Scalar Fused Multiply-Add Instructions Produce Floating-Point Matrix Arithmetic Provably Accurate to the Penultimate Digit

YVES NIEVERGELT
Eastern Washington University

Combined with doubly compensated summation, scalar fused multiply-add instructions redefine the concept of floating-point arithmetic, because they allow for the computation of sums of real or complex matrix products accurate to the penultimate digit. Particular cases include complex arithmetic, dot products, cross products, residuals of linear systems, determinants of small matrices, discriminants of quadratic, cubic, or quartic equations, and polynomials.

Categories and Subject Descriptors: B.2.0 [**Arithmetic and Logic Structures**]: General; B.7.1 [**Integrated Circuits**]: Types and design styles—*algorithms implemented in hardware*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.0 [**Computer Systems Organization**]: General—*instruction set design*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computations on matrices*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic, error analysis, multiple precision arithmetic, numerical algorithms*; G.4 [**Mathematical Software**]: *algorithm design and analysis, certification and testing, reliability and robustness*

General Terms: Algorithms, Design, Languages, Standardization, Theory

Additional Key Words and Phrases: Doubly compensated summation, floating-point arithmetic, fused multiply-add instruction, matrix arithmetic, provable accuracy, rounding error

1. INTRODUCTION

A decade ago, David Goldberg presented the IEEE 754-1985 Standard in "What every computer scientist should know about floating-point arithmetic" [Goldberg 1991]. Since then, a new floating-point operation—called the "fused multiply-add" or "floating-point multiply and accumulate" and its variant the "fused multiply-subtract"—has been planned or already implemented by several manufacturers: the instructions FMPYADD and FMPYSUB on

This work was supported in part by a Faculty Research Grant from Eastern Washington University, and completed in part at the University of Washington in Seattle.
Author's address: Department of Mathematics, Eastern Washington University, 216 Kingston Hall, Cheney, WA 99004-2418.

Hewlett-Packard's PA-8000 and PA-8200 PA-RISC [Scott et al. 1997], fma on IBM's RISC System/6000 [Hokenek and Montoye 1990, p. 24; Markstein 1990; Montoye et al. 1990; Oehler and Groves 1990, p. 24; Warren 1990], IBM's System/390 [Abbott et al. 1999], and IBM's POWER3 [O'Connell and White 2000], fma on Intel's Itanium [Intel Corporation 2001], and the vectorized vmaddfp on Motorola's AltiVec [Apple Computer, Inc. 2002] processors. This article shows that, rather than being merely another floating-point operation, *the "fused multiply-add" instruction redefines the whole concept of floating-point arithmetic*.

To this end, the present section reviews the notation for floating-point arithmetic necessary to describe "fused multiply-add" instructions. Besides exact integer arithmetic in a limited range, a typical digital computer works with neither rational nor real or complex numbers, but, instead, with a finite set F of "floating-point" numbers, which depend on the particular computer. Specifically, a *floating-point number* f with m significant digits in base B can be written in the form $f = (-1)^s * B^E * b_0.b_1 \ldots b_{m-1}$, which can be defined by

$$f = (-1)^s * B^E * b_0.b_1 \cdots b_{m-1} := (-1)^s * B^E * \sum_{k=0}^{m-1} b_k B^{-k}. \tag{1}$$

Here $s \in \{0, 1\}$ is the *sign bit*, E is an integer *exponent* in a range $E_{\min} \leq E \leq E_{\max}$, and $b_0, \ldots, b_{m-1} \in \{0, \ldots, B-1\}$ are the *significant digits* of the *mantissa* $(-1)^s * b_0.b_1 \ldots b_{m-1}$. The following abbreviations will prove convenient:

$$\operatorname{sign}(f) := (-1)^s, \tag{2}$$

$$\mathtt{E}(f) := E, \tag{3}$$

$$\mathtt{M}(f) := (-1)^s * b_0.b_1 \cdots b_{m-1}. \tag{4}$$

A measure of precision called a *Unit in the Last Place* (ulp) is defined for each $f \in \mathtt{F}$, $f \neq 0$, by Higham [1996, p. 43], Overton [2001, p. 14], and Priest [1992, p. 7]

$$\mathtt{ulp}(f) := B^{\mathtt{E}(f)} * 0.0 \cdots 01 = B^{\mathtt{E}(f)+1-m}. \tag{5}$$

The unit ulp will be used here only to measure the size of inaccuracies, in examples (1) and (3). Depending on the computer or the exposition, the exponent and the fractional point can be shifted in opposite directions [Higham 1996, p. 40; Overton 2001, p. 12; Wilkinson 1963, p. 2], for instance,

$$B^E * b_0.b_1 \cdots b_{m-2}b_{m-1} = B^{E+1} * 0.b_0b_1 \cdots b_{m-2}b_{m-1}. \tag{6}$$

While IEEE standards require complying computers and calculators to accept data and deliver results in either binary ($B = 2$) [Institute of Electrical and Electronic Engineers 1985] or decimal ($B = 10$) [Institute of Electrical and Electronic Engineers 1987] format, to take advantage of simpler circuits some current machines perform computations with other bases, for instance, hexadecimal ($B = 16$) [Abbott et al. 1999; Montoye et al. 1990], as also in other computers listed in Table I.

