More accurate complex multiplication for embedded processors

Claude-Pierre Jeannerod*, Christophe Monat†, Laurent Thévenoux*

*Inria – Laboratoire LIP (CNRS, ENS de Lyon, Inria, UCBL), Université de Lyon, France
† STMicroelectronics, Compilation Expertise Center, Grenoble, France

***Abstract*—This paper presents some work in progress on the development of fast and accurate support for complex floating-point arithmetic on embedded processors. Focusing on the case of multiplication, we describe algorithms and implementations for computing both the real and imaginary parts with high relative accuracy. We show that, in practice, such accuracy guarantees can be achieved with reasonable overhead compared with conventional algorithms (which are those offered by current implementations and for which the real or imaginary part of a product can have no correct digit at all). For example, the average execution-time overheads when computing an FFT on the ARM Cortex-A53 and -A57 processors range from 1.04x to 1.17x only, while arithmetic costs suggest overheads from 1.5x to 1.8x.**

I. Introduction

Complex arithmetic is ubiquitous in scientific computing and signal processing applications, and for recent embedded processors with floating-point units it is in general supported by the compiler via direct floating-point translations of the usual formulas. For example, given four floating-point numbers a, b, c, d, the sum and the product of the complex numbers $a + ib$ and $c + id$ are obtained by evaluating in floating-point the real and imaginary parts of $(a + c) + i(b + d)$ and $(ac - bd) + i(ad + bc)$, respectively.

For complex addition this is entirely satisfactory: assuming that the available floating-point units comply with the IEEE 754 standard [1], both $a+c$ and $b+d$ will be computed with perfect accuracy as $\mathrm{RN}(a + c)$ and $\mathrm{RN}(b + d)$, with RN denoting round to nearest. For complex multiplication, however, evaluating $ac - bd$ and $ad + bc$ as

$$\mathrm{RN}(\mathrm{RN}(ac) - \mathrm{RN}(bd)) \quad \text{and} \quad \mathrm{RN}(\mathrm{RN}(ad) + \mathrm{RN}(bc))$$

can result in one of these two quantities having no correct digit at all. For example, in double precision, taking $a = 1 + 2^{-51}$, $b = 1 + 3 \cdot 2^{-52}$, $c = 1 - 2^{-53}$ and $d = 1 - 3 \cdot 2^{-53}$ gives

$$\mathrm{RN}(\mathrm{RN}(ac) - \mathrm{RN}(bd)) = 0,$$

while the exact result $ac - bd$ equals $7 \cdot 2^{-105} \approx 1.72 \times 10^{-31}$. In other words, the computed real part has relative error 1 and thus not even a single correct digit.

With a fused multiply-add (FMA) instruction, which evaluates expressions of the form $ab + c$ with one rounding as $\mathrm{RN}(ab+c)$, things can be even worse. Indeed, not only all the digits of $ac - bd$ but also its sign can be computed incorrectly, depending on the product chosen to be fused. For example, with the same inputs as before we now have

$$\mathrm{RN}(ac - \mathrm{RN}(bd)) = 1.11\ldots \times 10^{-16},$$
$$\mathrm{RN}(\mathrm{RN}(ac) - bd) = -1.11\ldots \times 10^{-16}.$$

Our goal here is to propose to replace these conventional schemes by ones which are proved to be always highly accurate, and to show that the price to pay for such numerical quality is, in practice, reasonably low. Specifically, we have implemented in GCC two alternative complex multiplication algorithms, CMulCHT and CMulKahan, capable of producing both the real and imaginary parts with high relative accuracy. These algorithms rely on the use of FMA instructions for computing efficiently (some of) the rounding errors of the products ac, bd, ad, bc, and then refining the accuracy of $ac - bd$ and $ad + bc$, as proposed by Cornea, Harrison, Tang [2] and Kahan [3]. Although CMulCHT and CMulKahan require 2.3x and 2x more floating-point operations than their conventional counterparts (without or with an FMA, respectively), we show that for typical applications like complex FFTs on ARM Cortex-A53 and -A57, the average execution-time overheads range from 1.04x to 1.17x only.

