A parallel compensated Horner scheme for SIMD architecture

Stef Graillat
Sorbonne Université, CNRS, LIP6
Paris, France
stef.graillat@lip6.fr

Youness Ibrahimy
youness.ibra@gmail.com

Clothilde Jeangoudoux
IDEMIA
Paris, France
clothilde.jeangoudoux@gmail.com

Christoph Lauter
Department of Computer Science
University of Texas at El Paso
El Paso, Texas, USA
christoph.lauter@christoph-lauter.org

***Abstract*—A parallel algorithm for accurate polynomial evaluation is proposed for SIMD architectures. This is a parallelized version of the compensated Horner scheme using error-free transformations. The proposed parallel algorithm in this paper is fast and is designed to achieve a result as if computed in twice the working precision and then rounded to the working precision. Numerical results are presented showing the performance of this new parallel algorithm.**

***Index Terms*—polynomial evaluation, compensated algorithms, error-free transformations, Horner scheme, rounding errors, parallel algorithms, SIMD, AVX**

I. Introduction

Polynomials appear in almost all areas in scientific computing. Generally the problem involved is to solve equations or systems of polynomial equations often in many variables. The wide range of use of polynomial systems needs to have fast and reliable methods to solve them. Basically, there are two general approaches: symbolic and numeric. The symbolic approach is based either on the theory of Gröbner bases or on the theory of resultants. For the numeric approach, it is the use of iterative methods like Newton's method or homotopy continuation methods. These iterative methods need to evaluate polynomials and their derivatives.

One of the three main processes associated with polynomials is evaluation; the two other ones being interpolation and root finding. The classic Horner scheme is the optimal algorithm with respect to algebraic complexity for evaluating a univariate polynomial p with given coefficients in the monomial basis. Higham [8, chap. 5] devotes an entire chapter to polynomials and more especially to polynomial evaluation.

SIMD stands for "Single Instruction, Multiple Data". It is a type of computer architecture that allows multiple operations to be performed simultaneously on a whole vector of data. In a SIMD architecture, a single instruction is executed on multiple data items at once, rather than on just one item at a time. There are different SIMD instruction sets, including Advanced Vector Extensions (AVX) and Streaming SIMD Extensions (SSE), which are supported by modern CPUs and GPUs. AVX2 or AVX512 are more recent SIMD instruction sets. They support operations on 256-bit or 512-bit vectors and includes more advanced instructions for floating-point arithmetic, such as FMA (fused multiply-add), which allows for a single instruction to perform both multiplication and addition on a vector of data. Both SSE and AVX instruction sets are used for a variety of applications, including multimedia processing, scientific simulations, and cryptography. They can significantly improve performance in these applications by allowing the CPU to perform multiple computations in parallel.

The compensated Horner scheme is a fast and accurate algorithm to evaluate polynomials. By accurate it means that it achieves a result as accurate as if computed in twice the working precision and then rounded to the working precision. The aim of this paper is to derive a parallel version of the algorithm that is suited for SIMD architectures. To our knowledge, literature on polynomials evaluation with SIMD contains nothing but examples with very small degrees, such as used for evaluation of elementary functions, where argument reduction is possible. In the article, we want to evaluate accurately and fastly polynomials with high degree. Such polynomials of high degrees which must be evaluated with high performance and high accuracy are commonly used for example as proxies for mathematical special functions, which are *not* elementary, and given for instance by differential equations [12]. Obtaining a faithful rounding is often necessary, which the compensated Horner scheme can guarantee, without requiring the use of high-precision arithmetic, such as IEEE754 binary128.

The paper is organized as follows. We introduce assump-

This work was partly supported by the NuSCAP (ANR-20-CE48-0014) project of the French National Agency for Research (ANR)

tions on floating-point arithmetic and notations for error analysis as well as review algorithms the error-free transformations in Section II. Section III is devoted to the presentation of the compensated Horner scheme. In Section IV we present some algorithms for accurately computing summation and integer exponentiation. The new parallel compensated Horner scheme is presented in Section V. Finally Section VI is devoted to numerical experiments.

II. Floating-point arithmetic and error-free transformations

Throughout the paper, we assume to work with binary floating-point arithmetic adhering to the IEEE 754 floating-point standard [9], [13]. We assume that no overflow nor underflow occurs. We suppose that the prevailing rounding direction (rounding mode) is round-to-nearest-ties-to-even. The set of floating-point numbers is denoted by $\mathbb{F}$, the relative rounding error by $\mathbf{u}$. For the IEEE 754 binary64 format (double precision), we have $\mathbf{u} = 2^{-53}$ and for the binary32 format (single precision), we have $\mathbf{u} = 2^{-24}$.

