A survey of multiple precision computation using floating-point arithmetic

Fourth International Workshop on Taylor Methods

Christoph Quirin Lauter

Laboratoire de l'Informatique et du Parallélisme
École Normale Supérieure de Lyon

Boca Raton, December 16 - 19
Motivation

Exact floating-point arithmetic

Double-double, triple-double and expansion arithmetic
crlibm\(^1\): correctly rounded elementary function library

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\textsuperscript{1}: correctly rounded elementary function library

- Elementary functions \textit{as in an usual} libm:
  - exp
  - sin
  - cos
  - ...
crlibm\(^1\): correctly rounded elementary function library

- Elementary functions as in an usual libm:
  - exp
  - sin
  - cos
  - ...

- Evaluating elementary functions means evaluating polynomials

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\(^1\): correctly rounded elementary function library

- Elementary functions \textit{as in an usual} \texttt{libm}:
  - \texttt{exp}
  - \texttt{sin}
  - \texttt{cos}
  - ...  

- Evaluating elementary functions means evaluating polynomials
- Correct rounding requires high accuracy and \textit{complete proofs}

\(^1\text{http://lipforge.ens-lyon.fr/www/crlibm/}\)
Need for more precision

- IEEE 754 double precision offers 53 bits of precision
- In crlibm, we need an accuracy of “120 correct bits”
- In Taylor models, no use of high order polynomials if the remainder grows too fast
Need for more precision

- IEEE 754 double precision offers 53 bits of precision
- In crlibm, we need an accuracy of “120 correct bits”
- In Taylor models, no use of high order polynomials if the remainder grows too fast
- First approach:
  - Use an integer based fixed high precision floating-point library
  - Necessity to leave the floating-point pipeline
  - High impact on performance (factor 100)
IEEE 754 double precision offers 53 bits of precision
In crlibm, we need an accuracy of “120 correct bits”
In Taylor models, no use of high order polynomials if the remainder grows too fast
First approach:
- Use an integer based fixed high precision floating-point library
- Necessity to leave the floating-point pipeline
- High impact on performance (factor 100)
Second approach:
- Emulate higher precision in floating-point
- Reusage of already computed floating-point values possible
- No conversions, fill completely floating-point pipeline
- Speed-up by at least a factor 10 w.r.t. the first approach
- Same quality of certification possible
Floating-point expansions:

- **High precision significand**: 159 bits
- **FP - expansion**: 53 bits, 53 bits, 53 bits

Operations on expansions: for example addition:

```
<table>
<thead>
<tr>
<th>a_h</th>
<th>a_m</th>
<th>a_l</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_h</td>
<td>b_m</td>
<td>b_l</td>
</tr>
</tbody>
</table>
```

```
| c_h | c_m | c_l | δ (error) |
```
Need for exact floating-point arithmetic

- We want to implement:

\[
\begin{array}{c}
\phantom{+} a_h & a_m & a_l \\
+ & b_h & b_m & b_l \\
\hline
\phantom{+} c_h & c_m & c_l \\
\end{array}
\]

\[\delta \text{ (error)}\]

- Single step:

\[
\begin{array}{c}
\phantom{+} a_h \\
+ & b_h \\
\hline
\phantom{+} \text{temp}1_h & \text{temp}1_l \\
\end{array}
\]

\[\text{temp}1_h + \text{temp}1_l = a_h + b_h\]
Exact floating-point arithmetic

Motivation

Exact floating-point arithmetic

Double-double, triple-double and expansion arithmetic
Floating-point arithmetic can produce round-off error

\[
\begin{align*}
  a \otimes b &= a \cdot b \cdot (1 + \varepsilon) \\
  \text{where } |\varepsilon| &\leq 2^{-p}
\end{align*}
\]
Floating-point arithmetic can produce round-off error

\[ a \otimes b = a \cdot b \cdot (1 + \varepsilon) \]

where \(|\varepsilon| \leq 2^{-p}\)

A floating-point operation is called exact if its result is the mathematical one

\[ a \otimes b = a \cdot b \]

\[ \varepsilon = 0 \]
Floating-point arithmetic can produce round-off error

\[ a \otimes b = a \cdot b \cdot (1 + \varepsilon) \]

where \(|\varepsilon| \leq 2^{-p}\)

