More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms

Philippe Langlois, Nicolas Louvet

▶ **To cite this version:**

Philippe Langlois, Nicolas Louvet. More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms. 2007. hal-00165020

HAL Id: hal-00165020
https://hal.science/hal-00165020v1

Preprint submitted on 24 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms

Philippe Langlois and Nicolas Louvet
langlois@univ-perp.fr,nicolas.louvet@univ-perp.fr

***Abstract*— The compensated Horner algorithm and the Horner algorithm with double-double arithmetic improve the accuracy of polynomial evaluation in IEEE-754 floating point arithmetic. Both yield a polynomial evaluation as accurate as if it was computed with the classic Horner algorithm in twice the working precision. Both algorithms also share the same low-level computation of the floating point rounding errors and cost a similar number of floating point operations. We report numerical experiments to exhibit that the compensated algorithm runs at least twice as fast as the double-double one on modern processors. We propose to explain such efficiency by identifying more instruction level parallelism in the compensated implementation. Such property also applies to other compensated algorithms for summation, dot product and triangular linear system solving. More generally this paper illustrates how this kind of performance analysis may be useful to highlight the actual efficiency of numerical algorithms.**

***Index Terms*— Accurate polynomial evaluation, Horner algorithm, compensated Horner algorithm, floating point arithmetic, IEEE-754 standard, instruction level parallelism, performance evaluation.**

I. Introduction

In this paper, we consider polynomial evaluation in floating point arithmetic restricted to entries and polynomial coefficients being floating point values. Such cases appear for example when evaluating elementary functions [1] and in geometric computations where accurate polynomial evaluation is crucial [2], [3].

A. Accurate Polynomial Evaluation

The following inequality bounds the accuracy of the floating point result $\widehat{p}(x)$ of the polynomial evaluation $p(x)$, for example with the classic Horner algorithm. We have

$$\frac{|p(x) - \widehat{p}(x)|}{|p(x)|} \leq \mathrm{cond}(p,x) \times \mathcal{O}(\mathbf{u}), \tag{1}$$

where $\mathbf{u}$ is the computing precision and the condition number $\mathrm{cond}(p,x)$ is a scalar larger than 1 that only depends on the entry x and on p coefficients —its expression will be given further. Hence the computed value $\widehat{p}(x)$ suffers from less exact digits than what the computing precision provides. This loss of accuracy may be arbitrarily large as evaluating the polynomial p at the x entry is more ill-conditioned, as for example in the neighborhood of a multiple root.

When the computing precision $\mathbf{u}$ is not sufficient (compared to $\mathrm{cond}(p,x)$) to guarantee a desired accuracy in $\widehat{p}(x)$, several solutions implementing a computation with more bits exist. Double-double algorithms are well-known and well-used solutions to simulate twice the IEEE-754 double precision [4], [5]. The compensated Horner algorithm introduced in [6] is an alternative to the Horner algorithm implemented with double-double arithmetic. In both cases the accuracy of computed $\widehat{p}(x)$ is improved and now verifies

$$\frac{|p(x) - \widehat{p}(x)|}{|p(x)|} \leq \mathbf{u} + \mathrm{cond}(p,x) \times \mathcal{O}(\mathbf{u}^2). \tag{2}$$

Comparing to Relation (1), this relation means that the computed value is now as accurate as the result of the Horner algorithm performed in twice the working precision with a final rounding back to this working precision —the same behavior is mentioned in [7] for compensated summation and dot product.

As for Relation (1) the accuracy of the compensated result still depends on the condition number and may be arbitrarily bad for ill-conditioned polynomial evaluations. Nevertheless, this bound tells us that the compensated Horner algorithm may yield a full precision accuracy for not too ill-conditioned polynomials, that is for p and x such that the second term $\mathrm{cond}(p,x) \times \mathcal{O}(\mathbf{u}^2)$ is small compared to the working precision $\mathbf{u}$. In [8] we prove that the compensated evaluation is faithfully rounded for condition numbers up to $\mathcal{O}(\mathbf{u}^{-1})$. By faithful rounding we mean that the computed result is one of the two floating point neighbors of the exact result $p(x)$. We also provide a dynamical test to answer to the question "is the computed compensated result a faithful rounding of the exact evaluation?" thanks to a computable and validated bound for the final absolute error in $\widehat{p}(x)$.

B. Previous Results and Motivation for Efficiency Analysis

Compensated Horner evaluation is fast. By fast we mean that it runs at least twice as fast as the double-double Horner counterpart still providing the same output accuracy. The implementation core of these "double-like" algorithms is the computation of the rounding errors generated by the floating point operators. For compensated implementations, these rounding errors are used to correct the result of the original algorithm. Such low-level computation depends on the arithmetic attributes. In [9] we present experimental results to exhibit how to benefit from the fused multiply and add operator. It appears that FMA should be avoided in the evaluation part of the compensated algorithm but preferred when computing the rounding errors. Measures showing the efficiency of the compensated Horner algorithm also when FMA is not available are briefly presented in [8].

In both cases these experiments illustrate the practical efficiency of the compensated algorithm we announced before. Nevertheless we were not able to explain why the measured overhead factor introduced by the compensated evaluation

Ph. Langlois and N. Louvet are members of DALI Research Team, ELIAUS Laboratory at Université de Perpignan, France.

TABLE I
HOW TO EXPLAIN THE OVERHEAD DIFFERENCES BETWEEN FLOP COUNTS AND MEASURED TIMES?

	Horner	**Compensated Horner**	**Horner with double-double**
Number of flop	$2n$	$22n+5$	$28n+4$
Overhead w.r.t. Horner: number of flop	1	≈ 11	≈ 14
Overhead w.r.t. Horner: range of measured times	1	$2.7-3.2$	$8.5-9.7$

algorithm is significantly better than the one introduced by its double-double counterpart. Counting of floating point operations is still commonly used in the field of numerical analysis to compare the performances of different numerical algorithms. But this classic technique is clearly not sufficient to answer to the open question we address here, as this is summarized in Table I. How to explain that computed Horner actually runs twice as fast as the double-double Horner whereas their flop counts are very similar? Let us remark that the same property is identified but still unexplained for summation and dot product in [7].

In this paper we propose to answer to this open question presenting how the actual efficiency of the compensated Horner algorithm can be explained thanks to its *instruction level parallelism* (ILP). Quoting Hennessy and Patterson [10, p.172], "all processors since about 1985 ...use pipelining to overlap the execution of instructions and improve performances. This potential overlap among instructions is called instruction-level parallelism since the instructions can be evaluated in parallel."

We propose a detailed analysis of the ILP of compensated Horner and Horner with double-double algorithms. We quantify the average number of instructions that can be theoretically executed in one clock cycle on an ideal processor. This ideal processor is one where all the artificial constraints on ILP are removed [10, p.240]. In this context, the theoretical IPC (instructions per clock) is about six times better for the compensated Horner algorithm than for its double-double counterpart. Every double-double arithmetic operation ends with a renormalization step [4], [5]. We also show that avoiding these renormalization steps the compensated Horner algorithm presents more ILP than the Horner algorithm with double-double arithmetic. We conclude that the compensated algorithm exhibits more potential to benefit from the superscalar facilities of modern processors. In our point of view, this gives a qualitative explanation of its practical efficiency.

C. Outline

The paper is organized as follows. In Section II we describe the main steps from the classic Horner algorithm to the compensated Horner algorithm. This Section summarizes some results already presented in [8]: error free transformations of arithmetic operations, extension to an error free transformation of the Horner polynomial evaluation, final correction and the corresponding theoretical accuracy bound. We present experimental measures of the running times for the compensated Horner algorithm and the challenging Horner with double-double arithmetic in Section III. Since the classic flop count fails to explain these observed results we devote the last Section IV to a detailed comparison of these two algorithms. We introduce the notions of ILP and IPC on an ideal processor. Then we highlight the common parts within the compensated and the double-double Horner algorithms and prove that the compensated implementation benefits for more ILP than the double-double one. Appendix contains all the measures previously analyzed in Section III.