We denote by $\mathrm{fl}(\cdot)$ the result of a floating-point computation, where all operations inside parentheses are done in floating-point working precision. In the absence of underflow and overflow, floating-point operations in IEEE 754 satisfy [8]

$$\mathrm{fl}(a \circ b) = (a \circ b)(1 + \varepsilon_1) = (a \circ b)/(1 + \varepsilon_2) \text{ for } \circ = \{+, -, \cdot, /\} \text{ and } |\varepsilon_\nu| \le \mathbf{u}.$$

This implies that

$$\begin{aligned} |a \circ b - \mathrm{fl}(a \circ b)| \le \mathbf{u}|a \circ b| \text{ and} \\ |a \circ b - \mathrm{fl}(a \circ b)| \le \mathbf{u}|\,\mathrm{fl}(a \circ b)| \text{ for } \circ = \{+, -, \cdot, /\}. \end{aligned} \quad (1)$$

We use standard notation for error estimations. The quantities γ_n are defined as usual [8] by

$$\gamma_n := \frac{n\mathbf{u}}{1 - n\mathbf{u}} \quad \text{for } n \in \mathbb{N},$$

where we implicitly assume that $n\mathbf{u} \le 1$. A direct calculation shows that $(1+\gamma_i)(1+\gamma_j) \le 1 + \gamma_{i+j}$ and $(1+u)\gamma_i \le \gamma_{i+1}$.

As a matter of course, for $a, b \in \mathbb{F}$, we have $a \circ b \in \mathbb{R}$ and $a \circledcirc b := \mathrm{fl}(a \circ b) \in \mathbb{F}$ but in general we do not have $a \circ b \in \mathbb{F}$ – a rounding occurs in general. However, it is known that for the basic operations $+, -, \cdot$, this very rounding error of a floating-point operation is itself a floating-point number (see for example [3] for a proof):

$$\begin{array}{lcll} x = a \oplus b & \Rightarrow & a + b = x + y & \text{with } y \in \mathbb{F}, \\ x = a \ominus b & \Rightarrow & a - b = x + y & \text{with } y \in \mathbb{F}, \\ x = a \otimes b & \Rightarrow & a \times b = x + y & \text{with } y \in \mathbb{F}. \end{array} \quad (2)$$

We call these operations transforming a pair of floating-point numbers (a, b) into another pair of floating-point numbers (x, y) *error-free transformations*.

Fortunately, the quantities x and y in (2) can be computed exactly in floating-point arithmetic. For the algorithms, we use Matlab-like notations. For addition, we can use the following algorithm by Knuth [11, Thm B. p.236].

Algorithm II.1 (Knuth [11])**.** Error-free transformation of the sum of two floating-point numbers

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

Another algorithm to compute an error-free transformation is the following algorithm from Dekker [3]. The drawback of this algorithm is that we have $x + y = a + b$ provided that $|a| \ge |b|$.

Algorithm II.2 (Dekker [3])**.** Error-free transformation of the sum of two floating-point numbers.

```
function $[x, y]$ = FastTwoSum$(a, b)$
  $x = a \oplus b$
  $y = (a \ominus x) \oplus b$
```

For the error-free transformation of a product, we use the Fused-Multiply-and-Add (FMA) operator which is now widely available [2], [14]. For $a, b, c \in \mathbb{F}$, the result of $\mathtt{FMA}(a, b, c)$ is the nearest floating-point number of $a \cdot b + c \in \mathbb{R}$. In the absence of underflow and overflow, the FMA satisfies

$$\begin{aligned} \mathtt{FMA}(a, b, c) &= (a \cdot b + c)(1 + \varepsilon_1) \\ &= (a \cdot b + c)/(1 + \varepsilon_2) \quad \text{with } |\varepsilon_\nu| \le \mathbf{u}. \end{aligned}$$

Algorithm II.3 (Ogita, Rump and Oishi [15])**.** Error-free transformation of the product of two floating-point numbers using an FMA.

```
function $[x, y]$ = TwoProduct$(a, b)$
  $x = a \otimes b$
  $y = \mathtt{FMA}(a, b, -x)$
```

The following theorem summarizes the properties of algorithms TwoSum and TwoProduct.

Theorem II.1 (Ogita, Rump and Oishi [15])**.** *Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \mathtt{TwoSum}(a, b)$ (Algorithm II.1). Then,*

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \le \mathbf{u}|x|, \quad |y| \le \mathbf{u}|a + b|. \quad (3)$$

The algorithm TwoSum *requires* 6 *flops.*
Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \mathtt{TwoProduct}(a, b)$ (Algorithm II.3). Then,

$$a \cdot b = x + y, \quad x = a \otimes b, \quad |y| \le \mathbf{u}|x|, \quad |y| \le \mathbf{u}|a \cdot b|. \quad (4)$$

The FMA *based algorithm* TwoProduct *requires* 2 *flops.*

Algorithm II.4. Computation of the sum of floating-point numbers

```
function res = Sum$(p)$
  $s_1 = p_1$
  for $i = 2 : n$
    $s_i = s_{i-1} \oplus p_i$
  res $= s_n$
```

Lemma II.2. *Suppose Algorithm* `Sum` *is applied to floating-point number* $p_i \in \mathbb{F}$, $1 \leq i \leq n$. *Let* $s := \sum p_i$, $S := \sum |p_i|$ *and* $n\mathbf{u} < 1$. *Then, one has*

$$|\mathtt{res} - s| \leq \gamma_{n-1} S.$$

III. Compensated Horner scheme

We now want to accurately compute the evaluation of a polynomial at a given point. We present hereafter a compensated algorithm for Horner scheme. One can find a more detailed description of the algorithm in [6], [7]. We first recall the classic algorithm for Horner scheme and give an error bound. We then present the compensated Horner scheme together with an error bound.