The rest of the paper is organized as follows. After some reminders about floating-point arithmetic in Section II, we describe algorithms CMulCHT and CMulKahan in Section III, together with their main numerical features. The practical performances of these algorithms are demonstrated and analyzed in Section IV. We conclude in Section V with several extensions that this preliminary study suggests.

II. Background

Hereafter, inputs and computed values will be floating-point numbers in base 2 and precision p, that is, elements of the set

$$\mathbb{F} = \{0\} \cup \{M \cdot 2^E \ : \ M, E \in \mathbb{Z}, \ \ 2^{p-1} \leqslant |M| < 2^p\}.$$

For simplicity, we shall ignore exceptions like underflow and overflow by leaving the exponent range unbounded.

We write RN to denote *round to nearest even*, which maps any $t \in \mathbb{R}$ to the unique $f \in \mathbb{F}$ satisfying the following properties: f is nearest t and, in case of a tie, its integer significand M is even. This corresponds to the rounding-direction attribute *roundTiesToEven* of the IEEE 754 standard [1] and is the default way of rounding in many applications.

We assume floating-point arithmetic over $\mathbb{F}$, correctly rounded according to RN. This means that for $a, b \in \mathbb{F}$ and $\mathrm{op} \in \{+, -, \times, /\}$, instead of returning the exact result $r = a \,\mathrm{op}\, b$, we return the floating-point number $\widehat{r}$ defined by

$$\widehat{r} := \mathrm{RN}(a \,\mathrm{op}\, b).$$

Each basic operation is thus performed with high relative accuracy. Indeed, it is well known that the exact result r and its rounded value $\widehat{r}$ are related as follows (see [4, p. 232]):

$$\widehat{r} = (a \,\mathrm{op}\, b)(1+\delta), \qquad |\delta| < 2^{-p} =: u.$$

This implies that the relative error $|\widehat{r}-r|/|r|$ is upper bounded by the quantity u, called *unit roundoff*, independently of the values of a and b. For the usual IEEE 754 formats, u is tiny: $u = 2^{-24} \approx 5.9 \times 10^{-8}$ for single precision, and $u = 2^{-53} \approx 1.1 \times 10^{-16}$ for double precision.

Similarly, when an FMA is used to evaluate $ab+c$, we have

$$\widehat{r} := \mathrm{RN}(ab+c) = (ab+c)(1+\delta), \qquad |\delta| < u.$$

Furthermore, since for $a, b \in \mathbb{F}$ the error $e := ab - \mathrm{RN}(ab)$ is itself in $\mathbb{F}$ (see for example [5]), we see that a correctly-rounded product and its exact error term can be obtained in just two floating-point operations as follows:

$$c := \mathrm{RN}(ab), \quad e := \mathrm{RN}(ab - c).$$

This is the key property underlying algorithms CMULCHT and CMULKAHAN (which are detailed in the next section together with the conventional ones).

III. Complex multiplication algorithms

We describe here the four algorithms we consider for our experiments and recall their main accuracy properties.

A. Algorithm CMUL

This algorithm implements the classical scheme for complex multiplication, which evaluates the real part $R = ac - bd$ and the imaginary part $I = ad + bc$ as follows:

$$\widehat{R} := \mathrm{RN}\big(\mathrm{RN}(ac) - \mathrm{RN}(bd)\big);$$
$$\widehat{I} := \mathrm{RN}\big(\mathrm{RN}(ad) + \mathrm{RN}(bc)\big);$$

Writing $z = R + iI$ and $\widehat{z} = \widehat{R} + i\widehat{I}$, it was shown in [6] that the *normwise* relative error $|\widehat{z} - z|/|z|$ is at most $\sqrt{5}u$ and thus always tiny. However, we have seen in introduction that the *componentwise* relative error, defined as the maximum of $|\widehat{R} - R|/|R|$ and $|\widehat{I} - I|/|I|$, can be 1, which means that all the digits of $\widehat{R}$ or $\widehat{I}$ can be wrong.