D. Notations and Hypothesis

Throughout the paper, we assume a floating point arithmetic compliant with the IEEE-754 floating point standard [11]. 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 [12, 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 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 $\oplus$, $\ominus$, $\otimes$ and $\oslash$ represent respectively the floating point addition, subtraction, multiplication and division. For more complicated arithmetic expressions, $\mathrm{fl}(\cdot)$ 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, $a \oplus b = \mathrm{fl}(a+b)$.

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 $\mathrm{fl}(a \circ b)$ of $a \circ b$ is such that

$$\mathrm{fl}(a \circ b) = (a \circ b)(1+\varepsilon_1) = (a \circ b)/(1+\varepsilon_2), \text{with} \quad |\varepsilon_1|, |\varepsilon_2| \leq \mathbf{u}. \tag{3}$$

It is classic to keep track of the $(1+\varepsilon)$ factors when nesting k arithmetic operations using $\gamma_k := k\mathbf{u}/(1-k\mathbf{u})$. We have $\prod_{i=1}^{k}(1+\varepsilon_i)^{\pm 1} \leq 1+\gamma_k$ [12, chap. 3]. When using these notations, we always implicitly assume $k\mathbf{u} < 1$ and $k > 0$.

II. FROM HORNER TO COMPENSATED HORNER ALGORITHM

The compensated Horner algorithm improves the classic Horner iteration by computing a correcting term to compensate the rounding errors the Horner iteration generates in floating point arithmetic. Main results about compensated Horner algorithm are summarized in this section and may be skipped if [8] is already known.

A. Polynomial evaluation and Horner algorithm

The classic condition number of the evaluation of $p(x) = \sum_{i=0}^{n} a_i x^i$ at a given entry x is [13]

$$\mathrm{cond}(p,x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|\sum_{i=0}^{n} a_i x^i|} := \frac{\widetilde{p}(x)}{|p(x)|}. \tag{4}$$

For any floating point value x we denote by $\mathsf{Horner}(p,x)$ the result of the floating point evaluation of the polynomial p at x using the classic Horner algorithm recalled below.

Algorithm 1 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

The accuracy of Algorithm 1 verifies introductory inequality (1) with $\mathcal{C}(\mathbf{u}) = \gamma_{2n}$ and previous condition number (4). Clearly, the condition number $\mathrm{cond}(p,x)$ 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.

More accuracy can be reached at the same computing precision thanks to error free transformation (EFT). We review well known results concerning the EFT of the elementary floating point operations $+$, $-$ and $\times$. Then we introduce an EFT for polynomial evaluation proving that the error generated by the Horner algorithm is exactly the sum of two polynomials with floating point coefficients.

B. EFT for the elementary operations

Let $\circ$ be an operator in $\{+,-,\times\}$, a and b be two floating point numbers, and $\widehat{x} = \mathrm{fl}(a \circ b)$. Then there exists a floating point value y such that $a \circ b = \widehat{x} + y$. The difference y between the exact result and the computed result is the rounding error generated by the computation of $\widehat{x}$. Let us emphasize that this relation between four floating point values relies on real operators and exact equality, *i.e.*, not on approximate floating point counterparts. Ogita *et al.* [7] name such a transformation an error free transformation. The practical interest of the EFT comes from next Algorithms 2 and 3 that compute the exact error term y for $\circ = +$ and $\circ = \times$.

For the EFT of the addition we use the well known TwoSum algorithm by Knuth [14, p.236] that requires 6 flop (floating point operations). TwoProd by Veltkamp and Dekker [15] performs the EFT of the product and requires 17 flop. We also describe Dekker's FastTwoSum version of the EFT for the addition that will be used further for double-double computation.

Algorithm 2 EFT of the sum of two floating point numbers

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

We notice that algorithms TwoSum and TwoProd only require well optimizable floating point operations. They apply for the IEEE rounding to the nearest rounding mode. They do not use branches, nor access to the mantissa that can be time-consuming. FastTwoSum costs less flop than TwoSum but only applies to sorted entries. When this condition is not *a priori* satisfied, it is well known that a dynamic sorting ruins the actual performances

Algorithm 3 EFT of the product of two floating point numbers

function $[x,y] = \mathsf{TwoProd}(a,b)$
  $x = a \otimes b$
  % splitting of a and b to high and low parts
  % $splitter = (1 + 2^{\lfloor t \rfloor})$ is a predefined constant,
  % with t the mantissa length
  $a_s = splitter \otimes a$; $\quad b_s = splitter \otimes b$
  $a_h = a_s \ominus (a_s \ominus a)$; $\quad b_h = b_s \ominus (b_s \ominus b)$
  $a_l = a - a_h$; $\quad b_l = b - b_h$
  % rounding error in $a \otimes b$
  $y = a_l \otimes b_l \ominus (((x \ominus a_h \otimes b_h) \ominus a_l \otimes b_h) \ominus a_h \otimes b_l)$

Algorithm 4 EFT of the sum of two **sorted** floating point numbers

function $[x,y] = \mathsf{FastTwoSum}(a,b)$
  % **Assume** $|a| \geq |b|$
  $x = a \oplus b$
  $y = (a \ominus x) \oplus b$

of FastTwoSum on superscalar processors [7]. We also mention that a significant improvement of TwoProd is defined when a Fused-Multiply-and-Add operator is available as Intel Itanium or IBM PowerPC [16]. We detailed how to benefit from such instruction in [9].

C. An EFT for the Horner algorithm

The next EFT for the polynomial evaluation with the Horner algorithm exhibits the exact rounding error generated by the Horner algorithm together with an algorithm to compute it.

Algorithm 5 EFT for the Horner algorithm

function $[r_0, p_\pi, p_\sigma] = \mathsf{EFTHorner}(p,x)$
  $r_n = a_n$
  for $i = n-1 : -1 : 0$
    $[p_i, \pi_i] = \mathsf{TwoProd}(r_{i+1}, x)$
    $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
    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 1 ([8]): 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 Algorithm 5 computes both $\mathsf{Horner}(p,x)$ and two polynomials p_π and p_σ of degree $n-1$ with floating point coefficients, such that $[\mathsf{Horner}(p,x), p_\pi, p_\sigma] = \mathsf{EFTHorner}(p,x)$. If no underflow occurs, the polynomial evaluation verifies

$$p(x) = \mathsf{Horner}(p,x) + (p_\pi + p_\sigma)(x). \tag{5}$$

Relation (5) means that algorithm EFTHorner is an EFT for polynomial evaluation with the Horner algorithm.

D. Compensated Horner algorithm

From Theorem 1 the forward error in the floating point evaluation of $p(x)$ with the Horner algorithm is

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

where both polynomials p_π and p_σ are exactly identified by EFTHorner (Algorithm 5) —this latter also computes Horner (p, x). Therefore, the key of the compensated algorithm is to compute, in the working precision, the approximate Horner $(p_\pi \oplus p_\sigma, x)$ of the final error c and then a corrected result

$$\overline{r} = \mathsf{Horner}\,(p, x) \oplus \mathsf{Horner}\,(p_\pi \oplus p_\sigma, x)\,.$$

We say that Horner $(p_\pi \oplus p_\sigma, x)$ is a correcting term for Horner (p, x). The compensated result $\bar{r}$ is expected to be more accurate than Horner (p, x) as proved in next section. The next Algorithm 6 implements the compensated Horner algorithm within only one loop inlining the computation of the Horner EFT (Algorithm 5), the computation of the correcting term Horner $(p_\pi \oplus p_\sigma, x)$ and the final correction.