The classical method for evaluating a polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

is the Horner scheme which consists in the following algorithm.

Algorithm III.1. Polynomial evaluation with Horner's scheme

```
function $\mathtt{res} = \mathtt{Horner}(p, x)$
$s_n = a_n$
for $i = n-1 : -1 : 0$
    $s_i = s_{i+1} \otimes x \oplus a_i$
end
$\mathtt{res} = s_0$
```

A forward error bound for the result of Algorithm III.1 is (see [8, p.95]):

$$|p(x) - \mathtt{res}| \leq \gamma_{2n} \sum_{i=0}^{n} |a_i||x|^i = \gamma_{2n} \widetilde{p}(|x|)$$

where $\widetilde{p}(x) = \sum_{i=0}^{n} |a_i| x^i$. It is very interesting to express and interpret this result in terms of the *condition number* of the polynomial evaluation defined by

$$\mathrm{cond}(p, x) = \frac{\sum_{i=0}^{n} |a_i||x|^i}{|p(x)|} = \frac{\widetilde{p}(|x|)}{|p(x)|}. \quad (5)$$

The condition number measures the sensitivity of the evaluation with respect to small perturbations of the coefficients of the polynomial. We will refer to the $\mathrm{cond}(p, x)$ quantity as the *condition number* in the text and we will use the $\mathrm{cond}(p, x)$ quantity directly in formulas. Thus we have

$$\frac{|p(x) - \mathtt{res}|}{|p(x)|} \leq \gamma_{2n} \, \mathrm{cond}(p, x).$$

If an FMA instruction is available, then the statement $s_i = s_{i+1} \otimes x \oplus a_i$ in Algorithm III.1 can be re-written $s_i = \mathtt{FMA}(s_{i+1}, x, a_i)$ which slightly improves the error bound. Using an FMA this way, the computed result now satisfies

$$\frac{|p(x) - \mathtt{res}|}{|p(x)|} \leq \gamma_n \, \mathrm{cond}(p, x).$$

One can modify the Horner scheme to compute the rounding error at each elementary operation that are a sum and a product. This is done in Algorithm III.2 (see [6], [7]).

Algorithm III.2 ([6], [7])**.** Polynomial evaluation with a compensated Horner's scheme

```
function $\mathtt{res} = \mathtt{CompHorner}(p, x)$
$s_n = a_n$
$r_n = 0$
for $i = n-1 : -1 : 0$
    $[p_i, \pi_i] = \mathtt{TwoProduct}(s_{i+1}, x)$
    $[s_i, \sigma_i] = \mathtt{TwoSum}(p_i, a_i)$
    $r_i = r_{i+1} \otimes x \oplus (\pi_i \oplus \sigma_i)$
end
$\mathtt{res} = s_0 \oplus r_0$
```

If one denotes by p_π and p_σ the two following polynomials

$$p_\pi = \sum_{i=0}^{n-1} \pi_i x^i, \qquad p_\sigma = \sum_{i=0}^{n-1} \sigma_i x^i,$$

then one can show, thanks to error-free transformations that

$$p(x) = s_0 + p_\pi(x) + p_\sigma(x).$$

If one looks at the previous algorithm closely, it is then clear that $s_0 = \mathtt{Horner}(p, x)$. As a consequence, one can derive a new error-free transformation for polynomial evaluation

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

The compensated Horner scheme first computes $p_\pi(x) + p_\sigma(x)$ which correspond to the rounding errors and then add the obtained value to the result of the classic Horner scheme $\mathtt{Horner}(p, x)$.

We will show that the results computed by Algorithm III.2 admit significantly better error bounds than those computed with the classical Horner scheme. We argue that Algorithm III.2 provides results as if they were computed using twice the working precision. This is summed up in the following theorem.

Theorem III.1 ([6], [7])**.** *Consider a polynomial* p *of degree* n *with floating-point coefficients, and a floating-point value* x. *The forward error in the compensated Horner algorithm is such that*

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

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 (5) for polynomial evaluation gives

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

In other words, the bound for the relative error of the computed result is essentially γ_{2n}^2 times the condition number of the polynomial evaluation, plus the unavoidable term $\mathbf{u}$ for rounding the result to the working precision. In particular, if

$\mathrm{cond}(p,x) < \gamma_{2n}^{-1}$, then the relative accuracy of the result is bounded by a constant of the order of $\mathbf{u}$. 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 $\gamma_{2n}^{-1} \approx (2n\mathbf{u})^{-1}$. Besides that, (7) tells us that the computed result is as accurate as if computed by the classic Horner algorithm with twice the working precision, and then rounded to the working precision.

IV. Accurate exponentiation and summation

In this section, we present an accurate algorithm to compute the sum of floating-point numbers [15] as well as another accurate algorithm to compute the integer exponentiation of a floating-point number [4].