A *floating-point representation* is a function float : $\mathbb{R} \to \mathtt{F}$ that approximates each real number $x \in \mathbb{R}$ by a floating-point number $\mathtt{float}(x) \in \mathtt{F}$. The

Table I. Fixed and Floating-Point Parameters

SYSTEM	PRECISION	B	m	$E_{\min}$	$E_{\max}$
Burroughs 5500	single	8	13	−51	77
Control Data 6600	single	2	48	−975	1071
Control Data Cyber 70	single	2	48	−1022	1022
Control Data Cyber 70	double	2	96	−1022	1022
Cray-1	single	2	48	−8192	8191
Cray-1	double	2	96	−8192	8191
DEC VAX G format	double	2	53	−1023	1023
DEC VAX D format	double	2	56	−127	127
DEUCE	fixed	2	48	1	1
HP-15C	single	10	10	−99	99
HP-15C	arrays	10	13	−99	99
HP 28 & 48 series	single	10	12	−499	499
HP 28 & 48 series	matrix	10	15	−499	499
IBM 650	single	10	8	−50	49
IBM 3090	single	16	6	−64	63
IBM 3090	double	16	14	−64	63
IBM 3090	extended	16	28	−64	63
IBM 7090	single	2	27	−128	127
IBM System/360	single	16	6	−64	63
IBM System/360	double	16	14	−64	63
IBM System/390	short	2	24	−126	127
IBM System/390	long	2	53	−1022	1023
IBM System/390	extended	2	113	−16382	16383
Intel's Itanium	extended	2	64	−32766	32767
IEEE Standard 754-1985	single	2	24	−125	128
IEEE Standard 754-1985	double	2	53	−1021	1024
SEAC	fixed	2	50	1	1
TI-1706 II	single	10	8	−7	7

representation `float` is *faithful* if and only if `float`(x) is either the largest element of `F` that does not exceed x (the greatest lower bound for x in `F`), in which case `float` *truncates* x, or the smallest element of `F` that is at least as large as x (the least upper bound for x in `F`), in which case `float` *rounds up* x [Dekker 1971, p. 226]. In either case, x and `float`(x) differ from each other by less than one `ulp[float`(x)`]`, because if $x \in$ `F`, then $x =$ `float`(x).

A *properly rounding* faithful floating-point representation approximates x by one of the at most two floating-point numbers `float`(x) closest to x. Thus, `float` *rounds to "the" nearest* floating-point number, so that x and `float`(x) differ from each other by at most `ulp[float`(x)`]`$/2$ (or `ulp[float`(x)`]`$/(2B)$ if `float`(x) is a power of B and $|x| <$ `float`(x)). The choice of rounding—up or down—will not matter for the present purposes [Dekker 1971, p. 227].

A floating-point representation is *monotonic* if and only if `float`$(v) \leq$ `float`(w) for all real $v \leq w$. For instance, each of the four IEEE standard rounding modes, in particular every properly rounding floating-point representation, is also monotonic. As just defined, however, a merely faithful floating-point representation need not be monotonic, because it could round every real x between any two floating-point numbers $p < q$ randomly to p or q, so that $p =$ `float`$(w) <$ `float`$(v) = q$ for some real $p < v < w < q$. Monotonicity will only be needed as indicated.

If $|x|$ exceeds the largest magnitude representable in $\mathtt{F}$, then `float` represents x by an exception flag called an *overflow*. Similarly, if $x \neq 0$ and $|x|$ is smaller than the smallest positive magnitude representable in $\mathtt{F}$, then `float` tags x with an exception flag called an *underflow*, and may—but need not—still include a few significant digits in the mantissa in a representation called *gradual underflow*.

The floating-point arithmetic operations $\oplus, \ominus, \circledast, \oslash$ are *faithful* or *properly rounding* provided that so is `float` and if $r \circledcirc s = \mathtt{float}(r \circ s)$ for every operation $\circ \in \{+, -, *, /\}$ and all floating-point numbers $r, s \in \mathtt{F}$ (except for divisions by 0) [Dekker 1971, p. 227], with corresponding exceptions if $r \circ s$ overflows or underflows.

Despite the accuracy of each operation, the result of a sequence of as few as two properly rounding floating-point operations can suffer from inaccuracies in *all* its significant digits. Such extreme inaccuracies can occur, for instance, in the computation of scalar affine operations of the type $f(a, x; y) := (a * x) + y$ by the sequence of two floating-point operations $(a \circledast x) \oplus y$, as in Example 1.