Algorithm 6 Compensated Horner algorithm

function $\overline{r} = \mathsf{CompHorner}(P, x)$
  $r_n = a_i$; $c_n = 0$
  for $i = n-1 : -1 : 0$
    $[a_i, \pi_i] = \mathsf{TwoProd}(r_{i+1}, x)$
    $[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
    $c_i = c_{i+1} \otimes x \oplus (\pi_i \oplus \sigma_i)$
  end
  % Here $r_0 = \mathsf{Horner}\,(p, x)$,
  % and $c_0 = \mathsf{Horner}\,(p_\pi \oplus p_\sigma, x)$
  $\overline{r} = r_0 \oplus c_0$

E. Accuracy of the Compensated Horner Algorithm

Next result proves that the result of a polynomial evaluation computed with the compensated Horner algorithm (Algorithm 6) is as accurate as if computed by the Horner algorithm using twice the working precision and then rounded to the working precision.

Theorem 2 ([8]): Consider a polynomial p of degree n with floating point coefficients, and x a floating point value. If no underflow occurs,

$$|\mathsf{CompHorner}\,(p, x) - p(x)| \leq \mathbf{u}|p(x)| + \gamma_{2n}^2\,\widetilde{p}(x). \qquad (6)$$

It is interesting to interpret the previous theorem in terms of the condition number of the polynomial evaluation of p at x. Combining the error bound (6) with the condition number (4) of polynomial evaluation gives the precise writing of our introductory inequality (2),

$$\frac{|\mathsf{CompHorner}\,(p, x) - p(x)|}{|p(x)|} \leq \mathbf{u} + \gamma_{2n}^2\,\mathrm{cond}(p, x). \qquad (7)$$

Since $\gamma_{2n}^2 = \mathcal{O}(\mathbf{u}^2)$ relation (7) essentially tells us that the compensated result is as accurate as if computed by the classic Horner algorithm in twice the working precision, with a final rounding back to the working precision [8].

Fig. 1 illustrates the accuracy behavior of Horner and CompHorner w.r.t. the condition number. More detailed experiments may be found in [8]. We generate polynomials of degree 50 whose condition numbers vary from about 10^2 to 10^{35}. We see that even for small condition numbers we already lose some accuracy with the Horner evaluation. We also observe that the compensated algorithm exhibits the expected behavior : as long as the condition number is smaller than $\mathbf{u}^{-1}$, the relative error is of the order of the working precision $\mathbf{u}$. Then, for condition numbers between $\mathbf{u}^{-1}$ and $\mathbf{u}^{-2}$, this relative error degrades to no accuracy at all. As usual, the *a priori* error bound (7) appears to be pessimistic by many orders of magnitude.

III. Experimental Results for Performance Analysis

Now we start to focus the open question that motivates this paper presenting our implementations and the corresponding measured running-times for Compensated Horner algorithm and Horner algorithm with double-double.

A. Implementation of the Compensated Horner Algorithm

Since every loop of Horner (Algorithm 1) includes a multiplication by x, every loop of CompHorner (Algorithm 6) introduces one TwoProd applied to the same x. Hence the split of x is only performed once (out of the loop) to reduce the flop count. The following C code implements this simplification. Algorithm CompHorner now requires $22n + 5$ flops.

```
double CompHorner(double *P, unsigned int n, double x) {
  double p, r, c, pi, sig, x_hi, x_lo, hi, lo, t;
  int i;

  /* (x_hi, x_lo) = Split(x) */
  t = x*splitter; x_hi = t-(t-x); x_lo = x-x_hi;

  r = P[n]; c = 0.0;
  for(i=n-1; i>=0; i--) {
    /* (p, pi) = TwoProd(r, x); */
    p = r*x;
    t = r*_splitter_; hi = t-(t-r); lo = r-hi;
    pi = (((hi*x_hi-p)+hi*x_lo)+lo*x_hi)+lo*x_lo;

    /* (s, sigma) = TwoSum(p, P[i]); */
    r = p+P[i];
    t = r-p;
    sig = (p-(r-t)) + (P[i]-t);

    /* Computation of the correcting term */
    c = c*x+(pi+sig);
  }
  return(r+c);
}
```

B. Horner Algorithm with Double-Double Computation

In Algorithm 7, we implement the Horner algorithm performed with double-double arithmetic.

Algorithm 7 Horner algorithm with double-doubles

function $r = \mathsf{DDHorner}(P, x)$
  $sh_n = a_i$; $sl_n = 0$
  for $i = n-1 : -1 : 0$

% double-double = double-double $\times$ double:
% $(ph_i, pl_i) = (sh_{i+1}, sl_{i+1}) \otimes x$
$[th, tl] = \mathsf{TwoProd}(sh_{i+1}, x)$
$tl = sl_{i+1} \otimes x \oplus tl$
$[ph_i, pl_i] = \mathsf{FastTwoSum}(th, tl)$

% double-double = double-double + double:
% $(sh_i, sl_i) = (ph_i, pl_i) \oplus a_i$
$[th, tl] = \mathsf{TwoSum}(ph_i, a_i)$
$tl = tl \oplus pl_i$
$[sh_i, sl_i] = \mathsf{FastTwoSum}(th, tl)$

  end
  $r = sh_0$

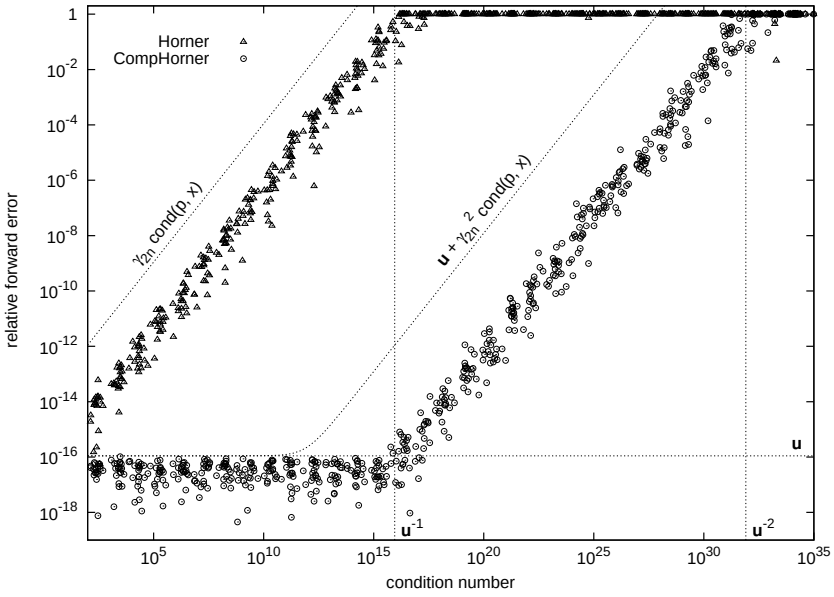


Fig. 1. Measured accuracy and theoretical bounds of Horner and Compensated Horner algorithms.

Double-doubles are managed as proposed by the authors of [5], [17]. For our purpose, it suffices to know that a double-double number a is the pair (ah, al) of IEEE-754 floating point numbers with $a = ah + al$ and $|al| \leq \mathbf{u}|ah|$. As we will see in the sequel, this property requires a renormalization step after every arithmetic operation with double-double values. To implement the Horner algorithm using the double-double format, we only need two basic operations: i) the product of a double-double number by a double number, and ii) the addition of a double number to a double-double number. These operations are represented in boxes in Algorithm 7. We notice that every double-double operation performs a final renormalization step using algorithm FastTwoSum (Algorithm 4). This renormalization is compulsory to ensure that the computed result of the corresponding double-double operation is still a valid double-double number. For example, the line $[sh_i, sl_i] = \mathsf{FastTwoSum}(th, tl)$ in Algorithm 4 ensures that the floating point pair (sh_i, sl_i) actually satisfies $|sl_i| \leq \mathbf{u}|sh_i|$ and so is a valid double-double number.