A floating-point operation is called exact if its result is the mathematical one

\[ a \otimes b = a \cdot b \]
\[ \varepsilon = 0 \]

However: floating-point arithmetic is often exact:

- Floating-point numbers are scaled integers
- If no integer overflow occurs, operations are exact on integers
- Just factorize the scale (where possible)

\[ a \otimes b = 2^{E_a} \cdot m_a \otimes 2^{E_b} \cdot m_b = 2^{E_a+E_b} \cdot \circ (m_a \cdot m_b) \]

where \(\circ\) is the rounding operator satisfying

\[ \forall x \in \mathbb{R} . \circ (x) = x \]
If tomorrow, you want to implement what I am going to show in the next slides, remember that...

- **Code here is in C** and that *Fortran* behaves differently
  - Implicit parentheses are elsewhere but our exact FP arithmetic requires the indicated operation order
  - Typing of *mixed precision expressions* is different
  - “Optimizations” the compiler is allowed to do are different

- Declaring variables as `double x,y,z;` does not imply usage of IEEE 754 double precision on most systems

- **Round-to-nearest rounding mode** required by some exact arithmetic sequences, in particular for exact multiplication

- Special care is needed for *subnormals*, underflow and overflow
Let be \( a, b \in \mathbb{F} \) such that

\[ \text{sgn}(a) = \text{sgn}(b) \]

and

\[ \frac{1}{2} \cdot |a| \leq |b| \leq 2 \cdot |a| \]

Thus

\[ a \oplus b = a - b \]
Sterbenz’ lemma

Let be \( a, b \in \mathbb{F} \) such that

\[
\text{sgn}(a) = \text{sgn}(b)
\]

and

\[
\frac{1}{2} \cdot |a| \leq |b| \leq 2 \cdot |a|
\]

Thus

\[
a \ominus b = a - b
\]

\[
\begin{align*}
2^E \\
a &= 2^E \cdot m_a \\
- b &= 2^E \cdot m_b \\
a \ominus b &= 2^E \cdot (m_a - m_b)
\end{align*}
\]

- At the base of **most extended precision addition algorithms**
- **Independent** of the rounding mode
- Proof intuition: factor the scale of both scaled integers that are \( a \) and \( b \)
Let round-to-nearest the current rounding mode in IEEE 754.
Let \( a, b \in \mathbb{F} \) such that \( |a| \geq |b| \).
Let be \( s, r \in \mathbb{F} \) computed by

\[
\begin{align*}
1 & \quad s = a + b \\
2 & \quad t = s - a \\
3 & \quad r = b - t ;
\end{align*}
\]

Thus

\[
s + r = a + b
\]

and

\[
|r| \leq \text{ulp}(s)
\]
Let round-to-nearest the current rounding mode in IEEE 754.
Let $a, b \in \mathbb{F}$ such that $|a| \geq |b|$.
Let be $s, r \in \mathbb{F}$ computed by

\begin{align*}
1. & \quad s = a + b; \\
2. & \quad t = s - a; \\
3. & \quad r = b - t;
\end{align*}

Thus

$$s + r = a + b$$

and

$$|r| \leq \text{ulp}(s)$$

Proof intuition: apply Sterbenz’ lemma
Let round-to-nearest the current rounding mode in IEEE 754. Let \( a, b \in \mathbb{F} \) such that \(|a| \geq |b|\). Let be \( s, r \in \mathbb{F} \) computed by

\[
\begin{align*}
1 & \quad s = a + b ; \\
2 & \quad t = s - a ; \\
3 & \quad r = b - t ;
\end{align*}
\]

Thus \( s + r = a + b \)

and \( |r| \leq \text{ulp}(s) \)

Proof intuition: apply Sterbenz’ lemma

Meaning of \( s \) and \( r \): \( s \) is a \textit{approximate} sum, \( r \) the absolute \textit{error}
2Sum

Let round-to-nearest the current rounding mode in IEEE 754.
Let \( a, b \in \mathbb{F} \).
Let be \( s, r \in \mathbb{F} \) computed by

\[
\begin{aligned}
& s = a + b; \\
& \textbf{if} (\text{fabs}(a) \geq \text{fabs}(b)) \{ \\
& \quad t = s - a; \\
& \quad r = b - t; \\
& \} \ \textbf{else} \ { \\
& \quad t = s - b; \\
& \quad r = a - t; \\
& \}
\end{aligned}
\]

Thus

\[ s + r = a + b \]

and

\[ |r| \leq \text{ulp}(s) \]
There are branches!

