Accurate Complex Multiplication in Floating-Point Arithmetic

Vincent Lefèvre  Jean-Michel Muller.
Université de Lyon, CNRS, Inria, France.

Arith26,
Kyoto, June 2019
Accurate complex multiplication in FP arithmetic

- \( \omega \cdot x \), emphasis on the case where \( \Re(\omega) \) and \( \Im(\omega) \) are double-word numbers—i.e., pairs (high-order, low-order) of FP numbers;
- **Applications**: Fourier transforms, iterated products.

**Assumptions:**
- radix-2, precision-\( p \), FP arithmetic;
- rounded to nearest (RN) FP operations;
- an FMA instruction is available;
- underflow/overflow do not occur.

**Bound on relative error of (real) operations:**

\[
|\text{RN}(a + b) - (a + b)| \leq \frac{u}{1 + u} \cdot |a + b| < u \cdot |a + b|,
\]

where \( u \) (rounding unit) equals \( 2^{-p} \).
Some variables: **double-word (DW) numbers**

- also called double-double in the literature;
- \( \nu \in \mathbb{R} \) represented by a pair of FP numbers \( \nu_h \) and \( \nu_\ell \) such that
  \[
  \nu = \nu_h + \nu_\ell, \\
  |\nu_\ell| \leq \frac{1}{2} \text{ulp}(\nu) \leq u \cdot |\nu|.
  \]

- algorithms and libraries for manipulating DW numbers: QD (Hida, Li & Bailey), Campary (Joldes, Popescu & others),
- use the 2Sum, Fast2Sum & Fast2Mult algorithms (see later).
Naive algorithms for complex FP multiplication

- straightforward transcription of the formula

\[ z = (x^R + ix^I) \cdot (y^R + iy^I) = (x^R y^R - x^I y^I) + i \cdot (x^I y^R + x^R y^I); \]

- bad solution if componentwise relative error is to be minimized;

- adequate solution if normwise relative error is at stake.

(\( \hat{z} \) approximates \( z \) with normwise error \(|(\hat{z} - z)/z|\))

**Algorithms:**

- if no FMA instruction is available

\[
\begin{align*}
\hat{z}^R &= \text{RN}(\text{RN}(x^R y^R) - \text{RN}(x^I y^I)), \\
\hat{z}^I &= \text{RN}(\text{RN}(x^R y^I) + \text{RN}(x^I y^R)).
\end{align*}
\]

(1)

- if an FMA instruction is available

\[
\begin{align*}
\hat{z}^R &= \text{RN}(x^R y^R - x^R y^I), \\
\hat{z}^I &= \text{RN}(x^R y^I + x^I y^R).
\end{align*}
\]

(2)
Naive algorithms for complex multiplication

• if no FMA instruction is available

\[
\begin{align*}
\hat{z}^R &= \text{RN}(\text{RN}(x^R y^R) - \text{RN}(x' y'))), \\
\hat{z}^I &= \text{RN}(\text{RN}(x^R y'^I) + \text{RN}(x'^I y^R)).
\end{align*}
\] 

(1)

• if an FMA instruction is available

\[
\begin{align*}
\hat{z}^R &= \text{RN}(x^R y^R - \text{RN}(x' y'))), \\
\hat{z}^I &= \text{RN}(x^R y'^I + \text{RN}(x'^I y^R)).
\end{align*}
\] 

(2)

Asymptotically optimal bounds on the normwise relative error of (1) and (2) are known:

• Brent et al (2007): bound $\sqrt{5} \cdot u$ for (1),

• Jeannerod et al. (2017): bound $2 \cdot u$ for (2).
Accurate complex multiplication

Our goal:

- smaller normwise relative errors,
- closer to the best possible one (i.e., $u$, unless we output DW numbers),
- at the cost of more complex algorithms.