B. Algorithm CMULFMA

This is the algorithm obtained by using the FMA operation in the obvious way:

$$\widehat{R} := \mathrm{RN}\big(ac - \mathrm{RN}(bd)\big);$$
$$\widehat{I} := \mathrm{RN}\big(ad + \mathrm{RN}(bc)\big);$$

Of course, three variants of this scheme are possible, depending on the products to which RN is applied. But without a priori knowledge of a, b, c, d and due to the symmetry properties of $\mathbb{F}$ and RN, their worst-case error properties are the same. Consequently, we have chosen to implement only the scheme shown above, where RN is applied to bd and bc.

Algorithm CMULFMA uses fewer floating-point operations than CMUL, but its worst-case numerical behavior is essentially the same: its normwise relative error is tightly bounded by $2u$ (as shown in [7]), while its componentwise relative error can be 1 or more (as seen in introduction). Again, this implies that $\widehat{R}$ or $\widehat{I}$ can be totally inaccurate.

C. Algorithm CMULCHT

This algorithm is the accurate counterpart of CMUL and consists of two calls to the CHT algorithm from [2]:

$$\begin{aligned}
p_1 &:= \mathrm{RN}(ac); & p_2 &:= \mathrm{RN}(bd);\\
e_1 &:= \mathrm{RN}(ac - p_1); & e_2 &:= \mathrm{RN}(bd - p_2);\\
\widehat{R} &:= \mathrm{RN}\big(\mathrm{RN}(p_1 - p_2) + \mathrm{RN}(e_1 - e_2)\big);\\
p_3 &:= \mathrm{RN}(ad); & p_4 &:= \mathrm{RN}(bc);\\
e_3 &:= \mathrm{RN}(ad - p_3); & e_4 &:= \mathrm{RN}(bc - p_4);\\
\widehat{I} &:= \mathrm{RN}\big(\mathrm{RN}(p_3 + p_4) + \mathrm{RN}(e_3 + e_4)\big);
\end{aligned}$$

The rounding error analysis of the CHT algorithm implies that $|\widehat{R} - R| \leqslant 2u|R|$ and $|\widehat{I} - I| \leqslant 2u|I|$; see [8]. Hence both $\widehat{R}$ and $\widehat{I}$ are computed with high relative accuracy.

D. Algorithm CMULKAHAN

If instead of CHT we use Kahan's technique [3], we obtain the following accurate counterpart of CMULFMA:

$$\begin{aligned}
p_1 &:= \mathrm{RN}(ac);\\
\widehat{R} &:= \mathrm{RN}\big(\mathrm{RN}(p_1 - bd) + \mathrm{RN}(ac - p_1)\big);\\
p_3 &:= \mathrm{RN}(ad);\\
\widehat{I} &:= \mathrm{RN}\big(\mathrm{RN}(p_3 + bc) + \mathrm{RN}(ad - p_3)\big);
\end{aligned}$$

Again, three variants could be considered, depending on which products are chosen to set up p_1 and p_3. Furthermore, it follows from [9] that the relative errors of both $\widehat{R}$ and $\widehat{I}$ are at most $2u$.

IV. Experimental results

In this section we first describe the experimental environments we use both in terms of hardware and software. Then we illustrate the performance of our alternative complex multiplication algorithms through the case study of some Fast Fourier Transform (FFT) code. Performance measurements are provided by hardware counters by means of GCC's function instrumentation facility and the PAPI library [10].

A. Experimental environment description

We focus on ARMv8 hardware such as Cortex-A53 (CA53) and Cortex-A57 (CA57), in the 64-bit AArch64 execution state. The CA53 is an 8-stage (up to 10) pipelined processor with 2-way superscalar, in-order execution pipeline. The CA57 is a 15-stage (up to 24) pipelined processor with 3-way superscalar, out-of-order execution pipeline. Since there is no