We also provide hereafter a C code implementation of algorithm DDHorner. We count that algorithm DDHorner requires $28n + 4$ flops.

```
double DDHorner(double *P, unsigned int n, double x) {
  double r_h, r_l, t_h, t_l, x_hi, x_lo, hi, lo, t;
  int i;

  /* (x_hi, x_lo) = Split(x) */
  t = x*splitter; x_hi = t-(t-x); x_lo = x-x_hi;

  r_h = P[n]; r_l = 0.0;
  for(i=n-1; i>=0; i--) {
    /* (r_h, r_l) = (r_h, r_l) * x */
    t = r_h*splitter; hi = t-(t-r_h); lo = (r_h-hi);
    t_h = r_h*x;
    t_l = ((((hi*x_hi-t_h)+hi*x_lo)+lo*x_hi)+lo*x_lo);
    t_l += r_l*x;
    r_h = t_h+t_l;
    r_l = (t_h-r_h)+t_l;

    /* (r_h, r_l) = (r_h, r_l) + P[i] */
    t_h = r_h+P[i];
    t = t_h-r_h;
    t_l = ((r_h-(t_h-t))+(P[i]-t))+r_l;
    r_h = t_h+t_l;
    r_l = (t_h-r_h)+t_l;
  }
  return(r_h);
}
```

C. Experimental results

All our experiments are performed using IEEE-754 double precision, and the algorithms are implemented in C code. The experimental environments are listed in Table II.

TABLE II
EXPERIMENTAL ENVIRONMENTS

env.	description
(I)	Intel Pentium 4, 3 GHz, GCC 4.1.2 `-std=c99 -march=pentium4 -mfpmath=387 -O3 -funroll-all-loops`
(II)	Intel Pentium 4, 3 GHz, ICC 9.1 `-c99 -mtune=pentium4 -O3 -funroll-loops -mp1`
(III)	Intel Pentium 4, 3 GHz, GCC 4.1.2 `-std=c99 -march=pentium4 -mfpmath=sse -O3 -funroll-all-loops`
(IV)	Intel Pentium 4, 3 GHz, ICC 9.1 `-c99 -mtune=pentium4 -msse2 -O3 -funroll-loops -fp-model source`
(V)	AMD Athlon 64 3200+, 2 GHz, GCC 4.1.2 `-std=c99 -march=athlon64 -m3dnow -O3 -funroll-all-loops`
(VI)	Itanium 2, 1.5 GHz, GCC 4.1.1 `-std=c99 -mtune=itanium2 -O3 -funroll-all-loops`
(VII)	Itanium 2, 1.5 GHz, ICC 9.1 `-c99 -O3 -mp -IPF_fma`

In this table, GCC denotes the GNU Compiler Collection and ICC the Intel C Compiler. In environments (I) to (V), no FMA instruction is available, so we use the C codes presented in the previous subsections to implement algorithms CompHorner and DDHorner. In environments (VI) and (VII), that is on the Intel Itanium architecture which provides a FMA instruction, we use

the improvements of these algorithms presented in [9]. Anyway we use the same programming techniques for the implementations of the routines CompHorner and DDHorner. All timings are done with the cache warmed to minimize the memory traffic overcost.

Our measures are performed with 39 polynomials whose degrees vary from 10 to 200 by step of 5. The coefficients and the argument of these polynomials are randomly generated. For every algorithm and every degree, we measure the ratio of the computing time over the computing time of the Horner algorithm.

We display the average value of these ratios for CompHorner and DDHorner in Table III. All detailed results for considered environments are presented in the Appendix.

TABLE III
AVERAGE MEASURED OVERHEAD IN CompHorner AND DDHorner W.R.T. Horner .

env.	CompHorner/Horner	DDHorner/Horner
(I)	2.8	8.5
(II)	2.7	9.0
(III)	3.0	8.9
(IV)	3.2	9.7
(V)	3.2	8.7
(VI)	2.9	7.0
(VII)	1.5	5.9

First we notice that the measured slowdown factors are always significantly smaller than expected if the flop count is only considered. We have indeed from previous flop counts,

$$\frac{\mathsf{CompHorner}}{\mathsf{Horner}} = \frac{22n+5}{2n} \approx 11,$$

whereas the compensated algorithm CompHorner is only about 3 times slower than the classic Horner algorithm. The same remark applies to algorithm DDHorner for which

$$\frac{\mathsf{DDHorner}}{\mathsf{Horner}} = \frac{28n+4}{2n} \approx 14,$$

and that appears to be only about 8 times slower than the Horner algorithm.

These results also show that the compensated algorithm is more than twice faster than the Horner algorithm with double-double computation while, as already noticed, previous comparison suggests

$$\frac{\mathsf{DDHorner}}{\mathsf{CompHorner}} = \frac{28n+4}{22n+5} \approx 1.3.$$

Flop counts are therefore not sufficient to explain such behaviors.

IV. MORE ILP IN CompHorner EXPLAINS ITS ACTUAL PERFORMANCE

It is well know that most modern processors are capable of executing several instructions concurrently. Quoting Hennessy and Patterson [10, p.172]:

> All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performances. This potential overlap among instructions is called instruction-level parallelism since the instructions can be evaluated in parallel.

As explained in Section I, the term instruction-level parallelism refers to the degree to which the instructions of a program can be executed in parallel. Real programs are usually written in a serial fashion, using high-level languages such as C and Fortran. To exploit the implicit parallelism available among the instructions of a given program, many hardware (processor) and software (compiler) techniques have been developed.

In the sequel, we first underline the main algorithmic difference between CompHorner and DDHorner: CompHorner and DDHorner perform essentially the same floating point operations, but the renormalization steps required for double-double computations are avoided in CompHorner. Next, we will prove that CompHorner presents as a consequence more ILP, which certainly explains its better practical performance on modern superscalar processors. We perform our analysis assuming that no FMA is available, that is considering only the implementations of CompHorner and DDHorner described in Section III.

A. Comparison between CompHorner and DDHorner

Let us now compare algorithms CompHorner and DDHorner. For this purpose, we consider below a slightly modified version of the compensated Horner algorithm. Compared to Algorithm 6, we only reorder the floating point operations involved in the computation of c_i with respect to c_{i+1}, x , π_i and σ_i.

function $r = \mathsf{CompHorner}'(P, x)$
$r_n = a_i$; $c_n = 0$
for $i = n-1 : -1 : 0$

$[a_i, \pi_i] = \mathsf{TwoProd}(r_{i+1}, x)$
$t = c_{i+1} \otimes x \oplus \pi_i$