Algorithm IV.1 ([15]). Compensated summation algorithm

```
function res = CompSum($p$)
  $\pi_1 = p_1$ ; $\sigma_1 = 0$;
  for $i = 2 : n$
    $[\pi_i, q_i] = \mathtt{TwoSum}(\pi_{i-1}, p_i)$
    $\sigma_i = \sigma_{i-1} \oplus q_i$
  res $= \pi_n \oplus \sigma_n$
```

The following proposition shows that the computed result by CompSum(p) is as accurate as if computed with twice the working precision and then rounded to the current precision.

Proposition IV.1 ([15]). *Suppose Algorithm* CompSum *is applied to floating-point number $p_i \in \mathbb{F}$, $1 \le i \le n$. Let $s := \sum p_i$, $S := \sum |p_i|$ and $n\mathbf{u} < 1$. Then, one has*

$$|\mathtt{res} - s| \le \mathbf{u}|s| + \gamma_{n-1}^2 S.$$

Compensated methods are a possible way to improve the accuracy. Another possibility is to increase the working precision. For this purpose, one can use the Bailey's *double-double* [1]: double-double numbers are represented as an unevaluated sum of a leading double and a trailing double. More precisely, a double-double number a is the pair (a_h, a_l) of floating-point numbers with $a = a_h + a_l$ and $|a_l| \le \mathbf{u}|a_h|$.

In the sequel, we present two algorithms to compute the product of two double-double or a double times a double-double. Those algorithms are taken from [10].

Algorithm IV.2 ([10]). Multiplication of double-double number by a double number

```
function $[r_h, r_l]$ = prod_dd_d($a, b_h, b_l$)
  $[t_1, t_2] = \mathtt{TwoProd}(a, b_h)$
  $t_3 = (a \oplus b_l) \oplus t_2$
  $[r_h, r_l] = \mathtt{TwoProd}(t_1, t_3)$
```

Algorithm IV.3 ([10]). Multiplication of two double-double numbers

```
function $[r_h, r_l]$ = prod_dd_dd($a_h, a_l, b_h, b_l$)
  $[t_1, t_2] = \mathtt{TwoProd}(a_h, b_h)$
  $t_3 = ((a_h \otimes b_l) \oplus (a_l \otimes b_h)) \oplus t_2$
  $[r_h, r_l] = \mathtt{TwoProd}(t_1, t_3)$
```

Theorem IV.2 ([10]). *Let be $a_h + a_l$ and $b_h + b_l$ the double-double arguments of Algorithm* prod_dd_dd. *Then the returned values r_h and r_l satisfy*

$$r_h + r_l = ((a_h + a_l) \cdot (b_h + b_l))(1 + \varepsilon)$$

where ε is bounded as follows : $|\varepsilon| \le 7\mathbf{u}^2$. Furthermore, we have $|r_l| \le \mathbf{u}|r_h|$.

We can now use those algorithms to accurately evaluate the integer power of a floating-point number.

Algorithm IV.4 ([4]). Power evaluation with a compensated scheme

```
function res = CompLogPower($x, n$)
% $n = (n_t n_{t-1} \cdots n_1 n_0)_2$
  $[h, l] = [1, 0]$
  for $i = t : -1 : 0$
    $[h, l]$ = prod_dd_dd($h, l, h, l$)
    if $n_i = 1$
       $[h, l]$ = prod_dd_d($x, h, l$)
    end
  end
  res $= [h, l]$
```

Theorem IV.3 ([4]). *The two values h and l returned by Algorithm* CompLogPower *satisfy*

$$h + l = x^n(1 + \varepsilon)$$

with

$$(1 - 7\mathbf{u}^2)^{n-1} \le 1 + \varepsilon \le (1 + 7\mathbf{u}^2)^{n-1}.$$

V. Parallel compensated Horner scheme

This algorithm is inspired from the parallel compensated summation and dot product algorithms of [16]. However, [16] just presents building blocks for summation, handling the multiplication part of a dot product in just a couple of lines. Evaluating a polynomial, which is our task here, is more challenging, as the multiplication steps require multiplying a native precision floating-point number, namely x, by a compensated, "double-double" floating-point number. While [16] is inspriring, we still need to develop techniques for our polynomial evaluation problems on our own.

Let us assume $p(x) = \sum_{i=0}^{n} a_i x^i$ with $n + 1 = K \times M$

$$p(x) = \sum_{l=0}^{K-1} x^{lM} p_l(x) \text{ with } p_l(x) = \sum_{k=0}^{M-1} a_{k+lM} x^k.$$

We first present a parallel version of the classic Horner scheme. The outline of the algorithm is as follow:

1) distribute the "sub-polynomials" p_l to the K SIMD lanes;
2) each SIMD lane l computes x^{lM} and evaluates $p_l(x)$ and stores their product into a vector q;
3) use a recursive summation algorithm to compute the sum of the elements of q.