Example 1. With $m := 3$ binary digits in base $B := 2$, let $a := 1.10$, $x := 1.10$, and $y := -10.1$. From these values $(a * x) + y = -0.01$, but floating-point arithmetic gives -0.10, thus with a relative error equal to 100% of the exact result:

$$\begin{aligned} a \circledast x &= (1.10) \circledast (1.10) &= \mathtt{float}(10.01) &= 10.0; \\ (a \circledast x) \oplus y &= (10.0) \oplus (-10.1) &= \mathtt{float}(-0.10) &= -0.10. \end{aligned} \tag{7}$$

Similar errors can occur with any number m of significant digits. For instance, if

$$\begin{aligned} a &:= 1 + 2^{1-m} &= 1.00 \cdots 001, \\ x &:= 2 - 2^{1-m} &= 1.11 \cdots 111, \\ y &:= 2^{1-m} - 2 &= -1.11 \cdots 111, \end{aligned} \tag{8}$$

then $(a * x) + y = 2^{2-m} - 2^{2-2m}$, but floating-point arithmetic gives 2^{1-m}:

$$a \circledast x = \mathtt{float}(2 + 2^{1-m} - 2^{2-2m}) \tag{9}$$

$$= \mathtt{float}(10.00 \cdots 00\,11 \cdots 11) \tag{10}$$

$$= 10.00 \cdots 00 \tag{11}$$

$$= 2, \tag{12}$$

$$(a \circledast x) \oplus y = \mathtt{float}[2 + (2^{1-m} - 2)] \tag{13}$$

$$= 2^{1-m}. \tag{14}$$

The cummulative rounding error is $(2^{2-m} - 2^{2-2m}) - 2^{1-m} = 2^{1-m} - 2^{2-2m}$, which amounts to about one Unit in the Last Place (`ulp`) of the *Operand with the Largest Magnitude* (`OLM`): here $\mathtt{ulp}(\mathtt{OLM}) = 2^{1-m}$ in $|x|$ or $|y|$.

The fact that as few as two consecutive floating-point operations can lead to completely erroneous results has caused concerns about the potential for disastrous errors, and has shown the need for improvements in accuracy and reliability:

> This not only is a challenging intellectual exercise but also is of crucial importance to industry, where large sums of money or even issues of human safety may hinge upon complicated numerical calculations. [. . .] There are so many examples of deceptively simple computations going haywire that one's confidence in the reliability of computations could be significantly eroded. —[Duff 1996].

As an example specific to applications of mathematics and physics in aerospace engineering, Kahan has documented how floating-point rounding errors once led to erroneous conclusions about the lift of aircraft wings [Kahan 1972, p. 1228]. Besides new engineering design, floating-point computations also occur in simulations through computer graphics, manufacturing, and maintenance of existing aircraft, all of which rely on geometric algorithms, usually with stunning success [Grandine 2000]. Nevertheless, the potential for errors from floating-point arithmetic cannot yet be dismissed:

> My suspicion is that the vast majority of code for geometric algorithms is faulty for these reasons. —[O'Rourke 1993, p, 91].

Indeed, the scalar affine operation $(a * x) + y$ demonstrated in example 1 is one of the elementary operations in the *Basic Linear Algebra Subprograms* (BLAS) [Coleman and Van Loan 1988, Ch. 2] that occur repeatedly within such fundamental algorithms as Horner's polynomial evaluation, Gaussian elimination, matrix algebra, and iterative methods, for instance, singular-value decompositions [Golub and Van Loan 1989, p. 4]. Consequently, Example 1 also shows that the highest provable accuracy for such algorithms must allow for cummulative rounding errors of about one unit in the last place of the operands with the largest magnitude, which can be larger than the exact result. This level of (in)accuracy is the one proved for the state-of-the-art matrix algorithms, from complex arithmetic [Hewlett-Packard Co. 1984, p. 75] to real dot products, matrix multiplication [Higham 1996, p. 76], and singular-value decompositions [Demmel and Kahan 1990].

Greater accuracy can result from the use of multiple-precision software packages. However, they require the explicit use of the base B and the number m of significant digits [Dekker 1971, p. 234, eq. (6.2)], which restricts their portability, and they can make computations impractically slow [Overton 2001, p. 93]. Multiple-precision facilities embedded in higher-level languages, for instance, `C` and `C-XSC` [Lawo 1993], `FORTRAN` and `ACRITH-XSC` [Walter 1993], or `PASCAL` and `PASCAL-XSC` [Hammer et al. 1993], require special compilers or hardware that simulate or implement, respectively, a fixed-point adder and accumulator of length greater than $|2E_{\max} - 2E_{\min}| + 2m$ [Bleher 2001, p. 100; Bohlender and Grüner 1983, p. 259; Kulisch and Miranker 1986, p. 26]. For Intel's Itanium 17-bit exponent range and 64-bit mantissa, the floating-point accumulator would have to accommodate more than $|2E_{\max} - 2E_{\min}| + 2m = 131\,066 + 128 = 131\,194$ binary bits!