We consider the product

$$\omega \cdot x,$$

with

$$\omega = \omega^R + i \cdot \omega^I$$
and
$$x = x^R + i \cdot x^I,$$

where:

- $\omega^R$ and $\omega^I$ are DW numbers (special case FP considered later)
- $x^R$ and $x^I$ are FP numbers.
Basic building blocks: Error-Free Transforms

Expressing \( a + b \) as a DW number

**Algorithm 1: 2Sum\((a, b)\).** Returns \( s \) and \( t \) such that \( s = \text{RN}(a + b) \) and \( t = a + b - s \)

\[
\begin{align*}
    s & \leftarrow \text{RN}(a + b) \\
    a' & \leftarrow \text{RN}(s - b) \\
    b' & \leftarrow \text{RN}(s - a') \\
    \delta_a & \leftarrow \text{RN}(a - a') \\
    \delta_b & \leftarrow \text{RN}(b - b') \\
    t & \leftarrow \text{RN}(\delta_a + \delta_b)
\end{align*}
\]

Expressing \( a \cdot b \) as a DW number

**Algorithm 2: Fast2Mult\((a, b)\).** Returns \( \pi \) and \( \rho \) such that \( \pi = \text{RN}(ab) \) and \( \rho = ab - \pi \)

\[
\begin{align*}
    \pi & \leftarrow \text{RN}(ab) \\
    \rho & \leftarrow \text{RN}(ab - \pi)
\end{align*}
\]
The multiplication algorithm

- $\omega^R = \Re(\omega)$ and $\omega^I = \Im(\omega)$: DW numbers, i.e.,
  \[
  \omega = \omega^R + i \cdot \omega^I = (\omega^R_h + \omega^R_\ell) + i \cdot (\omega^I_h + \omega^I_\ell),
  \]
  where $\omega^R_h$, $\omega^R_\ell$, $\omega^I_h$, and $\omega^I_\ell$ are FP numbers that satisfy:
  - $|\omega^R_\ell| \leq \frac{1}{2} \text{ulp}(\omega^R) \leq u \cdot |\omega^R|$;
  - $|\omega^I_\ell| \leq \frac{1}{2} \text{ulp}(\omega^I) \leq u \cdot |\omega^I|$.

- Real part $z^R$ of the result (similar for imaginary part):
  - difference $v^R_h$ of the high-order parts of $\omega^R_h x^R$ and $\omega^R_h x^R$,
  - add approximated sum $\gamma^R_\ell$ of all the error terms that may have a significant influence on the normwise relative error.

- rather straightforward algorithms: the tricky part is the error bounds.
Real part $(\omega_h^R + \omega_\ell^R) \cdot x^R - (\omega_h^I + \omega_\ell^I) \cdot x^I$
The multiplication algorithm

**Algorithm 3:** Computes $\omega \cdot x$, where the real & imaginary parts of $\omega = (\omega_h^R + \omega_\ell^R) + i \cdot (\omega_h^I + \omega_\ell^I)$ are DW, and the real & im. parts of $x$ are FP.

1: $t^R \leftarrow \text{RN}(\omega_\ell^I x^I)$
2: $\pi_\ell^R \leftarrow \text{RN}(\omega_\ell^R x^R - t^R)$
3: $(P_h^R, P_\ell^R) \leftarrow \text{Fast2Mult}(\omega_h^I, x^I)$
4: $r_\ell^R \leftarrow \text{RN}(\pi_\ell^R - P_\ell^R)$
5: $(Q_h^R, Q_\ell^R) \leftarrow \text{Fast2Mult}(\omega_h^R, x^R)$
6: $s_\ell^R \leftarrow \text{RN}(Q_\ell^R + r_\ell^R)$
7: $(v_h^R, v_\ell^R) \leftarrow \text{2Sum}(Q_h^R, -P_h^R)$
8: $\gamma_\ell^R \leftarrow \text{RN}(v_\ell^R + s_\ell^R)$
9: return $z^R = \text{RN}(v_h^R + \gamma_\ell^R)$ (real part)
10: $t^I \leftarrow \text{RN}(\omega_\ell^I x^R)$
11: $\pi_\ell^I \leftarrow \text{RN}(\omega_\ell^R x^I + t^I)$
12: $(P_h^I, P_\ell^I) \leftarrow \text{Fast2Mult}(\omega_h^I, x^R)$
13: $r_\ell^I \leftarrow \text{RN}(\pi_\ell^I + P_\ell^I)$
14: $(Q_h^I, Q_\ell^I) \leftarrow \text{Fast2Mult}(\omega_h^R, x^I)$
15: $s_\ell^I \leftarrow \text{RN}(Q_\ell^I + r_\ell^I)$
16: $(v_h^I, v_\ell^I) \leftarrow \text{2Sum}(Q_h^I, P_h^I)$
17: $\gamma_\ell^I \leftarrow \text{RN}(v_\ell^I + s_\ell^I)$
18: return $z^I = \text{RN}(v_h^I + \gamma_\ell^I)$ (imaginary part)
The multiplication algorithm

