A parallel compensated Horner scheme for SIMD architecture

Christoph Lauter

Computer Science Department, University of Texas at El Paso

Joint work with S. Graillat, Y. Ibrahimy, and C. Jeangoudoux

ARITH 2023, Portland, OR

Getting Things ~~Wrong~~ Right Fast

- The pre-ExaScale Summit Supercomputer can execute

  200795000000000000 operations

CC BY 2.0 C. Jones

Getting Things ~~Wrong~~ Right Fast

- The pre-ExaScale Summit Supercomputer can execute

  200795000000000000 operations *per second*

- Almost none of these operations are exactly correct

  Floating-point Operations are subject to roundoff error

Getting Things ~~Wrong~~ Right Fast

- The pre-ExaScale Summit Supercomputer can execute

  200795000000000000 operations *per second*

- Almost none of these operations are exactly correct

  Floating-point Operations are subject to roundoff error

- Can we still compute meaningful, rigorous results?
  - → Quantum field theory
  - → Supernova simulation
  - → Drugs research, Protein folding

Polynomials As Proxies for Functions

- Addition and Multiplication really fast on modern HW
- Division behind in performance

- General Transcendental Special Functions replaced by Polynomials

- Avoidance of domain splitting requires high degrees

- In IEEE754 FP arithmetic, the degree should stay well below the maximum exponent
  - $\Rightarrow$ Otherwise, constant underflow and overflow
  - $\Rightarrow$ Assume degree around 1024 for IEEE754 binary64

Need for Accuracy In Polynomial Evaluation

Horner evaluation:

$$p(x) = c_0 + x\,q(x)$$

- Cancellation can happen in the addition step
- Cancellation can even happen repeatedly in the Horner steps
- Faithful rounding: doubled precision needed
- Binary128 for Binary64 ?

The difficulty of evaluating a polynomial is captured by the condition number:

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

Need for Speed

IEEE754 binary128 precision up to 100 times slower than IEEE binary64

Error free transformations are properties and algorithms to compute the generated elementary rounding errors,

$$a, b \text{ entries } \in \mathbb{F}, \quad a \circ b = \text{fl}(a \circ b) + e, \text{ with } e \in \mathbb{F}$$

Key tools for accurate computation

- fixed length expansions libraries: double-double (Briggs, Bailey, Hida, Li, Lauter), quad-double (Bailey, Hida, Li)
- arbitrary length expansions libraries: Priest, Shewchuk, Joldes-Muller-Popescu
- compensated algorithms (Kahan, Priest, Ogita-Rump-Oishi)

Parallelizing the Unparallelizable Horner Scheme

- Horner Scheme is intrinsically serial

$$p(x) = c_0 + x\ (c_1 + x\ (c_2 + x\ (\dots) \dots))$$

- Parallelization needs to break the serial nature

$$\begin{aligned} p(x) &= p_0(x) + x^k\, p_1(x) + x^{2k}\, p_2 + \dots + x^{nk}\, p_n(x) \\ &= p_0(x) + x^k\ \left(p_1(x) + x^k\ (\dots) \dots\right) \end{aligned}$$

$$\begin{aligned} p(x) &= \tilde{p}_0(x^n) + x\, \tilde{p}_1(x^n) + x^2\, \tilde{p}_2(x^n) + \dots \\ &= \tilde{p}_0(x^n) + x\ (\tilde{p}_1(x^n) + x\ (\dots) \dots) \end{aligned}$$

Parallelizing the Unparallelizable Horner Scheme

- Horner Scheme is intrinsically serial

$$p(x) = c_0 + x\ (c_1 + x\ (c_2 + x\ (\dots)\dots))$$

- Parallelization needs to break the serial nature

$$\begin{aligned} p(x) &= p_0(x) + x^k\, p_1(x) + x^{2k}\, p_2 + \dots + x^{nk}\, p_n(x) \\ &= p_0(x) + x^k\ \left(p_1(x) + x^k\ (\dots)\dots\right) \end{aligned}$$

$$\begin{aligned} p(x) &= \tilde{p}_0(x^n) + x\,\tilde{p}_1(x^n) + x^2\,\tilde{p}_2(x^n) + \dots \\ &= \tilde{p}_0(x^n) + x\ (\tilde{p}_1(x^n) + x\ (\dots)\dots) \end{aligned}$$