Branches are expensive on current pipelined processors!
Let round-to-nearest the current rounding mode in IEEE 754. 
Let \( a, b \in \mathbb{F} \).
Let be \( s, r \in \mathbb{F} \) computed by

\[
\begin{align*}
1 & \quad s = a + b; \\
2 & \quad t1 = s - a; \\
3 & \quad t2 = s - b; \\
4 & \quad d1 = b - t1; \\
5 & \quad d2 = a - t2; \\
6 & \quad r = d1 + d2; \\
\end{align*}
\]

Thus

\[ s + r = a + b \]

and

\[ |r| \leq \text{ulp}(s) \]
2Sum - avoiding branches

Let round-to-nearest the current rounding mode in IEEE 754.
Let $a, b \in \mathbb{F}$.
Let be $s, r \in \mathbb{F}$ computed by

1. $s = a + b$
2. $t1 = s - a$
3. $t2 = s - b$
4. $d1 = b - t1$
5. $d2 = a - t2$
6. $r = d1 + d2$

Thus

$s + r = a + b$

and

$|r| \leq \text{ulp}(s)$

$\Rightarrow$ Up to 10% performance gain w.r.t. branching version!
Round-to-nearest mode required?

I am doing interval arithmetic and I do not like to change the rounding-mode!
Let \( a, b \in \mathbb{F} \) such that \(|a| \geq |b|\).
Let be \( s, r \in \mathbb{F} \) computed by

\[
\begin{align*}
  &1 \quad s = a + b; \\
  &2 \quad e = s - a; \\
  &3 \quad g = s - e; \\
  &4 \quad h = g - a; \\
  &5 \quad f = b - h; \\
  &6 \quad r = f - e; \\
  &7 \quad \textbf{if} \ (r + e \neq f) \ {\{ \\
  &8 \quad \quad s = a; \\
  &9 \quad \quad r = b; \} }
\end{align*}
\]

Thus

\[
s + r = a + b
\]

and

\[
|r| \leq \text{ulp}(s)
\]
Addition: $s + r = a + b$
Addition: \( s + r = a + b \)

Multiplication - similarly: \( s + r = a \cdot b \)
Multiplication - Introduction

- Addition: \( s + r = a + b \)
- Multiplication - similarly: \( s + r = a \cdot b \)
- Is this possible?
- Addition: \( s + r = a + b \)
- Multiplication - similarly: \( s + r = a \cdot b \)
- Is this possible?

\[
\begin{array}{c}
\text{a} \\
* \\
\text{b} \\
\hline \\
\text{s} \\
\text{r}
\end{array}
\]
Addition: \( s + r = a + b \)

Multiplication - similarly: \( s + r = a \cdot b \)

Is this possible?

The significand of \( a \cdot b \) holds on a sum of two FP-numbers \( s + r \).
Addition: \( s + r = a + b \)

Multiplication - similarly: \( s + r = a \cdot b \)

Is this possible?

The significand of \( a \cdot b \) holds on a sum of two FP-numbers \( s + r \)

How do we compute \( s \) and \( r \)?
Suppose that the system supports a fused-multiply-and-add (FMA) operation: \( \text{FMA}(a, b, c) = \circ(a \cdot b + c) \).

Let be \( a, b \in \mathbb{F} \).

Let be \( s, r \in \mathbb{F} \) computed by

\[
\begin{align*}
  s &= a \ast b; \\
  r &= \text{FMA}(a, b, -s); \quad //\ r = \circ(a \cdot b - s)
\end{align*}
\]

Thus

\[ s + r = a \cdot b \]

and

\[ |r| \leq \text{ulp}(s) \]
Multiplication - Graphical “proof”

\[ a \cdot b \]

\[ s = o (a \cdot b) \]

\[ \text{Cancellation} \]

\[ r \]
Let be $a, b \in \mathbb{F}_p$ on $p$ bits

We want $s + r = a \cdot b$
Let be $a, b \in \mathbb{F}_p$ on $p$ bits

We want $s + r = a \cdot b$

Let be $a_h + a_l = a$ and $b_h + b_l = b$

Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$
Let be \( a, b \in \mathbb{F}_p \) on \( p \) bits