To increase the accuracy without slowing down computations or waiting for inordinately long floating-point facilities, manufacturers are beginning to provide "properly rounding affine operations" called *fused multiply-add* (`fma`) instructions [Higham 1996, p. 60; Overton 2001, p. 50]. For instance, Intel Corporation's Itanium chip contains 82-bit floating-point registers (128 of them), each with a 64-bit significand field for the mantissa, a 17-bit exponent field, and 1 sign bit. These registers can be used for a fused multiply-add instruction defined as follows (where `FPSR` denotes the Floating-Point Status Register [Intel Corporation 2001]):

Definition 1. `fma` Floating-point Multiply Add Instruction

(qp) fma.pc.sf f1 = f3, f4, f2

> The product of floating-point register (FR) `f3` and FR `f4` is computed to infinite precision and then FR `f2` is added to this product, again in infinite precision. The resulting value is then rounded to the precision indicated by `pc` (and possibly `FPSR.sf.pc` and `FPSR.sf.wrc`) using the rounding mode specified by `FPSR.sf.rc`. The rounded result is placed in FR `f1`. —[Intel Corporation 2001].

In other words, the fused multiply-add instruction `fma` computes

$$\texttt{fma}(a, x; y) := \texttt{float}[(a * x) + y] \tag{15}$$

as a 64-bit mantissa floating-point number closest to the real number $(a*x)+y$.

The purpose of the present work is to demonstrate how such *scalar* fused multiply-add instructions can be used to produce provably accurate affine *matrix* operations without resorting to multiple-precision software. To this end, Section 2 shows that the fused multiply-add instruction by itself does not suffice to produce accurate matrix arithmetic, thus documenting the need for its combination with other operations in Sections 5 and 6, after Sections 3 and 4 have established ways to avoid needless overflows or underflows.

2. NONMONOTONICITY OF FUSED MULTIPLY-ADD OPERATIONS

Operations such as $(r * s) - (u * v)$ occur in the discriminant of quadratic equations, and in the determinant of linear systems. If $(r*s)-(u*v) > 0$, then $r*s > u*v$, whence $r \circledast s \geq u \circledast v$ by the monotonicity of $\circledast$, and hence $(r \circledast s) \ominus (u \circledast v) \geq 0$ by the monotonicity of $\ominus$ [Higham 1996, p. 60]. In contrast, for the fused multiply-add operation, there exist floating-point numbers $r, s, u, v \in \mathtt{F}$ such that $(r*s)-(u*v) > 0$ but $\texttt{fma}[r, s; -(u \circledast v)] < 0$, as in Example 2. Because of such defects, the `fma` has been considered a "mixed blessing" [Overton 2001, p. 51].

Example 2. For each floating-point system with base B and m significant digits $b_0.b_1 \cdots b_{m-1}$, let

$$h := B^{-(2+\lceil (m-1)/2 \rceil)}, \tag{16}$$

$$k := B^{-(1+\lceil (m-1)/2 \rceil)}, \tag{17}$$

$$r := 1 - h, \tag{18}$$

$$s := 1 + h, \tag{19}$$
$$u := 1 - k, \tag{20}$$
$$v := 1 + k. \tag{21}$$

Thus,

$$(r * s) - (u * v) = (1 - h^2) - (1 - k^2) = k^2 - h^2 > 0, \tag{22}$$

but

$$u * v < r * s < u \circledast v = 1 = r \circledast s, \tag{23}$$

whence

$$\begin{aligned} \texttt{fma}[r, s; -(u \circledast v)] &= \texttt{float}[(r * s) - (u \circledast v)] = \texttt{float}[(1 - h^2) - 1] & (24)\\ &= \texttt{float}[-h^2] = -B^{-2(2+\lceil (m-1)/2 \rceil)} < 0. & (25)\end{aligned}$$

Nevertheless, $(u * v) - (r * s) < 0$ and also

$$\begin{aligned} \texttt{fma}[u, v; -(r \circledast s)] &= \texttt{float}[(u * v) - (r \circledast s)] = \texttt{float}[(1 - k^2) - 1] & (26)\\ &= \texttt{float}[-k^2] = -B^{-2(1+\lceil (m-1)/2 \rceil)} < 0. & (27)\end{aligned}$$

However, Theorem 1 guarantees that at least one order of the operands (which may depend on the operands) leads to the correct sign or to zero, but not to the wrong sign again.

THEOREM 1. *For each monotonic faithful floating-point arithmetic, if* $(r * s) - (u * v) > 0$ *but* $\texttt{fma}[r, s; -(u \circledast v)] < 0$*, then* $\texttt{fma}[u, v; -(r \circledast s)] \le 0$.

PROOF. If $(r * s) - (u * v) > 0$, then $u * v < r * s$, whence $u \circledast v \le r \circledast s$ by monotonicity of $\circledast$.

If also $\texttt{fma}[r, s; -(u \circledast v)] < 0$, then also $r * s < u \circledast v$, by contraposition and monotonicity of `float`, because $\texttt{fma}[r, s; -(u \circledast v)] = \texttt{float}[(r * s) - (u \circledast v)]$. From $r * s < u \circledast v$ now follows $r \circledast s \le u \circledast v$ by monotonicity of $\circledast$.

Consequently, $r \circledast s = u \circledast v$, so that $u * v < r * s \le r \circledast s = u \circledast v$.