- Only the very first form allows for FP error compensation

$$p(x) = p_0(x) + x^k\, p_1(x) + x^{2k}\, p_2(x) + \dots + x^{nk}\, p_n(x)$$

EFT for addition

$$x = a \oplus b \quad \Rightarrow \quad a + b = x + y \quad \text{with } y \in \mathbb{F},$$

Algorithm of Dekker (1971) and Knuth (1974)

Algorithm (EFT of the sum of 2 floating-point numbers)

function $[x, y] = \texttt{TwoSum}(a, b)$

$x = a \oplus b$

$z = x \ominus a$

$y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

EFT for multiplication

$$x = a \otimes b \quad \Rightarrow \quad a \times b = x + y \quad \text{with } y \in \mathbb{F},$$

Given $a, b, c \in \mathbb{F}$,

- $\texttt{FMA}(a, b, c)$ is the nearest floating-point number $a \times b + c \in \mathbb{F}$

Algorithm (EFT of the product of 2 floating-point numbers)

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

The FMA is available for example on PowerPC, Itanium, Cell, Xeon Phi, AMD and Nvidia GPU, Intel (Haswell), AMD (Bulldozer) processors.

Horner scheme

Algorithm

function $\texttt{res} = \texttt{Horner}(p, x)$ % $p(x) = \sum_{i=0}^{n} a_i x^i$

$s_n = a_n$

for $i = n - 1 : -1 : 0$

$p_i = s_{i+1} \otimes x$

$s_i = p_i \oplus a_i$

end

$\texttt{res} = s_0$

Condition number for the evaluation of $p(x)$:

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

Relative error bound: $$\frac{|p(x) - \texttt{Horner}(p, x)|}{|p(x)|} \leq \underbrace{\gamma_{2n}}_{\approx 2n\mathbf{u}} \mathrm{cond}(p, x)$$

Horner scheme

Algorithm

function $\texttt{res} = \texttt{Horner}(p, x)$ % $p(x) = \sum_{i=0}^{n} a_i x^i$

$s_n = a_n$

for $i = n - 1 : -1 : 0$

$p_i = s_{i+1} \otimes x$ % rounding error π_i

$s_i = p_i \oplus a_i$ % rounding error σ_i

end

$\texttt{res} = s_0$

Condition number for the evaluation of $p(x)$:

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

Relative error bound: $$\frac{|p(x) - \texttt{Horner}(p, x)|}{|p(x)|} \leq \underbrace{\gamma_{2n}}_{\approx 2n\mathbf{u}} \text{cond}(p, x)$$

EFT for Horner scheme

Algorithm (Graillat, Langlois, Louvet, 2008)

function $[h, p_\pi, p_\sigma] = \texttt{EFTHorner}(p, x)$

$s_n = a_n$

for $i = n-1 : -1 : 0$

$[p_i, \pi_i] = \texttt{TwoProd}(s_{i+1}, x)$

$[s_i, \sigma_i] = \texttt{TwoSum}(p_i, a_i)$

end

$h = s_0$

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

$$\boxed{p(x) = h + (p_\pi + p_\sigma)(x)} \qquad \text{with } h = \texttt{Horner}(p, x)$$

Compensated Horner scheme: Accuracy

Algorithm (Graillat, Langlois, Louvet, 2008)

function $\texttt{res} = \texttt{CompHorner}(p, x)$

$[h, p_\pi, p_\sigma] = \texttt{EFTHorner}(p, x)$

$c = \texttt{Horner}(p_\pi \oplus p_\sigma, x)$

$\texttt{res} = [h, c]$

Theorem (Graillat, Langlois, Louvet, 2008)

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 $\texttt{res} = [h, c] = \texttt{CompHorner}(p, x)$,

$$\frac{|h \oplus c - p(x)|}{|p(x)|} \leq \mathbf{u} + \underbrace{\gamma_{2n}^2}_{\approx 4n^2\mathbf{u}^2} \operatorname{cond}(p, x).$$

Compensated Algorithms And Double-Double

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 $|a_l| \leq \mathbf{u}|a_h|$.