Theorem 1
As soon as \( p \geq 4 \), the normwise relative error \( \eta \) of Algorithm 3 satisfies

\[
\eta < u + 33u^2.
\]

(remember: the best possible bound is \( u \))

Remarks:

- Condition “\( p \geq 4 \)” always holds in practice;
- Algorithm 3 easily transformed (see later) into an algorithm that returns the real and imaginary parts of \( z \) as DW numbers.
Sketch of the proof

first, we show that

\[ |z^R - \Re(wx)| \leq \alpha n^R + \beta N^R, \]
\[ |z^I - \Im(wx)| \leq \alpha n^I + \beta N^I, \]

with

\[ N^R = |\omega^R x^R| + |\omega^I x^I|, \]
\[ n^R = |\omega^R x^R - \omega^I x^I|, \]
\[ N^I = |\omega^R x^I| + |\omega^I x^R|, \]
\[ n^I = |\omega^R x^I + \omega^I x^R|, \]
\[ \alpha = u + 3u^2 + u^3, \]
\[ \beta = 15u^2 + 38u^3 + 39u^4 + 22u^5 + 7u^6 + u^7; \]

then we deduce

\[ \eta^2 = \frac{(z^R - \Re(wx))^2 + (z^I - \Im(wx))^2}{(\Re(wx))^2 + (\Im(wx))^2} \leq \alpha^2 + (2\alpha\beta + \beta^2) \cdot \frac{(N^R)^2 + (N^I)^2}{(n^R)^2 + (n^I)^2}; \]

the theorem follows, by using

\[ \frac{(N^R)^2 + (N^I)^2}{(n^R)^2 + (n^I)^2} \leq 2. \]
Obtaining the real and imaginary parts of \( z \) as DW numbers

- replace the FP addition \( z^R = \text{RN}(v_h^R + \gamma_\ell^R) \) of line 9 of Algorithm 3 by a call to \( 2\text{Sum}(v_h^R, \gamma_\ell^R) \),

- replace the FP addition \( z^I = \text{RN}(v_h^I + \gamma_\ell^I) \) of line 18 by a call to \( 2\text{Sum}(v_h^I, \gamma_\ell^I) \).

- resulting relative error

\[
\sqrt{241} \cdot u^2 + O(u^3) \approx 15.53u^2 + O(u^3)
\]

(instead of \( u + 33u^2 \)).

Interest:

- **iterative product** \( z_1 \times z_2 \times \cdots \times z_n \): keep the real and imaginary parts of the partial products as DW numbers,

- **Fourier transforms**: when computing \( z_1 \pm \omega z_2 \), keep \( \Re(\omega z_2) \) and \( \Im(\omega z_2) \) as DW numbers before the \( \pm \).
If $\omega^I$ and $\omega^R$ are floating-point numbers

$\omega^I_\ell = \omega^R_\ell = 0 \Rightarrow$ Algorithm 3 becomes simpler:

**Algorithm 4:** Complex multiplication $\omega \cdot x$, where $\Re(\omega)$ and $\Im(\omega)$ are FP numbers.

1: $(P^R_h, P^R_\ell) \leftarrow \text{Fast2Mult}(\omega^I, x^I)$
2: $(Q^R_h, Q^R_\ell) \leftarrow \text{Fast2Mult}(\omega^R, x^R)$
3: $s^R_\ell \leftarrow \text{RN}(Q^R_\ell - P^R_\ell)$
4: $(v^R_h, v^R_\ell) \leftarrow \text{2Sum}(Q^R_h, -P^R_h)$
5: $\gamma^R_\ell \leftarrow \text{RN}(v^R_\ell + s^R_\ell)$
6: **return** $z^R = \text{RN}(v^R_h + \gamma^R_\ell)$ (real part)
7: $(P^I_h, P^I_\ell) \leftarrow \text{Fast2Mult}(\omega^I, x^R)$
8: $(Q^I_h, Q^I_\ell) \leftarrow \text{Fast2Mult}(\omega^R, x^I)$
9: $s^I_\ell \leftarrow \text{RN}(Q^I_\ell + P^I_\ell)$
10: $(v^I_h, v^I_\ell) \leftarrow \text{2Sum}(Q^I_h, P^I_h)$
11: $\gamma^I_\ell \leftarrow \text{RN}(v^I_\ell + s^I_\ell)$
12: **return** $z^I = \text{RN}(v^I_h + \gamma^I_\ell)$ (imaginary part)
Real part