Therefore, $\texttt{fma}[u, v; -(r \circledast s)] = \texttt{float}[(u * v) - (r \circledast s)] \le 0$. □

Nevertheless, the fused multiply-add instruction *with* an accurate algorithm to add floating-point numbers will produce an accurate matrix arithmetic. The first task consists in extracting the exponent and the mantissa.

3. EXTRACTION OF THE EXPONENT AND MATISSA

Some computer systems provide commands to extract and set the exponent and the mantissa of a floating-point number, for instance, `C` [Kernighan and Ritchie 1988, p. 251], `FORTRAN 90` [Higham 1996, p. 498], and the HP-28 and HP 48 series calculators [Hewlett-Packard Co. 1986, p. 270; 1987, p. 219; 1990, pp. 148–149]. Yet such languages as `BASIC` [Coan 1977, pp. 163–168], `Pascal` [Wilson and Addyman 1978, pp. 23–25], IEEE Standard arithmetics [Institute of Electrical and Electronic Engineers 1985, 1987], and other computing systems do not provide facilities to read the exponent and mantissa off floating-point numbers. With such systems, the extraction of the exponent or the mantissa can involve a combination of arithmetic and logical operations that depend on the encoding of floating-point numbers by the hardware [Sterbenz 1974, Sect. 4.4], and *some programs to this effect reportedly fail on some machines* [Higham 1996, pp. 497–498]. Moreover, if f is an unnormalized floating-point number, for instance, gradually underflowing, then $|\mathtt{M}(f)| < 1$ and hence the exponent $\mathtt{E}(f)$ can differ from $\lfloor \log_B |f| \rfloor$, so that additional tests still become necesary.

Nevertheless, systems that lack such functions and facilities are amenable to a variant of an algorithm mentioned by Knuth: to compute $\lfloor \log_2(N/M) \rfloor$ with $N \geq M$, double M until it exceeds N [Knuth 1998, p. 206, # 15, & p. 660, # 15]. Because the last doubling of M can overflow, however, Knuth's algorithm requires a modification. Specifically, the modification proposed here begins with $f_0 := f$ and then consists of two procedures for two mutually exclusive cases.

In the first case, if $f \geq 1$, then the first procedure consists in dividing f_j by B while $f_j \geq B$. Upon termination, $j = \lfloor \log_B(f) \rfloor$ is the exponent of f_0 in base B, and f_j is the mantissa of f_0, in the range $1 \leq f_j < B$.

For $0 < f < 1$, this first procedure need not apply to $1/f$, because if f does not underflow, then $1/f$ can still overflow. Indeed, some machines allow for the exponent E to lie within an asymmetric range where $E_{\max} < -E_{\min}$. For example, $-8192 \leq E \leq 8191$ on the Cray-1, $-50 \leq E \leq 49$ on the IBM 650, $-64 \leq E \leq 63$ on the IBM 3090 and IBM System/360, and $-128 \leq E \leq 127$ on the IBM 7090 [Forsythe 1970, p. 933; Higham 1996, p. 41]. Therefore, if $0 < f < 1$, then the second procedure proposed here divides f_j by $1/B$ while $f_j < 1$. Upon termination, f_j is the mantissa of f_0 in the range $1 \leq f_j < B$, and $-j = \lfloor \log_B(f) \rfloor < 0$.

The division by $1/B < 1$, rather than a multiplication by $B > 1$, allows the same procedure to handle fixed-point numbers normalized in the range $[1, B)$, which was the range for DEUCE [Wilkinson 1963, p. 34] and SEAC [Henrici 1982, p. 6]. Such fixed-point machines can represent $1/B$ as $0.10 \cdots 0$ but lack an additional digit to represent B as $10.0 \cdots 0$. This second procedure need not apply to $1/f < 1$ for $f > 1$, because some machines allow exponents to lie in an oppositely biased asymetric range where $-E_{\max} < E_{\min}$, for instance, $-51 \leq E \leq 77$ on the Burroughs 5500, $-975 \leq E \leq 1071$ on the Control Data 6600 [Forsythe 1970, p. 933], and more generally $-125 \leq E \leq 128$ and $-1021 \leq E \leq 1024$ in IEEE single and double precision [Higham 1996, p. 41].

Algorithm 1 computes floating-point exponents and mantissas as just described.