Algorithm (Multiplication of double-double by a double)

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

Algorithm (Multiplication of two double-doubles)

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

Accuracy of Double-Double Multiplication

Theorem (Lauter, 2005, Joldes, Muller, Popescu, 2016)

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 :* $|\varepsilon| \leq 7\mathbf{u}^2$*. Furthermore, we have* $|r_l| \leq \mathbf{u}|r_h|$.

Computing Powers

Algorithm (Power evaluation with a compensated scheme, Graillat, 2009)

function $\texttt{res} = \texttt{CompLogPower}(x, n)$ $\quad\quad$ % $n = (n_t n_{t-1} \cdots n_1 n_0)_2$

$\quad [h, l] = [1, 0]$

$\quad$ for $i = t : -1 : 0$

$\quad\quad [h, l] = \texttt{prod\_dd\_dd}(h, l, h, l)$

$\quad\quad$ if $n_i = 1$

$\quad\quad\quad [h, l] = \texttt{prod\_dd\_d}(x, h, l)$

$\quad\quad$ end

$\quad$ end

$\quad \texttt{res} = [h, l]$

Complexity : $\mathcal{O}(\log n)$

Accuracy of Powering

Theorem (Graillat, 2009)

The two values h and l returned by Algorithm `CompLogPower` *satisfy*

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

with

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

For example, in double precision where $\mathbf{u} = 2^{-53}$, if $n < 2^{49} \approx 5 \cdot 10^{14}$, then we get a faithfully rounded result.

Summing Things Up

Algorithm (Compensated Summation, Ogita, Rump, Oishi, 2005)

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

Proposition (Ogita, Rump, Oishi, 2005)

Suppose Algorithm `CompSum` *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*

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

A Parallel Horner Scheme

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

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

Algorithm

function $\texttt{res} = \texttt{PHorner}(p, x)$

$K = (n+1)/M$

% begin parallel on K processors $(id = 0, \ldots, K-1)$

$y = x^{id \cdot M}$

$q(id) = y \otimes \texttt{Horner}(p_{id}, x)$

% end parallel

$\texttt{res} = \texttt{Sum}(q)$

A parallel compensated Horner scheme

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

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

Algorithm

function $\texttt{res} = \texttt{PCompHorner}(p, x)$

$K = (n+1)/M$

% begin parallel on K processors ($id = 0, \ldots, K-1$)

$[e, f] = \texttt{CompLogPower}(x, id \cdot M)$

$[r, c] = \texttt{CompHorner}(p_{id}, x)$

$[q(2 \cdot id), q(2 \cdot id + 1)] = \texttt{prod\_dd\_dd}(r, c, e, f)$

% end parallel

$\texttt{res} = \texttt{CompSum}(q)$

Accuracy of PCompHorner

Theorem

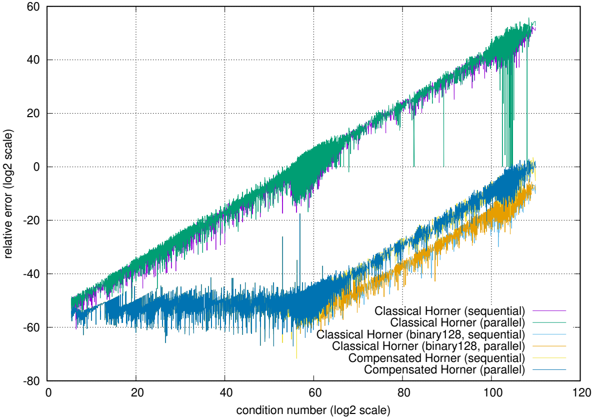
Let p be a polynomial of degree n with floating point coefficients, and x be a floating point value. Then if no underflow occurs, and $\texttt{res} = \texttt{PCompHorner}(p, x)$,