\[
\begin{align*}
\omega_\ell^I & \times x^I \\
\omega_\ell^R & \times x^R
\end{align*}
\]
Real part
If $\omega^I$ and $\omega^R$ are floating-point numbers

- Real and complex parts of Algorithm 4 similar to:
  - Cornea, Harrison and Tang’s algorithm for $ab + cd$ (with a “+” replaced by a 2Sum),
  - Alg. 5.3 in Ogita, Rump and Oishi’s *Accurate sum & dot product* (with different order of summation of $P^R_\ell$, $Q^R_\ell$ & $v^R_\ell$).

- The error bound $u + 33u^2$ of Theorem 1 still applies, but it can be slightly improved:

**Theorem 2**

*As soon as $p \geq 4$, the normwise relative error $\eta$ of Algorithm 4 satisfies*

$$\eta < u + 19u^2.$$
Implementations and experiments

- Main algorithm (Algorithm 3) implemented in binary64 (a.k.a. double-precision) arithmetic, compared with other solutions:
  - naive formula in binary64 arithmetic;
  - naive formula in binary128 arithmetic;
  - GNU MPFR with precision ranging from 53 to 106 bits.

- Loop over $N$ random inputs, itself inside another loop doing $K$ iterations;

- Goal of the external loop: get accurate timings without having to choose a large $N$, with input data that would not fit in the cache;

- For each test, we chose $(N, K) = (1024, 65536), (2048, 32768)$ and $(4096, 16384)$. 
Implementation and experiments

▶ tests run on two computers with a hardware FMA:
  • x86_64 with Intel Xeon E5-2609 v3 CPUs, under Linux (Debian/unstable), with GCC 8.2.0 and a Clang 8 preversion, using -march=native;
  • ppc64le with POWER9 CPUs, under Linux (CentOS 7), with GCC 8.2.1, using -mcpu=power9.

▶ options -O3 and -O2.

▶ With GCC, -O3 -fno-tree-slp-vectorize also used to avoid a loss of performance with some vectorized codes.

▶ In all cases, -static used to avoid the overhead due to function calls to dynamic libraries.
Table 1: Timings on x86_64 (in secs, for $NK = 2^{26}$ ops) with GCC. GNU MPFR is used with separate $\pm$ and $\times$.

<table>
<thead>
<tr>
<th></th>
<th>minimums</th>
<th>maximums</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N \rightarrow$</td>
<td>1024 2048 4096</td>
<td>1024 2048 4096</td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O3</td>
<td>Algorithm 3</td>
<td>0.92 0.97 0.97</td>
</tr>
<tr>
<td></td>
<td>Naive, Binary64</td>
<td>0.61 0.61 0.62</td>
</tr>
<tr>
<td></td>
<td>GNU MPFR</td>
<td>12.59 13.01 13.12</td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-O2</td>
<td>Algorithm 3</td>
<td>0.91 0.97 0.97</td>
</tr>
<tr>
<td></td>
<td>Naive, Binary64</td>
<td>0.61 0.62 0.62</td>
</tr>
<tr>
<td></td>
<td>GNU MPFR</td>
<td>12.31 12.74 12.85</td>
</tr>
</tbody>
</table>
Implementación y experimentos