Algorithm 1 (Extracts the Exponent and the Mantissa).

```
DATA: $B$, $1/B$, $f \in \mathtt{F}$.
RESULTS: $E := \lfloor \log_B(|f|) \rfloor \in \{\mathtt{NaN}\} \cup \mathbb{Z}$,
$M := \mathtt{M}(f) \in \mathtt{F}$, $|M| \in \{0\} \cup [1, B)$.
START
If $f = 0$, then
    $M := f$ (preserves $+0$ and $-0$),
    $E :=$ NaN;
else (if $f \neq 0$)
    $M := |f|$,
    $E := 0$;
```

```
  if $M < 1$, then
    while $M < 1$ do
      $M := M/(1/B)$,
      $E := E - 1$;
    end while;
  else (if $M \geq 1$)
    while $M \geq B$ do
      $M := M/B$,
      $E := E + 1$;
    end while;
  end if;
  $M :=$ sign$(f)M$;
end if.
STOP.
```

4. SCALING ALGORITHM

There exist floating-point numbers $u, v \in \mathtt{F}$ such that the floating-point product $u \circledast v$ overflows or underflows, even though the mathematical product and difference $(u * v) - (u \circledast v)$ would otherwise be representable in $\mathtt{F}$. Heuristic scaling methods can help in avoiding such intermediate overflow and underflow, but such methods can depend on the base B and the range of exponents $[E_{\min}, E_{\max}]$ allowed by the floating-point hardware, as outlined in a different context by Sterbenz [1974, p. 251]. To avoid such needless overflows, Algorithm 2 avoids overflow (or underflow) for all data $u, v \in \mathtt{F}$ for which the result $\mathtt{fma}[u, v; -(u \circledast v)]$ does not overflow (or underflow), without any reference to the range of exponents.

Algorithm 2 (Avoids Overflow or Underflow of $u \circledast v$ in $\mathtt{fma}[u, v; -(u \circledast v)]$).

```
DATA: Two floating-point numbers $u, v \in \mathtt{F}$.
RESULT: A floating-point number $\mathtt{scaled\_fma}[u, v; -(u \circledast v)]$ or NaN.
START
If $u = 0$ or $v = 0$, then
  $\mathtt{scaled\_fma}[u, v; -(u \circledast v)] := 0$;
else (if $u \neq 0$ and $v \neq 0$)
  $r := \mathtt{M}(u)$,
  $s := \mathtt{M}(v)$,
  $\mathtt{scaled\_fma}[u, v; -(u \circledast v)] := \mathtt{fma}[r, s; -(r \circledast s)] \circledast B^{\mathtt{E}(u)} \circledast B^{\mathtt{E}(v)}$.
STOP.
```

In Algorithm 2, the absolute value of each mantissa lies in the range $[1, B)$, whence $|r \circledast s|$ lies in the range $[1, B^2)$, which neither overflows nor underflows in all current floating-point systems (and in the fixed-point systems in Table I). Consequently, $|(r * s) - (r \circledast s)| < B^2$ does not overflow, and it does not underflow in floating-point systems where $E_{\min} \leq -2m$, which is the case for all current floating-point systems.

For all floating-point numbers $u, v, w \in \mathtt{F}$, the specification of the fma operation in Definition 1 guarantees that if the mathematical result $(u * v) + w$ does not overflow (or does not underflow), then neither does $\mathtt{fma}[u, v; w]$, because it yields $\mathtt{float}[(u * v) + w]$. Nevertheless, there exist some machines—in particular, hand-held calculators—that perform a "pseudo fused multiply-add operation" by computing the intermediate operations $u \circledast v$ and then

$\texttt{pseudo\_fma}[u, v; w] := (u \circledast v) \oplus w$ in extended—but not infinite—precision. For such machines, Algorithm 3 avoid overflows and underflows in the intermediate operations.

Algorithm 3 (Avoids Overflow or Underflow in $\texttt{pseudo\_fma}[u, v; w]$**).**

```
DATA: Three floating-point numbers $u, v, w \in \mathtt{F}$.
RESULT: A floating-point number $\texttt{scaled\_pseudo\_fma}[u, v; w]$ or NaN.
START
If $u = 0$ or $v = 0$ or $w = 0$, then
  $\texttt{scaled\_pseudo\_fma}[u, v; w] := \texttt{pseudo\_fma}[u, v; w]$;
else (if $u \neq 0$ and $v \neq 0$ and $w \neq 0$)
  $U := \mathtt{E}(u)$,
  $r := \mathtt{M}(u)$,
  $V := \mathtt{E}(v)$,
  $s := \mathtt{M}(v)$,
  $W := \mathtt{E}(w)$,
  $t := \mathtt{M}(w)$,
  if $W > U + V$ (so that $U + V < W \leq E_{\max}$), then
    $P := \lfloor (W + U - V)/2 \rfloor$,
    $Q := \lceil (W + V - U)/2 \rceil$,
    $r := \mathtt{M}(u) \circledast B^{U-P}$,
    $s := \mathtt{M}(v) \circledast B^{V-Q}$,
    $t := \mathtt{M}(w) \circledast B^{W-(P+Q)}$,
    $\texttt{scaled\_pseudo\_fma}[u, v; w] := \texttt{pseudo\_fma}[r, s; t] \circledast B^{P+Q}$;
  else (if $\mathtt{E}(w) \leq \mathtt{E}(u) + \mathtt{E}(v)$)
    $r := \mathtt{M}(u)$,
    $s := \mathtt{M}(v)$,
    $t := \mathtt{M}(w) \circledast B^{W-(U+V)}$,
    $\texttt{scaled\_pseudo\_fma}[u, v; w] := \texttt{pseudo\_fma}[r, s; t] \circledast B^{U+V}$;
  end if;
end if.
STOP.
```