$[r_i, \sigma_i] = \mathsf{TwoSum}(p_i, a_i)$
$c_i = t_i \oplus \sigma_i$

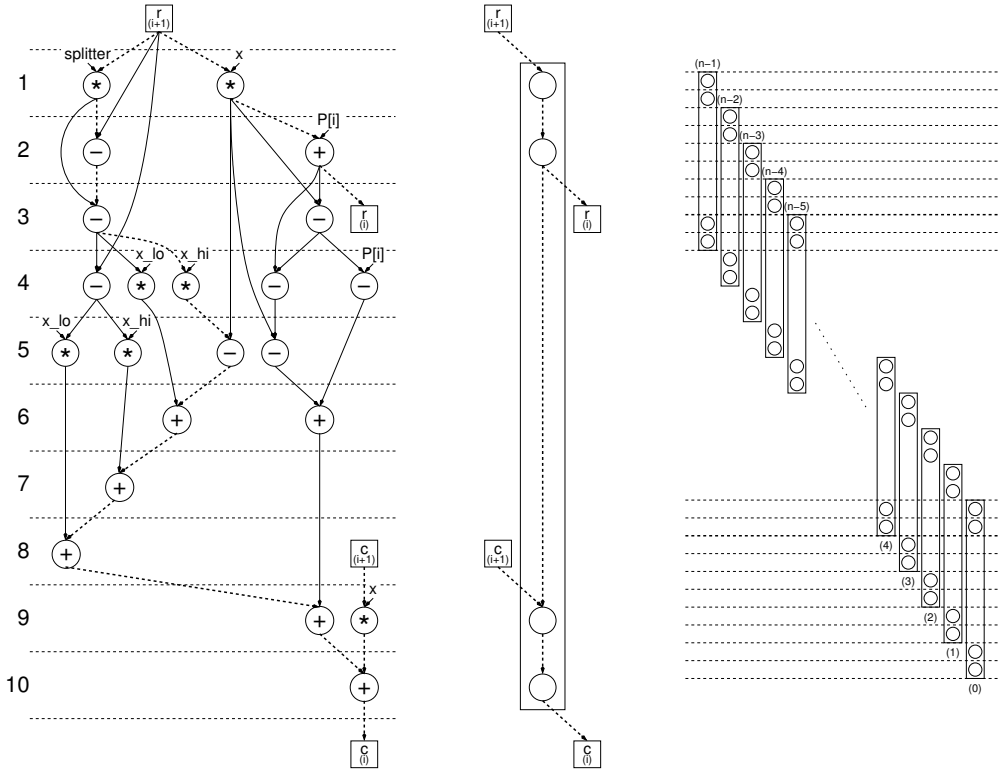
end
$r = r_0 \oplus c_0$

Comparing the previous algorithm with Algorithm 7, it is clear that CompHorner and DDHorner perform all the same floating point operations but the renormalization steps needed in algorithm DDHorner. The lines $[ph_i, pl_i] = \mathsf{FastTwoSum}(th, tl)$ and $[sh_i, sl_i] = \mathsf{FastTwoSum}(th, tl)$ in DDHornerare avoided in CompHorner and so this save $6n$ flops. Saving $6n$ flop is clearly not sufficient to explain the practical performance of CompHorner compared to DDHorner. Nevertheless, we will see in the next subsections that thanks to the suppression of the renormalization steps, the compensated evaluation algorithm introduces more ILP than DDHorner.

B. IPC and the ideal processor

A way to evaluate the ILP available in a given program is to compute its ICP (instruction per clock) on an ideal processor [10, p.240]. The IPC of a program, running on a given processor, is the average number of instructions of this program executed per clock cycle. An ideal processor is one where all artificial constraints on ILP are removed. The only limits in such a processor are those imposed by the actual data flows through either register or memory. More precisely, we assume that:

- the processor can execute an unlimited number of independent instructions in the same clock cycle;



(a) (b) (c)

Fig. 2. Data-flow graphs for algorithm CompHorner.

- all but true data dependencies are removed: any instruction in the program execution can be scheduled on the cycle immediately following the execution of the predecessor one on which it depends;
- branches are perfectly predicted: all conditional branches are predicted exactly;
- memory accesses are also perfect: all loads and stores always complete in one clock cycle.

These assumptions mean that this ideal processor can execute arbitrarily many operations in parallel, and that any sequence of dependent instructions can execute on successive cycles. In the sequel, we will refer to the IPC of a program running on such an ideal processor as its ideal IPC.

Now, let us study algorithm CompHorner running on this ideal processor when evaluating a polynomial of degree n: n iterations (numbered from $n-1$ down to 0) of the inner loop of algorithm CompHorner are then executed. We want to find the total latency of the execution of the inner loop, which is the number of clock cycles elapsed when executing n iterations.

C. Ideal IPC of CompHorner

We consider on Fig. 2.a the data-flow graph iteration i of the main loop of algorithm CompHorner. This data-flow graph is based on the C code implementation of CompHorner provided in Section III. The inputs in square boxes are critical inputs since they are the outputs of the previous iteration $i+1$. We can distinguish three critical paths of interest (represented with dashed edges) in this data-flow graph:

- one from r_{i+1} to r_i containing 2 instructions,
- one from r_{i+1} to c_i containing 10 instructions,
- one from c_{i+1} to c_i containing 2 instructions.

Since the critical path from r_{i+1} to c_i has length 10, the whole iteration can be executed within 10 cycles with the ideal processor. From these remarks, we represent on Fig. 2.b the execution of iteration i as a box of length 10 cycles, where:

- r_{i+1} is consumed at the first cycle of the iteration,
- r_i is produced at cycle 2,
- c_{i+1} is consumed at cycle 8,
- and c_i is produced at cycle 10.

On Fig. 2.c, we represent the execution of the n iterations on the ideal processor. We can see that one iteration starts every two cycles, and that two successive iteration executions overlap by 8 cycles. We deduce that the latency of the whole loop of algorithm CompHorner is $2n+8$ cycles. Since the whole loop requires $22n$ floating point instructions, the ideal IPC for the loop of algorithm CompHorner is

$$\text{IPC}_{\text{CompHorner}} = \frac{22n}{2n+8} \approx 11 \quad \text{instructions per cycle.}$$

D. Ideal IPC of DDHorner

On Fig. 3, we perform the same analysis to determine the total latency of DDHorner execution. We represent the data-flow graph for iteration i of algorithm DDHorner on Fig. 3.a. From the analysis of this data-flow graph, we represent on Fig. 3.b the execution of iteration i:

- sh_{i+1} is consumed at the first cycle of the iteration,
- sh_i is produced at cycle 17,
- sl_{i+1} is consumed at cycle 3,
- and sl_i is produced at cycle 19.