**Tabla 2**: Tiempos en x86_64 (en segundos, para $NK = 2^{26}$ operaciones) con Clang.

<table>
<thead>
<tr>
<th></th>
<th>$N$ →</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm 3</td>
<td></td>
<td>0.86</td>
<td>1.09</td>
<td>1.10</td>
<td>0.96</td>
<td>1.15</td>
<td>1.15</td>
</tr>
<tr>
<td>Naive, Binary64</td>
<td></td>
<td>0.39</td>
<td>0.61</td>
<td>0.63</td>
<td>0.47</td>
<td>0.65</td>
<td>0.66</td>
</tr>
<tr>
<td>GNU MPFR</td>
<td></td>
<td>12.24</td>
<td>12.63</td>
<td>12.72</td>
<td>22.91</td>
<td>22.94</td>
<td>22.97</td>
</tr>
<tr>
<td>clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithm 3</td>
<td></td>
<td>0.88</td>
<td>1.08</td>
<td>1.10</td>
<td>0.96</td>
<td>1.14</td>
<td>1.15</td>
</tr>
<tr>
<td>Naive, Binary64</td>
<td></td>
<td>0.40</td>
<td>0.61</td>
<td>0.63</td>
<td>0.48</td>
<td>0.65</td>
<td>0.66</td>
</tr>
<tr>
<td>GNU MPFR</td>
<td></td>
<td>12.15</td>
<td>12.54</td>
<td>12.65</td>
<td>23.15</td>
<td>23.21</td>
<td>23.21</td>
</tr>
</tbody>
</table>
Table 3: Timings on a POWER9 (in secs, for $NK = 2^{26}$ ops). The POWER9 has hardware support for Binary128.

<table>
<thead>
<tr>
<th>gcc -03 -f...</th>
<th>Algorithm 3</th>
<th>minimums</th>
<th>maximums</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N \rightarrow$</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>0.47</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>2.22</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>Naive, Binary64</td>
<td>16.42</td>
<td>16.59</td>
</tr>
<tr>
<td></td>
<td>Naive, Binary128</td>
<td>GNU MPFR</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>GNU MPFR</td>
<td>0.47</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>16.36</td>
<td>16.58</td>
<td>16.63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>gcc -02</th>
<th>Algorithm 3</th>
<th>minimums</th>
<th>maximums</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N \rightarrow$</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>0.47</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>2.22</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>Naive, Binary64</td>
<td>16.36</td>
<td>16.58</td>
</tr>
<tr>
<td></td>
<td>Naive, Binary128</td>
<td>GNU MPFR</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>GNU MPFR</td>
<td>0.47</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>2.22</td>
<td>2.22</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>16.36</td>
<td>16.58</td>
<td>16.63</td>
</tr>
</tbody>
</table>
Implementation and experiments

- **Naive formula in binary64** (inlined code) \( \approx \) two times as fast as our implementation of Algorithm 3, but significantly less accurate;

- **Naive formula in binary128**, using the `__float128` C type (inlined code):
  - x86_64: from 19 to 25 times as slow as Algorithm 3,
  - on POWER9: 2.3 times as slow.

- **GNU MPFR** using precisions from 53 to 106: from 11 to 26 times as slow as Algorithm 3 on x86_64, and from 17 to 31 times as slow on POWER9.

The error bound of Theorem 1 is tight: In Binary64 arithmetic, with

\[
\begin{align*}
\omega^R &= 0x1.d1ef9ea4aa013p-1 + 0x1.ae88ba2a277ep-56 \\
\omega^I &= 0x1.f5c28321df365p-81 + 0x1.c4c3e7b506d06p-135 \\
\chi^R &= 0x1.194f298b4d152p-1 \\
\chi^I &= 0x1.5c1fdca444f7cp-14
\end{align*}
\]

the normwise relative error is \(0.99999900913907117123u\).
Conclusion

- **Main algorithm:**
  - the real and imaginary parts of one of the operands are DW, and for the other one they are FP,
  - normwise relative error bound close to the best one ($u$) that one can guarantee,
  - only twice as slow as a naive multiplication,
  - much faster than binary128 or multiple-precision software.

- **2 variants:**
  - real and imaginary parts of the output are DW,
  - real and imaginary parts of the inputs are FP.
Conclusion

- **Main algorithm:**
  - the real and imaginary parts of one of the operands are DW, and for the other one they are FP,
  - normwise relative error bound close to the best one \((u)\) that one can guarantee,
  - only twice as slow as a naive multiplication,
  - much faster than binary128 or multiple-precision software.

- **2 variants:**
  - real and imaginary parts of the output are DW,
  - real and imaginary parts of the inputs are FP.

Thank you!