In the first case, $U + V < W \leq E_{\max}$; in particular, B^{U+V} can underflow but cannot overflow. From $P = \lfloor (W + U - V)/2 \rfloor$ and $Q = \lceil (W + V - U)/2 \rceil$ follows

$$U - P \leq U - \left(\frac{W + U - V}{2} - \frac{1}{2} \right) = \frac{U + V - W}{2} + \frac{1}{2} < \frac{1}{2}; \qquad (28)$$

$$V - Q \leq V - \left(\frac{W + V - U}{2} \right) = \frac{U + V - W}{2} < 0; \qquad (29)$$

Thus, neither $r = \mathtt{M}(u) \circledast B^{U-P}$ nor $s = \mathtt{M}(v) \circledast B^{V-Q}$ can overflow. Furthermore, because all exponents are integers, and because for all integers $K, L \in \mathbb{Z}$,

$$\left\lfloor \frac{K}{2} \right\rfloor + \left\lceil \frac{K}{2} + L \right\rceil = K + L, \qquad (30)$$

it follows that

$$P + Q = \left\lfloor \frac{W + U - V}{2} \right\rfloor + \left\lceil \frac{W + U - V}{2} + (V - U) \right\rceil = W. \qquad (31)$$

Thus, $t = \mathtt{M}(w) = \mathtt{M}(w) * B^{W-(P+Q)}$, which confirms the scaling factor B^{P+Q}.

In the second case, $W - (U + V) \leq 0$, so that $|t| < B$, which does not overflow but might underflow; also, $|r \circledast s| \in [1, B^2]$, which neither overflows nor underflows. Consequently, `pseudo_fma`$(r, s; t)$ maintains its accuracy and neither overflows nor underflows, provided only that $E_{\max} - E_{\min} \geq 2m$. Therefore, after rescaling `pseudo_fma`$(r, s; t)$, the result `scaled_pseudo_fma`$(u, v; w)$ can overflow or underflow if and only if the real number $(u * v) + w$ overflows or underflows.

5. EXACT PRODUCT BY FUSED MULTIPLY-ADD INSTRUCTIONS

Similar to Kahan's compensated summation, which in either base $B \in \{2, 3\}$ yields an exact floating-point sum by splitting it into two floating-point numbers [Dekker 1971, p. 228; Higham 1996, p. 92], Theorem 2 shows how to use a fused multiply-add instruction to get an exact floating-point product by splitting it into two floating-point numbers, with *any* base.

Theorem 2. *For all floating-point numbers $r, s \in [1/B, 1)$ in any base B, let*

$$p := r \circledast s, \tag{32}$$

$$q := \texttt{fma}(r, s; -p). \tag{33}$$

If the arithmetic is faithful, then, exactly,

$$r * s = p + q. \tag{34}$$

Proof. Because r and s are floating-point numbers, each with m significant digits, the digital expansion in base B of their mathematical product $r * s$ requires at most $2m$ digits:

$$r * s = (-1)^c * B^E * b_0.b_1 \cdots b_{m-1}b_m \cdots b_{2m-1}. \tag{35}$$

Hence, either of two rounding cases can arise. In the first case, rounding $r * s$ *down* truncates the last m digits, so that the floating-point result $r \circledast s$ and the exact mathematical difference $(r * s) - (r \circledast s)$ (which exists abstractly but is not yet computed) can be represented by two floating-point numbers with m digits each:

$$r \circledast s = (-1)^c * B^E * b_0.b_1 \cdots b_{m-1}, \tag{36}$$

$$(r * s) - (r \circledast s) = (-1)^c * B^{E-m} * b_m.b_{m+1} \cdots b_{2m-1}. \tag{37}$$

In the second case—which occurs only if $b_m \cdots b_{2m-2}b_{2m-1} \geq 0 \cdots 01 > 0$ with a faithful arithmetic—rounding $r * s$ *up* drops the last m digits and adds $1 * B^{1-m}$ to the first m digits of the mantissa of $r * s$, which produces a first floating-point number $r \circledast s$ with m digits:

$$r \circledast s = (-1)^c * B^E * [(b_0.b_1 \cdots b_{m-1}) + (B^{1-m} * 1.0 \cdots 0)] \tag{38}$$

$$= (-1)^c * B^{E'} * b'_0.b'_1 \cdots b'_{m-1}, \tag{39}$$

where $E' \in \{E, E + 1\}$ depending on the last carry over. The rounding contribution $1 * B^{1-m}$ then also appears in the exact mathematical difference:

$$(r * s) - (r \circledast s) = (-1)^c * B^{E'-(m-1)} * \left[(0.b_m \cdots b_{2m-1}) - (1.0 \cdots 0)\right]. \tag{40}$$