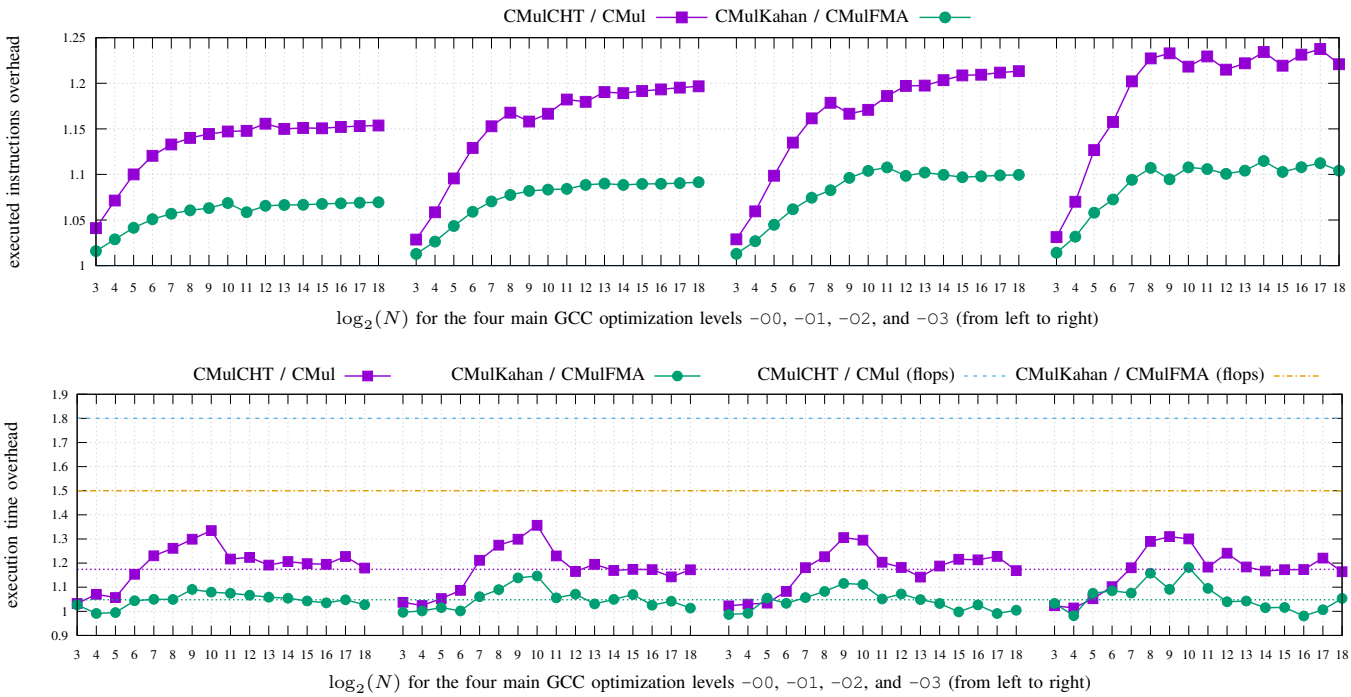

Fig. 1. **Overheads** in terms of **numbers of executed instructions** and **execution time** implied by CMULCHT and CMULKAHAN in double precision on a Cortex-A57 processor (for the four main optimization levels of GCC and for FFT of size $N \in \{2^3, 2^4, \ldots, 2^{18}\}$). In the bottom figure, dotted horizontal lines show the average values of execution time overheads, and dashed horizontal lines show the theoretical arithmetic cost overheads.

publicly available ARM documentation detailing these micro-architectures, we consider that the CA53 and CA57 follow the micro-architecture of their predecessors, the Cortex-A7 and Cortex-A15, as alleged in [11, p. 476]. We use an ARM Juno r1 development board to perform our measurements. On this board, the CA53 (revision r0p3) runs at 850 MHz and has two L1 caches of 32 KB and one L2 cache of 1 MB. The CA57 (revision r0p1) runs at 1.1 GHz, has one L1 instruction cache of 48 KB, one L1 data cache of 32 KB, and one L2 cache of 2 MB. Moreover, the board gets 7.8 GB of DDR3.