Algorithm V.1. Parallel Horner scheme

```
function res = PHorner(p, x)
  K = (n + 1)/M
  % begin parallel on K SIMD lanes (id = 0, ..., K − 1)
  y = x^{id·M}
  r = Horner(p_{id}, x)
  q(id) = y ⊗ r
  % end parallel
  res = Sum(q)
```

Theorem V.1. *Let p be a polynomial of degree n with floating-point coefficients, and x be a floating-point value. Then if no underflow occurs, and* $\mathtt{res} = \mathtt{PHorner}(p, x)$*,*

$$\frac{|\mathtt{res} - p(x)|}{|p(x)|} \leq \left[(M(K+1) - 2)\mathbf{u} + \mathcal{O}(\mathbf{u}^2)\right] \mathrm{cond}(p, x).$$

Proof. Let us assume $p(x) = \sum_{i=0}^{n} a_i x^i$ with $n+1 = K \times M$ and so

$$p(x) = \sum_{l=0}^{K-1} x^{lM} p_l(x) \quad \text{with} \quad p_l(x) = \sum_{k=0}^{M-1} a_{k+lM} x^k.$$

We have

$$\begin{aligned} |\mathtt{res} - p(x)| &\leq |\mathtt{res} - \sum_{l=0}^{K-1} q_l| + |\sum_{l=0}^{K-1} q_l - p(x)|, \\ &\leq |\mathtt{res} - \sum_{l=0}^{K-1} q_l| + \sum_{l=0}^{K-1} |q_l - x^{lM} p_l(x)|, \\ &\leq \gamma_{K-1} \sum_{l=0}^{K-1} |q_l| + \sum_{l=0}^{K-1} |q_l - x^{lM} p_l(x)|. \end{aligned} \quad (8)$$

As $q_l = y_l \otimes r_l$, we have $|q_l| \leq (1 + \mathbf{u})|y_l||r_l|$. Moreover, we have

$$|y_l - x^{lM}| \leq \gamma_{lM-1} |x^{lM}| \quad (9)$$

and

$$|r_l - p_l(x)| \leq \gamma_{2(M-1)} \widetilde{p}_l(|x|). \quad (10)$$

As a consequence, mixing the previous inequalities, we obtain

$$\begin{aligned} |q_l| &\leq (1 + u)(1 + \gamma_{lM-1})(1 + \gamma_{2(M-1)})|x^{lM}|\widetilde{p}_l(|x|) \\ &\leq (1 + \gamma_{(2+l)M-2})|x|^{lM} \widetilde{p}_l(|x|). \end{aligned}$$

We also have

$$\begin{aligned} |q_l - x^{lM} p_l(x)| &\leq |q_l - y_l r_l| + |y_l r_l - x^{lM} p_l(x)|, \\ &\leq \mathbf{u}|y_l||r_l| + \\ &\quad |(y_l - x^{lM}) r_l + (r_l - p_l(x)) x^{lM}|, \\ &\leq \mathbf{u}|y_l||r_l| + \\ &\quad |y_l - x^{lM}||r_l| + |r_l - p_l(x)||x|^{lM}. \end{aligned}$$

Using (9) and (10), it follows that

$$\begin{aligned} |q_l - x^{lM} p_l(x)| &\leq \mathbf{u}(1 + \gamma_{(2+l)M-1})|x|^{lM} \widetilde{p}_l(|x|) + \\ \gamma_{lM-1}|x^{lM}|(1 + \gamma_{2(M-1)})&\widetilde{p}_l(|x|) + \gamma_{2(M-1)} \widetilde{p}_l(|x|)|x|^{lM}. \end{aligned}$$

Using this and (8), we can deduce that

$$\begin{aligned} |\mathtt{res} - p(x)| \leq \gamma_{K-1}(1 + \gamma_{(K+1)M-2})&\widetilde{p}(|x|) + \\ [u(1 + \gamma_{(K+1)M-1}) &+ \\ \gamma_{(K-1)M-1}(1 + \gamma_{2(M-1)}) + \gamma_{2(M-1)}&]\widetilde{p}(|x|) \end{aligned} \quad (11)$$

If we keep only the terms in $\mathbf{u}$, we obtain

$$|\mathtt{res} - p(x)| \leq \left[(M(K+1) - 2)\mathbf{u} + \mathcal{O}(\mathbf{u}^2)\right] \widetilde{p}(|x|),$$

which concludes the proof. □

We can now outline a parallel compensated Horner scheme as follow:

1) distribute the "sub-polynomials" p_l to the K SIMD lanes;
2) each lane l computes x^{lM} with `CompLogPower` and evaluates $p_l(x)$ with `CompHorner` and stores their product computed with double-double into a vector q;
3) use a compensated summation algorithm `CompSum` to compute the sum of the elements of q.

Algorithm V.2. Parallel compensated Horner scheme

```
function res = PCompHorner(p, x)
  K = (n + 1)/M
  % begin parallel on K SIMD lanes (id = 0, ..., K − 1)
  [e, f] = CompLogPower(x, id · M)
  [r, c] = CompHorner(p_{id}, x)
  [q(2 · id − 1), q(2 · id)] = prod_dd_dd(r, c, e, f)
  % end parallel
  res = CompSum(q)
```

Theorem V.2. *Let p be a polynomial of degree n with floating-point coefficients, and x be a floating-point value. Then if no underflow occurs, and* $\mathtt{res} = \mathtt{PCompHorner}(p, x)$*,*

$$\begin{aligned} \frac{|\mathtt{res} - p(x)|}{|p(x)|} &\leq \mathbf{u} + \\ [(7 + 4(\frac{n+1-K}{K})^2 + 4n^2 + 1)\mathbf{u}^2 &+ \mathcal{O}(\mathbf{u}^3)] \,\mathrm{cond}(p, x). \end{aligned} \quad (12)$$