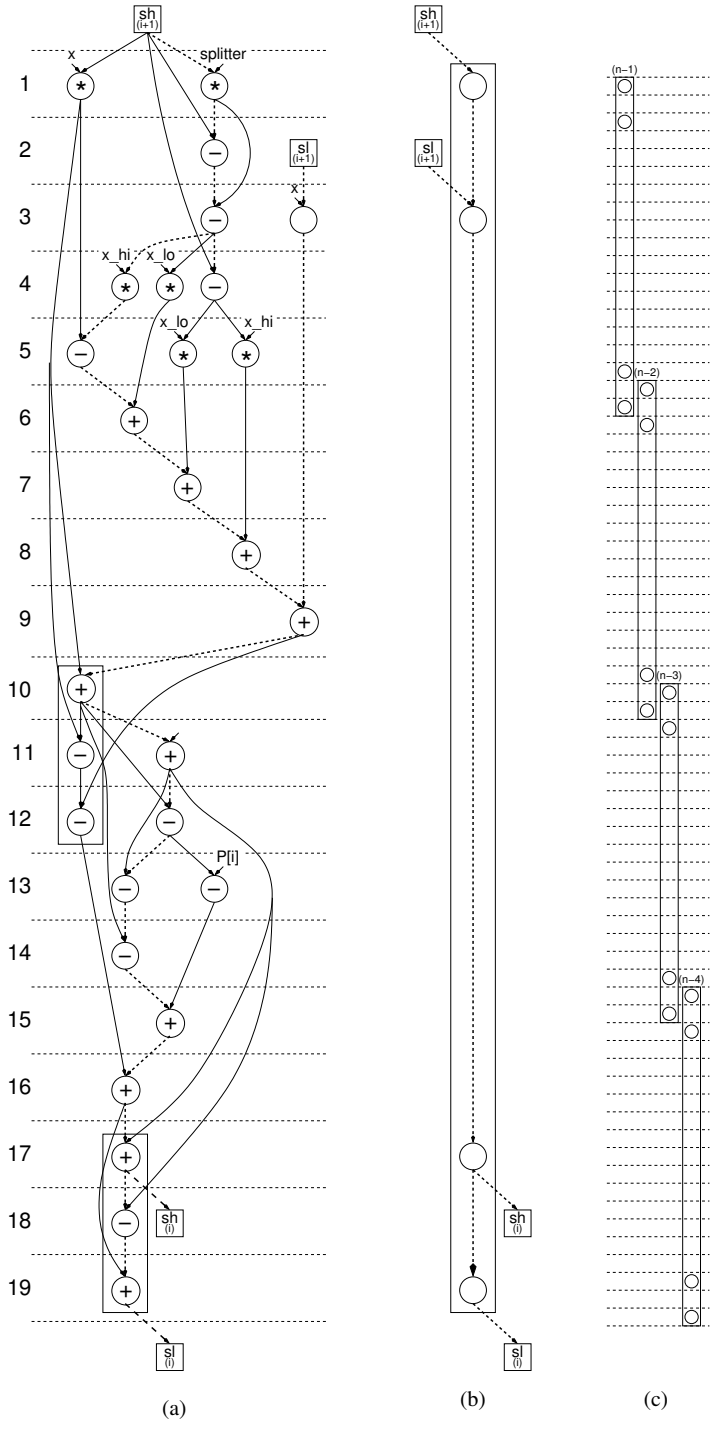


Fig. 3. Data-flow graphs for algorithm DDHorner.

From these remarks, we represent on Fig. 3.c the execution of n iterations on the ideal processor. One iteration starts every 17 cycles, and two successive iteration executions only overlap by 2 cycles. Therefore the latency of the whole loop of algorithm DDHorner is $17n + 2$ clock cycles on the ideal processor. Since the loop execution requires $28n$ floating point instructions, the IPC of DDHorner running on the ideal processor is

$$\mathrm{IPC}_{\mathsf{DDHorner}} = \frac{28n}{17n+2} \approx 1.65 \quad \text{instructions per cycle.}$$

E. Analysis

The ideal IPC of CompHorner is therefore much greater than the one of DDHorner:

$$\mathrm{IPC}_{\mathsf{CompHorner}} \approx 6.66 \times \mathrm{IPC}_{\mathsf{DDHorner}}.$$

Clearly this means that more ILP is available in CompHorner than in DDHorner. We stress that this theoretical analysis cannot explain in a quantitative manner the actual ratios reported in Table III. Indeed measured ratios are from real processors with limited resources while ideal IPC is computed

assuming a processor which exploits all the ILP available in the algorithms. However this certainly explains in a qualitative manner the better efficiency of the compensated algorithm on modern processors designed for exploiting ILP.

The consequence of the renormalization steps needed for double-double computations also appears clearly if we compare Fig. 3 to Fig. 2. They act has "bottlenecks" during the execution of DDHorner. The three floating point operations involved in every renormalization steps are represented in boxes on Fig. 3. If we compare one iteration of the loop of DDHorner to one iteration of CompHorner, it appears that:

- the latency of every iteration of DDHorner is larger because of the first renormalization step —cycles 10 to 12 on Fig. 3.(a),
- due to the second renormalization step —cycles 17 to 19 on Fig. 3.(a)— the overlap between two consecutive iterations of DDHorner is smaller.

The fact that CompHorner avoids any renormalization step is therefore the reason why it exhibits more ILP.

V. Conclusion

Compensated Horner algorithm yields more accurate polynomial evaluation than the classic Horner iteration. Its accuracy is similar to a Horner iteration performed in a doubled working precision. Compensated Horner evaluation is also very efficient, since it runs at least twice as fast as the double-double Horner counterpart still providing the same output accuracy.

We summarize our analysis as follows. Avoiding the renormalization steps needed for double-double computations, the compensated Horner algorithm presents more ILP than its counterpart using double-double arithmetic. In our point of view, this gives a qualitative explanation of its practical efficiency.

The same conclusion certainly holds for other compensated algorithms, that also avoids the renormalization steps. This is the case for:

- the improvements of the compensated Horner algorithm when a FMA is available [9],
- compensated triangular system solver presented in [18],
- compensated summation and dot product in [7].

Let us also emphasis that ILP analysis, as described in this paper, may also be very useful to explain and compare the efficiency of many other numerical algorithms.

The error-free transformations TwoSum and TwoProd are the keys to improve the precision of floating point computation, either with double-double arithmetic or compensated algorithms. Solutions to facilitate their portable implementation have been discussed during the current IEEE-754 revision work [19]. Let us cite the "tail operations" or the ADD3 operator —for $a, b, c \in \mathbb{F}$, $\mathsf{ADD3} = \mathrm{fl}(a + b + c)$. Unfortunately these new operators are not anymore in the draft. Nevertheless the current revision draft proposes the standardization of the FMA and of operations minNumMag and maxNumMag —for $a, b \in \mathbb{F}$, if $|a| \leq |b|$ then $\mathsf{minNumMag}(a, b) = a$, or b otherwise. These additional features will be useful to implement more efficiently the EFT for the multiplication and the addition within the TwoProd with FMA and the FastTwoSum algorithms.

Acknowledgment

The authors thank B. Goossens and D. Parello for stimulating discussions about performance analysis.

Appendix
Complete Experiments Results

Environment (I)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	263	578	1485	1.0	2.2	5.6
15	331	788	2160	1.0	2.4	6.5
20	428	998	2828	1.0	2.3	6.6
25	473	1208	3495	1.0	2.6	7.4
30	570	1418	4163	1.0	2.5	7.3
35	638	1627	4838	1.0	2.6	7.6
40	683	1868	5528	1.0	2.7	8.1
45	780	2070	6203	1.0	2.7	8.0
50	818	2287	6870	1.0	2.8	8.4
55	923	2490	7537	1.0	2.7	8.2
60	983	2708	8213	1.0	2.8	8.4
65	1028	2910	8880	1.0	2.8	8.6
70	1133	3128	9547	1.0	2.8	8.4
75	1200	3330	10223	1.0	2.8	8.5
80	1268	3540	10890	1.0	2.8	8.6
85	1335	3750	11558	1.0	2.8	8.7
90	1380	3961	12233	1.0	2.9	8.9
95	1485	4170	12900	1.0	2.8	8.7
100	1545	4388	13568	1.0	2.8	8.8
105	1590	4590	14243	1.0	2.9	9.0
110	1688	4808	14910	1.0	2.8	8.8
115	1763	5010	15578	1.0	2.8	8.8
120	1830	5220	16253	1.0	2.9	8.9
125	1876	5429	16920	1.0	2.9	9.0
130	1934	5648	17588	1.0	2.9	9.1
135	2040	5850	18263	1.0	2.9	9.0
140	2086	6060	18930	1.0	2.9	9.1
145	2152	6270	19605	1.0	2.9	9.1
150	2250	6480	20273	1.0	2.9	9.0
155	2318	6690	20939	1.0	2.9	9.0
160	2393	6908	21608	1.0	2.9	9.0
165	2491	7110	22283	1.0	2.9	8.9
170	2535	7321	22950	1.0	2.9	9.1
175	2640	7530	23618	1.0	2.9	8.9
180	2693	7741	24292	1.0	2.9	9.0
185	2745	7950	24960	1.0	2.9	9.1
190	2828	8167	25628	1.0	2.9	9.1
195	2910	8370	26303	1.0	2.9	9.0
200	2985	8588	26970	1.0	2.9	9.0
			average overhead		2.8	8.5