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

Numerical experiments: Accuracy

Linux Debian with 11th Gen Intel Core i5-1145G7 processor (4 cores, AVX2 @256bits regs) @ 2.60GHz,

compiling with clang version 11.0.1-2, options `-Wall -O3 -march=native -ftree-vectorize`

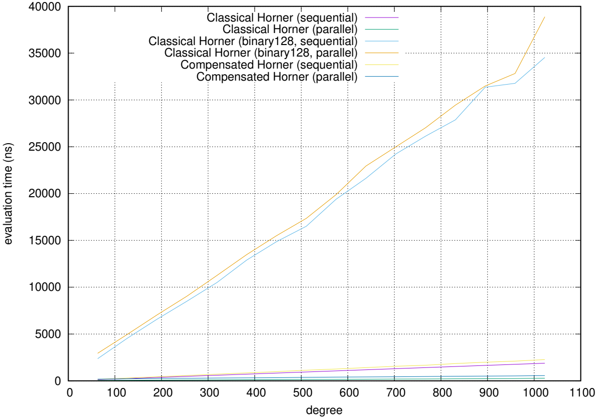


Lower is better.

Numerical experiments: Performance

Linux Debian with 11th Gen Intel Core i5-1145G7 processor (4 cores, AVX2 @256bits regs) @ 2.60GHz,

compiling with clang version 11.0.1-2, options `-Wall -O3 -march=native -ftree-vectorize`

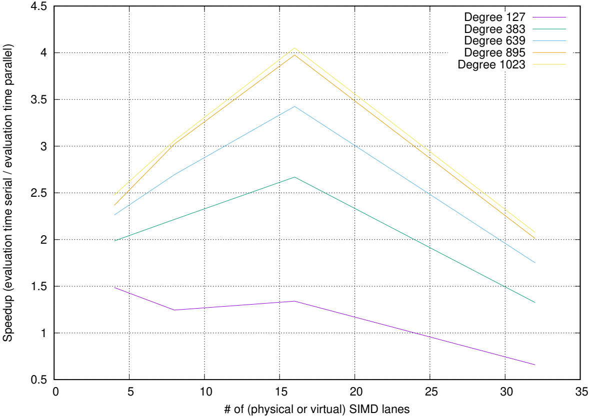


Lower is better.

Numerical experiments: Speedup vs. Lanes

Linux Debian with 11th Gen Intel Core i5-1145G7 processor (4 cores, AVX2 @256bits regs) @ 2.60GHz,

compiling with clang version 11.0.1-2, options `-Wall -O3 -march=native -ftree-vectorize`

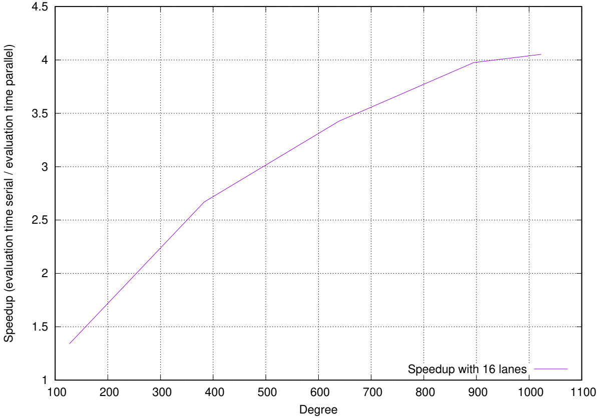


Higher is better.

Numerical experiments: Speedup vs. Degree

Linux Debian with 11th Gen Intel Core i5-1145G7 processor (4 cores, AVX2 @256bits regs) @ 2.60GHz,

compiling with clang version 11.0.1-2, options `-Wall -O3 -march=native -ftree-vectorize`



Higher is better.

Conclusion and future work

Conclusion

- We have presented a fast parallel compensated Horner scheme
- Scalability is acheived up to a certain point
- Accuracy is good, almost as good as using binary128 (100x)
- Polynomials stay of relatively low degree for IEEE754 FP Arithmetic

Future work

- Avoid use of powering algorithm, requires evaluation of derivatives
- Extend to polynomials with coefficients that are compensated
- Work on polynomial interpolation as another building brick