Proof. Let us assume $p(x) = \sum_{i=0}^{n} a_i x^i$ with $n+1 = K \times M$ and so

$$p(x) = \sum_{l=0}^{K-1} x^{lM} p_l(x) \quad \text{with} \quad p_l(x) = \sum_{k=0}^{M-1} a_{k+lM} x^k.$$

From the compensated Horner scheme, we deduce that

$$|(r_l + c_l) - p_l(x)| \leq \gamma_{2(M-1)}^2 \widetilde{p}_l(|x|). \quad (13)$$

With the use of `CompLogPower`, we can say that

$$|(e_l + f_l) - x^{lM}| \leq \overline{\gamma}_{lM} \text{ with } \overline{\gamma}_n := \frac{n\mathbf{u}^2}{1 - n\mathbf{u}^2} \quad (14)$$

Moreover using `prod_dd_dd` yields

$$|(q_{2l} + q_{2l+1}) - (r_l + c_l)(e_l + f_l)| \leq 7\mathbf{u}^2 |(r_l + c_l)(e_l + f_l)|.$$

The use of CompSum leads to

$$|\mathtt{res} - \sum_{l=0}^{K-1}(q_{2l}+q_{2l+1})| \leq \mathbf{u}|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1})| + \gamma_{2n}^2\sum_{l=0}^{K-1}(|q_{2l}|+|q_{2l+1}|).$$

As a consequence, we can deduce that

$$\begin{aligned}|\mathtt{res} - p(x)| &\leq |\mathtt{res} - \sum_{l=0}^{K-1}(q_{2l}+q_{2l+1})| + \\ &|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1}) - \sum_{l=0}^{K-1}x^{lM}p_l(x)|, \\ &\leq \mathbf{u}|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1})| + \gamma_{2n}^2\sum_{l=0}^{K-1}(|q_{2l}|+|q_{2l+1}|)+ \\ &|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1}) - \sum_{l=0}^{K-1}x^{lM}p_l(x)|.\end{aligned}$$

Let us now bound

$$\begin{aligned}|(q_{2l}+q_{2l+1}) - x^{lM}p_l(x)| &\leq |(q_{2l}+q_{2l+1}) - (r_l+c_l)(e_l+f_l)|+ \\ &|(r_l+c_l)(e_l+f_l) - x^{lM}p_l(x)| \\ &\leq 7\mathbf{u}^2|(r_l+c_l)(e_l+f_l)|+ \\ &|(e_l+f_l)||(r_l+c_l)-p_l(x)|+ \\ &|p_l(x)||(e_l+f_l)-x^{lM}|\end{aligned}$$

As

$$\begin{aligned}|e_l+f_l| &\leq (1+\overline{\gamma}_{lM})|x^{lM}|, \\ |r_l+c_l| &\leq (1+\gamma_{2(M-1)}^2)\widetilde{p_l}(|x|),\end{aligned}$$

and with (13) and (14), we can deduce that

$$\begin{aligned}|(q_{2l}+q_{2l+1}) - x^{lM}p_l(x)| \leq 7\mathbf{u}^2(1+\overline{\gamma}_{lM})(1+\gamma_{2(M-1)}^2)\cdot \\ |x^{lM}|\widetilde{p_l}(|x|) + (1+\overline{\gamma}_{lM})\gamma_{2(M-1)}^2|x^{lM}|\widetilde{p_l}(|x|)+ \\ \overline{\gamma}_{lM}|x^{lM}|\widetilde{p_l}(|x|). \quad (15)\end{aligned}$$

As a consequence, we have

$$\begin{aligned}|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1}) - \sum_{l=0}^{K-1}x^{KM}p_l(x)| \leq (7\mathbf{u}^2(1+\overline{\gamma}_{KM})\cdot \\ (1+\gamma_{2(M-1)}^2) + (1+\overline{\gamma}_{KM})\gamma_{2(M-1)}^2 + \overline{\gamma}_{KM})\widetilde{p}(|x|).\end{aligned}$$

If we can keep only the terms in $\mathbf{u}^2$, we have

$$\begin{aligned}|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1}) - \sum_{l=0}^{K-1}x^{KM}p_l(x)| \leq [(7+4(M-1)^2+ \\ KM)\mathbf{u}^2 + \mathcal{O}(\mathbf{u}^3)]\widetilde{p}(|x|).\end{aligned}$$

Besides,

$$|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1})| \leq |p(x)| + |\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1}) - p(x)|$$

By definition, $|q_{2l+1}| \leq \mathbf{u}|q_{2l}|$ and $\mathrm{fl}(q_{2l}+q_{2l+1}) = q_{2l}$ so $|q_{2l}|+|q_{2l+1}| \leq (1+\mathbf{u})|q_{2l}|$ and then $|q_{2l}|+|q_{2l+1}| \leq (1+\mathbf{u})^2|q_{2l}+q_{2l+1}|$. It follows that

$$\begin{aligned}|\mathtt{res}-p(x)| \leq \mathbf{u}|p(x)| + (1+\mathbf{u})|\sum_{l=0}^{K-1}(q_{2l}+q_{2l+1}) - p(x)| \\ \gamma_{2n}^2\sum_{l=0}^{K-1}(|q_{2l}|+|q_{2l+1}|).\end{aligned}$$