We want \( s + r = a \cdot b \)

Let be \( a_h + a_l = a \) and \( b_h + b_l = b \)

Clearly \( a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l \)

If \( a_h, a_l, b_h, b_l \) are written on at most \( p' \) bits, all products hold on \( 2 \cdot p' \) bits.
Let be $a, b \in \mathbb{F}_p$ on $p$ bits

We want $s + r = a \cdot b$

Let be $a_h + a_l = a$ and $b_h + b_l = b$

Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$

If $a_h, a_l, b_h, b_l$ are written on at most $p'$ bits, all products hold on $2 \cdot p'$ bits.

If $2 \cdot p' \leq p$ we can write:

$$a \cdot b = a_h \otimes b_h + a_h \otimes b_l + a_l \otimes b_h + a_l \otimes b_l$$
Let be $a, b \in \mathbb{F}_p$ on $p$ bits

We want $s + r = a \cdot b$

Let be $a_h + a_l = a$ and $b_h + b_l = b$

Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$

If $a_h, a_l, b_h, b_l$ are written on at most $p'$ bits, all products hold on $2 \cdot p'$ bits.

If $2 \cdot p' \leq p$ we can write:

$$a \cdot b = a_h \otimes b_h + a_h \otimes b_l + a_l \otimes b_h + a_l \otimes b_l$$

Since $a \cdot b$ holds on at most $2 \cdot p$ bits, there will be sufficient cancellation in the summation of the products producing $s + r$

$\Rightarrow$ Use here the exact 2Sum presented before.
Let be $a, b \in \mathbb{F}_p$ on $p$ bits

We want $s + r = a \cdot b$

Let be $a_h + a_l = a$ and $b_h + b_l = b$

Clearly $a \cdot b = a_h \cdot b_h + a_h \cdot b_l + a_l \cdot b_h + a_l \cdot b_l$

If $a_h, a_l, b_h, b_l$ are written on at most $p'$ bits, all products hold on $2 \cdot p'$ bits.

If $2 \cdot p' \leq p$ we can write:

$$a \cdot b = a_h \otimes b_h + a_h \otimes b_l + a_l \otimes b_h + a_l \otimes b_l$$

Since $a \cdot b$ holds on at most $2 \cdot p$ bits, there will be sufficient cancellation in the summation of the products producing $s + r$ ⇒ Use here the exact 2Sum presented before.

How can we compute $a_h + a_l = a$ ?
Let round-to-nearest the current rounding mode in IEEE 754.
Let \( a \in \mathbb{F}_p \) with precision \( p \).
Let be \( h, l \in \mathbb{F}_p \) computed by

\[
\begin{align*}
c &= 2^{p-k} + 1; \\
y &= c \cdot a; \\
z &= y - a; \\
h &= y - z; \\
l &= a - h;
\end{align*}
\]

Thus

\[ h + l = a \]

and \( h \) has at least \( k \) trailing zeros and \( l \) has at least \( p - k + 1 \) trailing zeros.
Division and square root

One can express only the backward error

\[ s = f(a - \delta) \]

instead of

\[ s + \delta = f(a) \]

as for addition and multiplication

Division:

\[ d = \frac{a - r}{b} \]

where \( d = a \odot b \in \mathbb{F} \) and \( r \in F \)

Square root:

\[ s = \sqrt{a - r} \]

where \( s = \circ \left( \sqrt{a} \right) \) and \( r \in \mathbb{F} \)

We can implement division and square root on expansions even with backward errors
Double-double, triple-double and expansion arithmetic

Motivation

Exact floating-point arithmetic

Double-double, triple-double and expansion arithmetic
Represent high precision numbers as *unevaluated sums* of floating-point numbers

\[ x = \sum_{i=1}^{n} x_i \]

Suppose native precision to be IEEE 754 double precision
- \( n = 2 \): “double-double” – \( \approx 102 \) bits of accuracy
- \( n = 3 \): “triple-double” – \( \approx 150 \) bits of accuracy
- \( n = 4 \): Bailey: “quad-double”
- any \( n \): expansions
Operations on expansions:

- Addition – Use 2Sum algorithm for carries
- Multiplication – Partial products using 2Mult, sum up using 2Sum
- Division – Euclid’s division using an exact backward error sequence or Newton’s method
- Square root – Newton’s method
- Renormalization – use 2Sums and tests for bringing expansions to a non-overlapping form
Operations on expansions:

- Addition – Use 2Sum algorithm for carries
- Multiplication – Partial products using 2Mult, sum up using 2Sum
- Division – Euclid’s division using an exact backward error sequence or Newton’s method
- Square root – Newton’s method
- Renormalization – use 2Sums and tests for bringing expansions to a non-overlapping form

Cost:

- **No conversions** between floating-point and integer \(\Rightarrow\) double-double and triple-double is much faster
- Expansions are inefficient: the exponents are redundant information
- Floating-point arithmetic has some bizarre behaviours: \(\Rightarrow\) general expansions seem to be more expensive than integer based methods because of a high number of tests
Double-double and triple-double in crlibm

- Full implementation of double-double
  - Versions for 2Sum and 2Mult optimized for different processors (FMA, FABS, \ldots)
  - All combinations double + double, double-double + double etc.
  - Accuracy proof for each operator; proof can already be formally verified (Gappa)

Almost complete implementation of triple-double
Based on double-double
Almost all combinations double, double-double or triple-double in operand or result
Accuracy proof of each operator
Approach for avoiding renormalizations whilst being rigorous
No branches on common machines
Correct (IEEE 754) rounding to double implemented
Automatic routines for generating double, double-double and triple-double code for evaluating complete polynomials in Horner's scheme with formal proof generation
Double-double and triple-double in crlibm

- Full implementation of double-double
  - Versions for 2Sum and 2Mult optimized for different processors (FMA, FABS, ...)
  - All combinations double + double, double-double + double etc.
  - **Accuracy proof** for each operator; proof can already be formally verified (Gappa)

- Almost complete implementation of triple-double
  - Based on double-double
  - Almost all combinations double, double-double or triple-double in operand or result
  - **Accuracy proof** of each operator
  - **Approach for avoiding renormalizations** whilst being rigorous
  - No branches on common machines
  - Correct (IEEE 754) **rounding to double** implemented
Double-double and triple-double in crlibm

- Full implementation of double-double
  - Versions for 2Sum and 2Mult optimized for different processors (FMA, FABS, ...)
  - All combinations double + double, double-double + double etc.
  - Accuracy proof for each operator; proof can already be formally verified (Gappa)
- Almost complete implementation of triple-double
  - Based on double-double
  - Almost all combinations double, double-double or triple-double in operand or result
  - Accuracy proof of each operator
  - Approach for avoiding renormalizations whilst being rigorous
  - No branches on common machines
  - Correct (IEEE 754) rounding to double implemented
- Automatic routines for generating double, double-double and triple-double code for evaluating complete polynomials in Horner’s scheme with formal proof generation
Logarithm - evaluate polynomials of degree about 12 – 20

<table>
<thead>
<tr>
<th>Library</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPFR - integer based multiprec.</td>
<td>12942</td>
</tr>
<tr>
<td>crlibm portable using integer based multiprec.</td>
<td>2748</td>
</tr>
<tr>
<td>crlibm portable using <strong>triple-double</strong></td>
<td><strong>266</strong></td>
</tr>
</tbody>
</table>

Exponential - evaluate polynomials of degree about 7 – 15

<table>
<thead>
<tr>
<th>Library</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPFR - integer based multiprec.</td>
<td>4908</td>
</tr>
<tr>
<td>crlibm portable using integer based multiprec.</td>
<td>1976</td>
</tr>
<tr>
<td>crlibm portable using <strong>triple-double</strong></td>
<td><strong>258</strong></td>
</tr>
</tbody>
</table>
Conclusion

- Presentation of exact floating-point arithmetic
- Overview over general techniques for expansions
- Double-double and triple-double are quite efficient
  - No branches needed
  - No conversions needed
  - Speed-up of a factor of about 10
- Rigourous proofs are possible (Gappa)
- General expansion algorithms known but rarely implemented