The four algorithms of Section III have been implemented in GCC and a new option has been introduced (`-fcx-lr-select=cht|default|fma|kahan`), which allows us to select one of them. The main contribution concerns the complex multiplication expansion. It occurs when GCC converts the high-level language representation (for example, _Complex data type in C) into its low-level intermediate representation (GIMPLE).

The new option is valid in C and Fortran, and is only available when the flag `-fcx-limited-range` is defined. This flag allows the user to omit checking that the result of a complex multiplication is NaN + iNaN. This flag authorizes the elimination of the special case checking otherwise required by the C99 standard [12, Appendix G, 5.1], leading to the emission of efficient in-line code instead of a call to a library function. Note that for all our experiments we shall compare alternative complex multiplications to the default ones in this "limited range" scope, but we could also apply them in a strict C99 context. The rationale behind this choice is to be as generic as possible and, therefore, valid for different versions of different languages.

The GCC compiler considered here is built from the sources provided by Linaro [13] (from the GIT's tag `linaro-6.2-2016.11-rc2`). It is built with glibc 2.24, binutils 2.27, gmp 6.1.1, mpfr 3.1.5, mpc 1.0.3, and isl 0.16.1. Moreover, we use the C language to write the programs used in our experiments.

B. Fast Fourier Transform case study

We consider FFT as in Algorithm 1, which computes in $\frac{3}{2}N\log_2(N)$ complex operations a discrete Fourier transform of size $N = 2^n$ defined by

$$X(j) = \sum_{k=0}^{N-1} A(k)\omega^{jk}, \quad j = 0, 1, \ldots, N-1,$$

where the $A(k)$ are given complex numbers and $\omega = e^{\frac{-2\pi i}{N}}$; see [14]. Assuming that the complex exponential terms ω^{jk} are precomputed, this algorithm uses $N\log_2(N)$ complex additions and $\frac{1}{2}N\log_2(N)$ complex multiplications.

Figure 1 shows the performance overheads due to using CMULCHT and CMULKAHAN instead of CMUL and CMULFMA, for double precision and on the CA57. Measurements are done for $N \in \{2^3, 2^4, \ldots, 2^{18}\}$ and the four main optimization levels `-O0`, `-O1`, `-O2`, `-O3` of GCC.

The top plots in Figure 1 show, not surprisingly, that the larger N is, the more the *instruction number* overhead grows. Although small, these overheads increase according to the

optimization level. Therefore, the more code size is reduced thanks to compiler optimizations, the more the extra floating-point operations used by the accurate algorithms become apparent. For example, in `-O3` and for $N = 2^{10}$, the number of executed instructions for CMUL is 202,074, while CMULCHT executes 246,173 of them.

Algorithm 1 Pseudocode for the Cooley–Tukey algorithm.

procedure FFT(A, N, s)
  if $N = 1$ **then**
    $X(0) \leftarrow A(0)$
  else
    $X(0, \ldots, N/2-1) \leftarrow \text{FFT}(A, N/2, 2s)$
    $X(N/2, \ldots, N-1) \leftarrow \text{FFT}(A+s, N/2, 2s)$
    for $k = 0$ to $N/2-1$ **do**
      $t \leftarrow X(k)$
      $X(k) \leftarrow t + e^{-2\pi ik/N} \cdot X(k+N/2)$
      $X(k+N/2) \leftarrow t - e^{-2\pi ik/N} \cdot X(k+N/2)$
    end for
  end if
  return $X(0, \ldots, N-1)$
end procedure

The bottom plots in Figure 1 show that, on the other hand, the *execution time* overheads are quite reasonable. On average (in the sense of the geometric mean), CMULKAHAN is only 1.04x slower than CMULFMA, and CMULCHT is only 1.17x slower than CMUL. We remark that overheads constantly grow until $2^8 \leqslant N \leqslant 2^{10}$ and then do not vary much for larger values of N. Indeed, for $N \in [2^8; 2^{10}]$, the data have size ranging from 32 KB to 128 KB and do not fit in L1 cache anymore. Memory transfer costs, for larger values of N, compensate part of the introduced overhead. We also remark that these overheads observed in practice are smaller than those suggested by mere flop counts. For example, the theoretical overhead (in flops) of CMULCHT over CMUL is 1.8x, while measurements expose overheads ranging from 0.98x[1] to 1.35x. This is mainly due to the instruction-level parallelism exploited by the pipelines of the CA57.