We also have

$$\begin{aligned}|q_{2l}+q_{2l+1}| &\leq (1+7\mathbf{u}^2)|e_l+f_l||r_l+c_l| \\ &\leq (1+7\mathbf{u}^2)(1+\overline{\gamma}_{lM})(1+\gamma_{2(M-1)}^2)|x^{lM}|\widetilde{p_l}(|x|)\end{aligned}$$

so

$$\sum_{l=0}^{K-1}(|q_{2l}|+|q_{2l+1}|) \leq (1+7\mathbf{u}^2)(1+\overline{\gamma}_{KM})(1+\gamma_{2(M-1)}^2)\widetilde{p}(|x|).$$

Then it follows that

$$\begin{aligned}|\mathtt{res}-p(x)| \leq \mathbf{u}|p(x)| \\ [(7+4(M-1)^2+KM+4n^2)\mathbf{u}^2+\mathcal{O}(\mathbf{u}^3)]\widetilde{p}(|x|),\end{aligned}$$

and as $n+1 = KM$, we conclude that

$$\begin{aligned}|\mathtt{res}-p(x)| \leq \mathbf{u}|p(x)|+ \\ [(8+4(\frac{n+1-K}{K})^2+n+4n^2)\mathbf{u}^2+\mathcal{O}(\mathbf{u}^3)]\widetilde{p}(|x|),\end{aligned}$$

□

VI. Numerical experiments

All the tests were done on a computer under Linux Debian with 11th Gen Intel Core i5-1145G7 processor (4 cores) at 2.60GHz. The code is compiled with clang version 11.0.1-2 with the option `-Wall -O3 -march=native -ftree-vectorize`. The SIMD architecture used is AVX2 (registers with 256 bits). For all the experiments with native Horner, compensated Horner and parallel compensated Horner, we used IEEE 754 binary64 (double) precision. We compare the preformance and accuracy as well to a classical Horner when using IEEE754 binary128 (quad) precision. The code is available at

https://gitlab.com/cquirin/parallel-compensated-horner.

We did some performance evaluation work with ARM systems, trying to leverage their NEON units. The results have been inconclusive so far; the culprit is not with our algorithm but with lacking compiler support. These results are presented at the conference; further analysis of non-x86 systems is left to future work.

Figure 1 represents a comparison between the classic Horner scheme, the compensated Horner sheme (CHS) and the parallel compensated Horner scheme (PCHS) with SIMD. The CHS and PCHS have a similar behavior and so share a similar accuracy that is as accurate as if computed with twice the working precision and then rounded to the working precision.

The degree of the polynomial used for the experiment is 1023. The x-axis corresponds to the condition number defined by Equation (5).

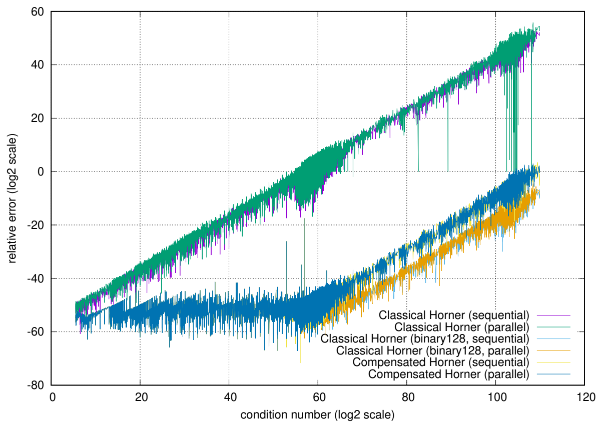

Figure 1. Comparison of the accuracy of evaluation algorithms with respect to condition number

Figure 2 shows of comparison of Horner scheme, CHS and PCHS in term of computing time depending on the degree of the polynomial. As expected, CHS is slower than the classical Horner scheme but PCHS is far more efficient than the classical Horner scheme and CHS. The SIMD architecture used is 16 virtual lanes. AVX2 has four physical lanes but the compiler can multiply that number by four, leveraging the different scheduling slots in the floating-point pipeline; we use all these 16 virtual lanes. Beyond 16 virtual lanes, register pressure becomes too high to allow for any additional benefits.

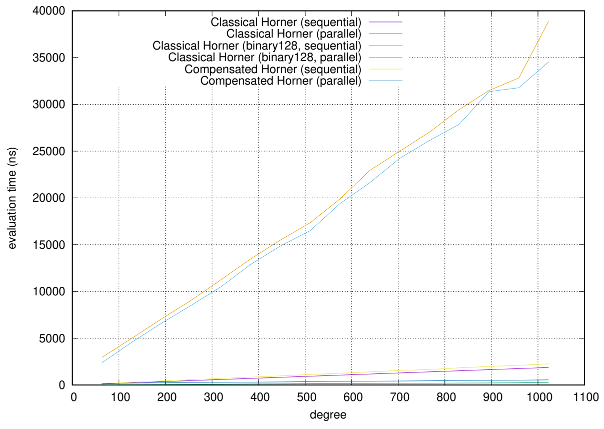

Figure 2. Comparison of the computing time with respect to the polynomial degree

Figure 3 shows the speedup of PCHS compared to CHS with respect to the number of SIMD lanes. As seen, the best result is obtained with 16 lanes that makes it possible to obtain a speedup of 4 for a polynomial with a large degree.