Environment (II)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	143	480	1357	1.0	3.4	9.5
15	195	683	2033	1.0	3.5	10.4
20	367	901	2782	1.0	2.5	7.6
25	435	1126	3458	1.0	2.6	7.9
30	518	1298	4140	1.0	2.5	8.0
35	592	1484	4815	1.0	2.5	8.1
40	652	1680	5460	1.0	2.6	8.4
45	735	1859	6150	1.0	2.5	8.4
50	796	2063	6848	1.0	2.6	8.6
55	870	2243	7515	1.0	2.6	8.6
60	929	2445	8144	1.0	2.6	8.8
65	1005	2611	8820	1.0	2.6	8.8
70	1080	2828	9502	1.0	2.6	8.8
75	1140	3007	10163	1.0	2.6	8.9
80	1215	3203	10845	1.0	2.6	8.9
85	1283	3391	11497	1.0	2.6	9.0
90	1358	3578	12166	1.0	2.6	9.0
95	1418	3780	12856	1.0	2.7	9.1
100	1492	3960	13516	1.0	2.7	9.1
105	1561	4148	14206	1.0	2.7	9.1
110	1635	4343	14888	1.0	2.7	9.1
115	1702	4530	15518	1.0	2.7	9.1
120	1778	4717	16185	1.0	2.7	9.1
125	1845	4904	16875	1.0	2.7	9.1
130	1920	5100	17543	1.0	2.7	9.1
135	1980	5288	18202	1.0	2.7	9.2
140	2055	5460	18877	1.0	2.7	9.2
145	2122	5663	19561	1.0	2.7	9.2
150	2198	5850	20221	1.0	2.7	9.2
155	2265	6038	20888	1.0	2.7	9.2
160	2340	6248	21548	1.0	2.7	9.2
165	2400	6435	22222	1.0	2.7	9.3
170	2476	6607	22891	1.0	2.7	9.2
175	2543	6802	23550	1.0	2.7	9.3
180	2610	6989	24225	1.0	2.7	9.3
185	2686	7178	24915	1.0	2.7	9.3
190	2760	7373	25576	1.0	2.7	9.3
195	2828	7568	26243	1.0	2.7	9.3
200	2896	7755	26918	1.0	2.7	9.3
			average overhead		2.8	8.5

Environment (III)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	255	578	1373	1.0	2.3	5.4
15	323	780	1980	1.0	2.4	6.1
20	383	975	2588	1.0	2.5	6.8
25	428	1170	3203	1.0	2.7	7.5
30	503	1373	3810	1.0	2.7	7.6
35	563	1568	4418	1.0	2.8	7.8
40	622	1808	5033	1.0	2.9	8.1
45	683	1988	5678	1.0	2.9	8.3
50	735	2183	6285	1.0	3.0	8.6
55	803	2378	6893	1.0	3.0	8.6
60	863	2588	7508	1.0	3.0	8.7
65	908	2776	8115	1.0	3.1	8.9
70	983	2978	8723	1.0	3.0	8.9
75	1034	3173	9338	1.0	3.1	9.0
80	1103	3375	9945	1.0	3.1	9.0
85	1163	3570	10560	1.0	3.1	9.1
90	1215	3765	11167	1.0	3.1	9.2
95	1283	3960	11775	1.0	3.1	9.2
100	1343	4163	12383	1.0	3.1	9.2
105	1388	4358	12998	1.0	3.1	9.4
110	1463	4560	13605	1.0	3.1	9.3
115	1523	4748	14220	1.0	3.1	9.3
120	1583	4950	14828	1.0	3.1	9.4
125	1643	5153	15435	1.0	3.1	9.4
130	1695	5348	16043	1.0	3.2	9.5
135	1763	5543	16658	1.0	3.1	9.4
140	1823	5745	17265	1.0	3.2	9.5
145	1890	5939	17880	1.0	3.1	9.5
150	1943	6134	18488	1.0	3.2	9.5
155	2003	6329	19095	1.0	3.2	9.5
160	2078	6533	19703	1.0	3.1	9.5
165	2138	6728	20318	1.0	3.1	9.5
170	2190	6923	20925	1.0	3.2	9.6
175	2258	7118	21540	1.0	3.2	9.5
180	2318	7327	22148	1.0	3.2	9.6
185	2370	7515	22755	1.0	3.2	9.6
190	2438	7718	23363	1.0	3.2	9.6
195	2498	7913	23978	1.0	3.2	9.6
200	2565	8115	24585	1.0	3.2	9.6
			average overhead		2.8	8.5

Environment (V)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	138	347	832	1.0	2.5	6.0
15	181	479	1216	1.0	2.6	6.7
20	220	635	1599	1.0	2.9	7.3
25	262	767	1996	1.0	2.9	7.6
30	300	901	2380	1.0	3.0	7.9
35	338	1033	2764	1.0	3.1	8.2
40	380	1180	3147	1.0	3.1	8.3
45	420	1312	3531	1.0	3.1	8.4
50	459	1446	3915	1.0	3.2	8.5
55	499	1578	4299	1.0	3.2	8.6
60	542	1725	4682	1.0	3.2	8.6
65	592	1857	5066	1.0	3.1	8.6
70	619	1991	5450	1.0	3.2	8.8
75	672	2123	5834	1.0	3.2	8.7
80	699	2279	6217	1.0	3.3	8.9
85	749	2402	6601	1.0	3.2	8.8
90	792	2536	6985	1.0	3.2	8.8
95	835	2668	7369	1.0	3.2	8.8
100	872	2815	7752	1.0	3.2	8.9
105	913	2947	8136	1.0	3.2	8.9
110	955	3081	8520	1.0	3.2	8.9
115	992	3213	8904	1.0	3.2	9.0
120	1035	3360	9287	1.0	3.2	9.0
125	1075	3492	9671	1.0	3.2	9.0
130	1112	3626	10055	1.0	3.3	9.0
135	1152	3758	10439	1.0	3.3	9.1
140	1193	3905	10822	1.0	3.3	9.1
145	1235	4037	11206	1.0	3.3	9.1
150	1272	4171	11590	1.0	3.3	9.1
155	1313	4303	11974	1.0	3.3	9.1
160	1352	4450	12357	1.0	3.3	9.1
165	1392	4582	12741	1.0	3.3	9.2
170	1432	4716	13125	1.0	3.3	9.2
175	1473	4848	13509	1.0	3.3	9.2
180	1515	4995	13892	1.0	3.3	9.2
185	1552	5127	14276	1.0	3.3	9.2
190	1592	5261	14660	1.0	3.3	9.2
195	1635	5393	15044	1.0	3.3	9.2
200	1672	5540	15427	1.0	3.3	9.2
			average overhead		2.8	8.5

Environment (IV)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	135	480	1298	1.0	3.6	9.6
15	188	690	1928	1.0	3.7	10.3
20	338	945	2670	1.0	2.8	7.9
25	406	1147	3292	1.0	2.8	8.1
30	465	1380	3953	1.0	3.0	8.5
35	510	1560	4568	1.0	3.1	9.0
40	570	1755	5206	1.0	3.1	9.1
45	638	1973	5835	1.0	3.1	9.1
50	697	2175	6465	1.0	3.1	9.3
55	758	2378	7110	1.0	3.1	9.4
60	826	2588	7740	1.0	3.1	9.4
65	870	2797	8392	1.0	3.2	9.6
70	938	3000	9015	1.0	3.2	9.6
75	997	3203	9645	1.0	3.2	9.7
80	1088	3405	10283	1.0	3.1	9.5
85	1118	3615	10920	1.0	3.2	9.8
90	1170	3818	11549	1.0	3.3	9.9
95	1238	4028	12180	1.0	3.3	9.8
100	1298	4222	12818	1.0	3.3	9.9
105	1380	4441	13455	1.0	3.2	9.8
110	1417	4634	14093	1.0	3.3	9.9
115	1499	4845	14729	1.0	3.2	9.8
120	1537	5040	15361	1.0	3.3	10.0
125	1590	5258	15990	1.0	3.3	10.1
130	1650	5453	16634	1.0	3.3	10.1
135	1718	5663	17265	1.0	3.3	10.0
140	1770	5865	17903	1.0	3.3	10.1
145	1838	6068	18532	1.0	3.3	10.1
150	1898	6270	19170	1.0	3.3	10.1
155	1949	6488	19808	1.0	3.3	10.2
160	2018	6682	20445	1.0	3.3	10.1
165	2070	6893	21068	1.0	3.3	10.2
170	2137	7103	21713	1.0	3.3	10.2
175	2183	7297	22342	1.0	3.3	10.2
180	2258	7515	22979	1.0	3.3	10.2
185	2318	7702	23626	1.0	3.3	10.2
190	2370	7919	24254	1.0	3.3	10.2
195	2430	8115	24893	1.0	3.3	10.2
200	2498	8317	25538	1.0	3.3	10.2
			average overhead		2.8	8.5