For this case study, the results we have obtained for single precision and the CA53 are similar. Focusing on the execution-time overhead, a global summary is given in the table below. On average, we see that the overhead due to using CMULCHT is of about 17%, while it is less than 10% for CMULKAHAN.

TABLE I
AVERAGE EXECUTION-TIME OVERHEAD (GEOMETRIC MEAN OVER ALL OPTIMIZATION LEVELS AND ALL VALUES OF N).

overhead	CMULCHT / CMUL		CMULKAHAN / CMULFMA	
target	Cortex-A53	Cortex-A57	Cortex-A53	Cortex-A57
single	1.1589	1.1660	1.0952	1.0759
double	1.1702	1.1740	1.0543	1.0480

V. CONCLUDING REMARKS

This work in progress shows encouraging results concerning the overhead of highly accurate complex floating-point multiplication. Accuracy improvement overheads are reasonable thanks to exploitation of the FMA and to the code generation that GCC provides. Overhead minimization is also partially due to the instruction-level parallelism provided by the multi-pipelined execution units of the CA53 and CA57. The FFT case study illustrates that computing with more accuracy only slows down the execution time by a factor from 1.04x to 1.17x on average.

In a future work, we will extend this study to other complex arithmetic operations such as division. We also plan to reduce the overheads by providing still accurate but faster specialized algorithms, as for example the multiplication of a complex number by its conjugate (which can be done faster than the original CMUL itself). Since all our algorithms are implemented in GCC, we plan to study their performances on other architectures as well, notably the x86_64. Finally, these improved multiplication algorithms are available as a patch on top of the GCC compiler, along with specific test cases and documentation. We intend to propose these changes to be integrated upstream in the near future.

[1] Although very close to 1, this overhead must be considered $\geqslant 1$. Overheads less than 1 are due to measurement inaccuracies.

ACKNOWLEDGMENT

This research was partly supported by *Programme Nano 2017*. Access to the Juno board used in this paper was gracefully granted by Laboratoire LIG and Inria.

REFERENCES

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754-2008*. New York: IEEE Computer Society, 2008.
[2] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Hillsboro, OR, USA: Intel Press, 2002.
[3] W. Kahan, "Matlab's loss is nobody's gain," 1998–2004, available at https://people.eecs.berkeley.edu/~wkahan/MxMulEps.pdf.
[4] D. E. Knuth, *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1998.
[5] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, pp. 224–242, 1971.
[6] R. Brent, C. Percival, and P. Zimmermann, "Error bounds on complex floating-point multiplication," *Math. Comp.*, vol. 76, pp. 1469–1481, 2007.
[7] C.-P. Jeannerod, P. Kornerup, N. Louvet, and J.-M. Muller, "Error bounds on complex floating-point multiplication with an FMA," *Math. Comp.*, vol. 86, pp. 881–898, 2017.
[8] C.-P. Jeannerod, "A radix-independent error analysis of the Cornea-Harrison-Tang method," *ACM Trans. Math. Software*, vol. 42, no. 3, pp. 19:1–19:20, 2016.
[9] C.-P. Jeannerod, N. Louvet, and J.-M. Muller, "Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants," *Math. Comp.*, vol. 82, pp. 2245–2264, 2013.
[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, pp. 189–204, 2000.
[11] S. Harris and D. Harris, *Digital Design and Computer Architecture: ARM Edition*. San Francisco, CA, USA: Morgan Kaufmann, 2015.
[12] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011.
[13] Linaro Engineering Organization, "GNU Compiler Collection's Git repository," available at https://git.linaro.org/toolchain/gcc.git, in development since 2010, cloned from [15].
[14] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comp.*, vol. 19, pp. 297–301, 1965.
[15] GNU Project, "GNU Compiler Collection's Git repository," available at https://gcc.gnu.org/git, in development since 1987.