}. For this purpose, we use the random generator of ill-conditioned dot product GenDot described in [16]. Here it allows us to generate 720 dot products of length $n = 100$, with condition numbers varying from 10^2 to 10^{35}. We always use the same set of dot products in all our experiments.

Figure 4 presents the results for classic dot product algorithm with and without FMA. We display the relative error of the computed result with respect to the condition num-

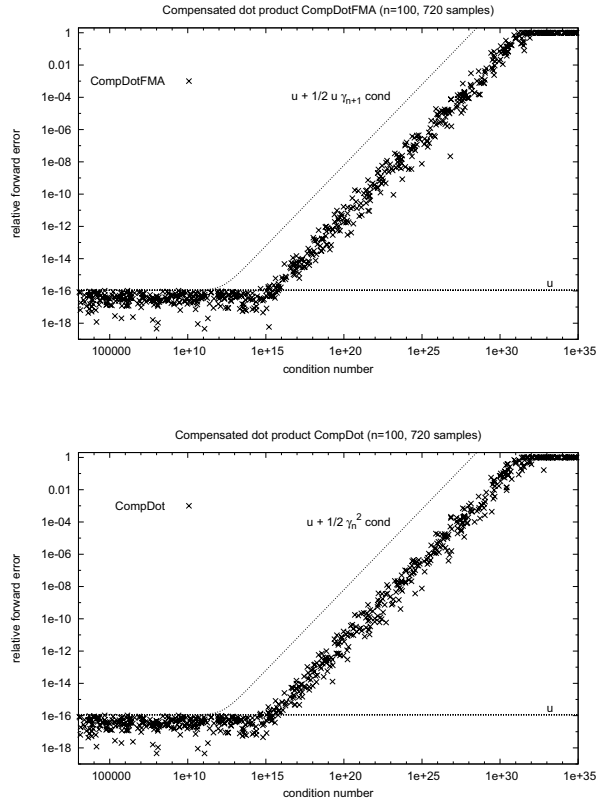

Figure 5. Accuracy of compensated dot products CompDotFMA **and** CompDot

ber. The dashed curves represent the relative error bounds derived from Relations (13) and (14). As we can see, the use of FMA does not significantly improve the accuracy of the result. Even if the theoretical bounds are pessimistic, they provide a reasonable estimate of the actual error bounds.

Figure 5 presents the results for compensated dot products CompDotFMA and CompDot. Again this figure illustrates that using the error-free transformation ThreeFMA does not improve the accuracy of the result compared to the error-free transformation TwoProd. Finally the accuracy of the considered algorithms does not benefit from the use of FMA as it was expected from the theoretical worst case bounds.

5.5. Running-time tests

The measured execution times are reported with Table 5. The timings are compared with ordinary dot product algorithm DotFMA. Last row also reports the theoretical ratios.

These results show that the compensated algorithms CompDotFMA and CompDot run both considerably faster than DDDot.

Table 5. Running-time ratios of dot products. Environment III (top) and IV (middle) are compared to theoretical flop counts (bottom) for various vector lengths n.

n	CompDot / DotFMA	CompDotFMA / DotFMA	DDDot / DotFMA
50	1.4	2.3	8.24
100	1.29	2.37	8.98
1000	1.24	2.63	10.46
10000	1.25	2.63	10.5
100000	1.07	1.76	6.27
50	1.63	2.61	9.87
100	1.35	2.43	9.65
1000	1.26	2.6	10.86
10000	1.25	2.62	10.97
100000	1.25	2.35	9.8
Theoret.	10	19	22

As previously observed for polynomial evaluation, the measured ratios of the compensating process overhead are always smaller than the theoretical values. Theoretical ratios just count the floating point operations and do not take into account the complex instruction reordering the compiler or the processor perform. Most modern processors are capable of executing several instructions in parallel, but it is not always easy to exploit this feature in real programs. In order to exploit the ability to perform multiple instructions in parallel, both the compiler and the processor must reconstruct the implicit parallelism in a program which is usually written in a serial fashion. In particular, the main part of the instruction scheduling is performed by the compiler to take advantage of the instruction-level parallelism on Intel Itanium architecture. On the other hand, the possibility of performing parallel execution of instructions is not only limited by the architecture and the compiler performances, but also by the instruction-level parallelism which is an intrinsic parameter of the algorithm. For instance a program may require long sequences of serial instructions that can not be performed in parallel with any other. Compensated algorithms here exhibit a better intrinsic instruction-level parallelism than double-double ones since they are implemented with no normalization step.

6. Acknowledgment

Authors thank T. Ogita, S.M. Rump and S. Oishi for [16] that motivates the analysis and development of compensated algorithms. They also thank S. Graillat (LIP6, UPMC Paris) for his contribution to previous results about compensated algorithms.

References

[1] High-precision software directory. URL = `http://crd.lbl.gov/~dhbailey/mpdist`.

[2] D. H. Bailey. Algorithm 719: Multiprecision translation and execution of Fortran programs. *ACM Trans. Math. Software*, 19(3):288–319, 1993.

[3] S. Boldo and J.-M. Muller. Some functions computable with a fused mac. In IEEE, editor, *IEEE Symposium on Computer Arithmetic ARITH'17*, Cape Cod, Massachusetts, USA, June 2005.

[4] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw.*, 4(1):57–70, 1978.

[5] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

[6] S. Graillat, P. Langlois, and N. Louvet. Accurate dot products with FMA. In G. Hanrot and P. Zimmermann, editors, *RNC-7, Real Numbers and Computer Conference, Nancy, France*, pages 141–142, July 2006. Extended version available on-line.

[7] S. Graillat, P. Langlois, and N. Louvet. Fused Multiply and Add implementations of the compensated Horner scheme. In P. Hertling, C. Hoffmann, W. Luther, and N. Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, Dagstuhl Seminar 6021, Jan. 2006. Extended version available on-line.

[8] Y. Hida, X. S. Li, and D. H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. In N. Burgess and L. Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162. Institute of Electrical and Electronics Engineers, June 2001.

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

[10] IEEE Computer Society, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[11] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[12] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, 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, June 2002.

[13] P. Markstein. *IA-64 and elementary functions. Speed and precision*. Hewlett-Packard Professionnal Books. Prentice-Hall PTR, 2000.

[14] The MPFR library. URL = `http://www.mpfr.org/`.

[15] Y. Nievergelt. Scalar fused multiply-add instuctions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software*, 29(1), Mar. 2003.

[16] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.