Environment (VI)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	167	321	583	1.0	1.9	3.5
15	200	425	816	1.0	2.1	4.1
20	232	528	1053	1.0	2.3	4.5
25	245	581	1286	1.0	2.4	5.2
30	279	686	1523	1.0	2.5	5.5
35	303	790	1756	1.0	2.6	5.8
40	335	893	1993	1.0	2.7	5.9
45	359	946	2226	1.0	2.6	6.2
50	382	1051	2463	1.0	2.8	6.4
55	415	1155	2696	1.0	2.8	6.5
60	439	1258	2933	1.0	2.9	6.7
65	460	1311	3166	1.0	2.9	6.9
70	494	1416	3403	1.0	2.9	6.9
75	518	1520	3636	1.0	2.9	7.0
80	550	1623	3873	1.0	3.0	7.0
85	574	1676	4106	1.0	2.9	7.2
90	597	1781	4343	1.0	3.0	7.3
95	630	1885	4576	1.0	3.0	7.3
100	654	1988	4813	1.0	3.0	7.4
105	675	2041	5046	1.0	3.0	7.5
110	709	2146	5283	1.0	3.0	7.5
115	733	2250	5516	1.0	3.1	7.5
120	765	2353	5753	1.0	3.1	7.5
125	789	2406	5986	1.0	3.0	7.6
130	812	2511	6223	1.0	3.1	7.7
135	845	2615	6456	1.0	3.1	7.6
140	869	2718	6693	1.0	3.1	7.7
145	890	2771	6926	1.0	3.1	7.8
150	924	2876	7163	1.0	3.1	7.8
155	948	2980	7396	1.0	3.1	7.8
160	980	3083	7633	1.0	3.1	7.8
165	1004	3136	7866	1.0	3.1	7.8
170	1027	3241	8103	1.0	3.2	7.9
175	1060	3345	8336	1.0	3.2	7.9
180	1084	3448	8573	1.0	3.2	7.9
185	1105	3501	8806	1.0	3.2	8.0
190	1139	3606	9043	1.0	3.2	7.9
195	1163	3710	9276	1.0	3.2	8.0
200	1195	3813	9513	1.0	3.2	8.0
			average overhead		2.8	8.5

Environment (VII)

	Execution time in clock cycles			Normalized execution time		
deg	H	CH	DDH	H	CH	DDH
10	386	448	793	1.0	1.2	2.1
15	406	488	1013	1.0	1.2	2.5
20	434	528	1233	1.0	1.2	2.8
25	457	568	1453	1.0	1.2	3.2
30	474	608	1673	1.0	1.3	3.5
35	506	648	1893	1.0	1.3	3.7
40	514	688	2113	1.0	1.3	4.1
45	537	728	2333	1.0	1.4	4.3
50	554	768	2553	1.0	1.4	4.6
55	577	808	2773	1.0	1.4	4.8
60	594	848	2993	1.0	1.4	5.0
65	617	888	3213	1.0	1.4	5.2
70	634	928	3433	1.0	1.5	5.4
75	666	968	3653	1.0	1.5	5.5
80	674	1008	3873	1.0	1.5	5.7
85	697	1048	4093	1.0	1.5	5.9
90	714	1089	4315	1.0	1.5	6.0
95	737	1128	4533	1.0	1.5	6.2
100	754	1168	4753	1.0	1.5	6.3
105	777	1208	4973	1.0	1.6	6.4
110	794	1248	5193	1.0	1.6	6.5
115	826	1288	5413	1.0	1.6	6.6
120	834	1328	5633	1.0	1.6	6.8
125	857	1368	5853	1.0	1.6	6.8
130	874	1408	6073	1.0	1.6	6.9
135	897	1448	6293	1.0	1.6	7.0
140	914	1488	6513	1.0	1.6	7.1
145	937	1528	6733	1.0	1.6	7.2
150	954	1568	6953	1.0	1.6	7.3
155	986	1609	7175	1.0	1.6	7.3
160	994	1648	7393	1.0	1.7	7.4
165	1017	1688	7613	1.0	1.7	7.5
170	1034	1728	7833	1.0	1.7	7.6
175	1057	1768	8053	1.0	1.7	7.6
180	1074	1808	8273	1.0	1.7	7.7
185	1097	1848	8493	1.0	1.7	7.7
190	1114	1888	8713	1.0	1.7	7.8
195	1146	1928	8933	1.0	1.7	7.8
200	1154	1968	9153	1.0	1.7	7.9
			average overhead		2.8	8.5

REFERENCES

[1] J.-M. Muller, *Elementary functions: algorithms and implementation*, 2nd ed. Birkhäuser, 2006.

[2] C. Li, S. Pion, and C.-K. Yap, "Recent progress in exact geometric computation," *Journal of Logic and Algebraic Programming*, vol. 64, no. 1, pp. 85–111, 2005.

[3] C. M. Hoffmann, G. Park, J.-R. Simard, and N. F. Stewart, "Residual iteration and accurate polynomial evaluation for shape-interrogation applications," in *SM '04: Proceedings of the ninth ACM symposium on Solid modeling and applications*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 9–14.

[4] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proceedings of IEEE ARITH-10*, 1991, pp. 132–144.

[5] 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 Trans. Math. Software*, vol. 28, no. 2, pp. 152–205, 2002.

[6] S. Graillat, P. Langlois, and N. Louvet, "Compensated Horner scheme," Univ. of Perpignan, France, Tech. Rep., 2005. [Online]. Available: http://webdali.univ-perp.fr/RR/

[7] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, 2005.

[8] P. Langlois and N. Louvet, "How to ensure a faithful polynomial evaluation with the compensated Horner algorithm?" in *18th IEEE International Symposium on Computer Arithmetic*, P. Kornerup and J.-M. Muller, Eds., no. ISBN 0-7695-2854-6. IEEE Computer Society, Jun. 2007, pp. 141–149.

[9] ——, "Operator dependant compensated algorithms," in *Proceedings of the 12th GAMM - IMACS - SCAN, Duisburg, Germany*, 2007.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 2nd ed. Morgan Kaufmann, 2003.

[11] *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985*, 1985.

[12] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.

[13] J. W. Demmel, *Applied Numerical Linear Algebra*. SIAM, 1997.

[14] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1998.

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

[16] P. Markstein, *IA-64 and elementary functions: speed and precision*. Prentice-Hall, 2000.

[17] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating point arithmetic," in *Proceedings of the IEEE ARITH-15*, 2001, pp. 155–162.

[18] P. Langlois and N. Louvet, "Solving triangular systems more accurately and efficiently," in *Proceedings of the 17th IMACS World Congress, Paris*, Jul. 2005, (Also available as DALI Research Report RR2005-02).

[19] "Draft standard for floating-point arithmetic P754," Oct. 2006, available at http://www.validlab.com/754R/.