Figure 4 shows a comparison of the speedup with respect ot the degree of the polynomial. One can see that this speedup is better when one uses polynomials of large degree.

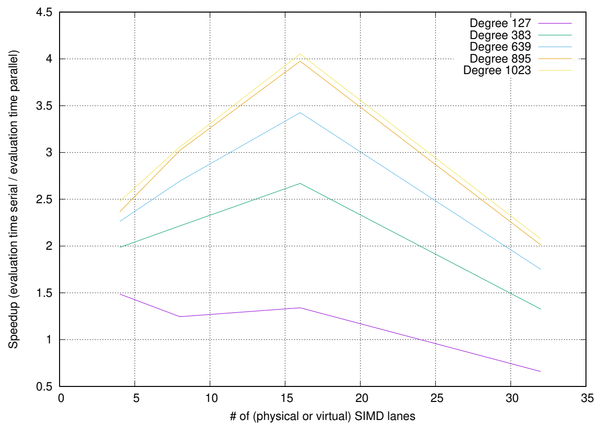

Figure 3. Comparison of the speedup with respect to the number of SIMD lanes

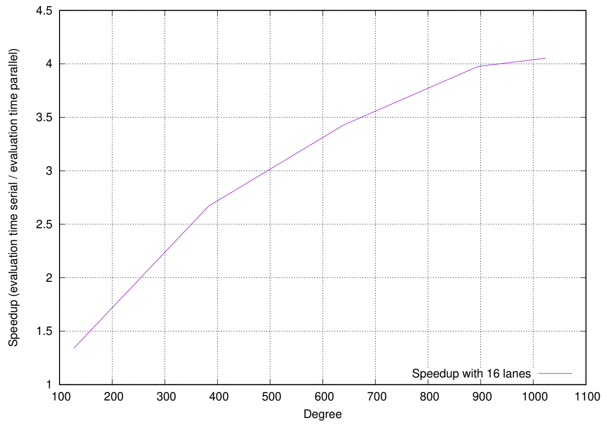

Figure 4. Comparison of the speedup with respect to the polynomial degree

VII. Conclusion

In this paper, we proposed a parallel algorithm to efficiently and accurately evaluate a polynomial. The algorithm provides a computed result which is as accurate as if computed in twice the working precision. The numerical experiments confirm that the proposed algorithm is fast in a SIMD environment. While the algorithm scales perfectly in theory, SIMD register pressure is a limiting factor to scalability in practice. This parallel algorithm can also be used to accurately evaluate rational functions [5]. This algorithm could also be used on shared memory computers with OpenMP or pthreads and even with MPI for distributed memory systems. However, it is expected that the overhead cost of these higher layers of parallelization can be compensated by the inherent speed of the algorithm only for very high degrees, for which other issues of floating-point arithmetic, such as limited exponent range, become preponderant. This will be developed in a future work.

Acknowledgments

The authors are grateful to the referees whose thoughtful comments significantly improved this article.

References

[1] David H. Bailey. *A Fortran-90 double-double library*, 2001. Available at URL = https://www.davidhbailey.com/dhbsoftware/.

[2] S. Boldo and J.-M. Muller. Some functions computable with a fused-mac. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the 17th Symposium on Computer Arithmetic*, pages 52–58, Cape Cod, USA, 2005.

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

[4] S. Graillat. Accurate floating-point product and exponentiation. *IEEE Trans. Comput.*, 58(7):994–1000, 2009.

[5] S. Graillat. An accurate algorithm for evaluating rational functions. *Appl. Math. Comput.*, 337:494–503, 2018.

[6] S. Graillat, Ph. Langlois, and N. Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan J. Indust. Appl. Math.*, 2-3(26):191–214, 2009. Special issue on State of the Art in Self-Validating Numerical Computations.

[7] S. Graillat, N. Louvet, and Ph. Langlois. Compensated Horner scheme. Research Report 04, Équipe de recherche DALI, Laboratoire LP2A, Université de Perpignan Via Domitia, France, 52 avenue Paul Alduy, 66860 Perpignan cedex, France, July 2005.

[8] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[9] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008),*. Institute of Electrical and Electronics Engineers, New York, 2019.

[10] M. Joldes, J.-M. Muller, and V. Popescu. Tight and rigorous error bounds for basic building blocks of double-word arithmetic. *ACM Trans. Math. Softw.*, 44(2), October 2017.

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

[12] Christoph Lauter and Marc Mezzarobba. Semi-automatic floating-point implementation of special functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 58–65, 2015.

[13] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018.

[14] Y. Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Trans. Math. Software*, 29(1):27–48, 2003.

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

[16] N. Yamanaka, T. Ogita, S. M. Rump, and S. Oishi. A parallel algorithm for accurate dot product. *Parallel Comput.*, 34(6-8):392–410, 2008.