However, because $b_m \cdots b_{2m-2}b_{2m-1} \geq 0 \cdots 01 > 0$, the following inequalities hold:

$$B^{-m} \leq 0.b_m \cdots b_{2m-1} \leq 1 - B^{-m}, \tag{41}$$

$$B^{-m} \leq 1 - 0.b_m \cdots b_{2m-1} \leq 1 - B^{-m}. \tag{42}$$

Hence, the exact mathematical difference requires only m digits:

$$(r * s) - (r \circledast s) = (-1)^c * B^{E'} * b'_m.b'_{m+1} \cdots b'_{2m-1}. \tag{43}$$

Thus, in either rounding case, $(r * s) - (r \circledast s)$ happens to be a floating-point number with at most m digits, which rounding to m digits leaves intact. Consequently, `fma` produces the exact result $(r * s) - (r \circledast s)$:

$$q = \texttt{fma}[r, s; -(r \circledast s)] = (r * s) - (r \circledast s). \tag{44}$$

Therefore, $(r * s) = (r \circledast s) + [(r * s) - (r \circledast s)] = p + q$ exactly. $\square$

With the same notation as in Algorithm 2, Theorem 2 extends from floating-point numbers $r, s \in [1/B, 1)$ to any floating-point numbers $u, v \in \texttt{F}$ by replacing `fma` by `scaled_fma`. Moreover, induction extends Theorem 2 to any number of factors, first as in Corollary 1, and second as in the subsequent Remark 1.

COROLLARY 1. *For each positive integer j, and for all floating-point numbers $x_1, \ldots, x_j$, floating-point multiplications and fused multiply-add instructions produce 2^{j-1} floating-point numbers $g_1, \ldots, g_{2^{j-1}}$ such that, exactly,*

$$\prod_{i=1}^{j} x_i = \sum_{i=1}^{2^{j-1}} g_i. \tag{45}$$

PROOF. To avoid needless overflows and underflows, collect all exponents beforehand, and hence assume that $1/B \leq |x_i| < 1$ for every $i \in \{1, \ldots, j\}$. Further scaling of the next factor x_i can also occur during the computation if the accumulated product risks underflowing. Theorem 2 shows that the corollary holds for $j := 2$. To extend the corollary from any j to $j + 1$ factors, assume that

$$\prod_{i=1}^{j} x_i = \sum_{i=1}^{2^{j-1}} g_i. \tag{46}$$

Then split each product $g_i * x_{j+1}$ as defined by Theorem 2:

$$\begin{aligned} p_i &:= g_i \circledast x_{j+1}, & (47)\\ p_{i+2^{j-1}} := q_i &:= \texttt{fma}(g_i, x_{j+1}; -p_i). & (48) \end{aligned}$$

Consequently,

$$\begin{aligned} \prod_{i=1}^{j+1} x_i &= \left(\prod_{i=1}^{j} x_i\right) * x_{j+1} & (49)\\ &= \left(\sum_{i=1}^{2^{j-1}} g_i\right) * x_{j+1} & (50)\\ &= \sum_{i=1}^{2^{j-1}} (g_i * x_{j+1}) & (51) \end{aligned}$$

$$= \sum_{i=1}^{2^{j-1}} (p_i + p_{i+2^{j-1}}) \tag{52}$$

$$= \sum_{i=1}^{2^j} p_i. \qquad \square \tag{53}$$

Remark 1. Additional software or hardware can reduce the number of summands in Corollary 1 from 2^{j-1} to j, because the product of j floating-point numbers requires at most $j * m$ consecutive bits. Yet such additional software or hardware could become as complex as the mutliple-precision software or superlong accumulator that the present considerations attempt to avoid. For instance, hardware to this effect could consist of Schmookler's Large Arithmetic and Logic Unit [Schmookler 1980] or of Kulisch's superlong accumulator [Kulisch and Miranker 1986].

Software to the same effect could consist of a logical test to perform a carry-over from one floating-point word to another. Specifically, if $x_1, x_2, x_3 \in [1/B, 1)$, then

$$\begin{aligned} x_1 * x_2 &= p_1 + q_1, && (54)\\ (x_1 * x_2) * x_3 &= (p_1 + q_1) * x_3 = (p_1 * x_3) + (q_1 * x_3) && (55)\\ &= (p'_1 + q'_1) + (p'_2 + q'_2) = p'_1 + (q'_1 + p'_2) + q'_2, && (56) \end{aligned}$$

where

$$\begin{aligned} p'_1 &\in [B^{-1}, 1), && (57)\\ q'_1, p'_2 &\in [B^{-2}, B^{-1}), && (58)\\ q'_2 &\in [B^{-3}, B^{-2}). && (59) \end{aligned}$$

Hence, either $q'_1 \oplus p'_2 < B^{-1}$ and $q'_1 \oplus p'_2 = q'_1 + p'_2$ collapses into one floating-point word, or $q'_1 \oplus p'_2 \geq B^{-1}$ and a logical test must then first subtract B^{-1} from the larger summand and carry B^{-1} over to p'_1 before addding exactly $(q'_1 \ominus B^{-1}) \oplus p'_2 = (q'_1 - B^{-1}) + p'_2$ also into one floating-point word. The inequality $x_1 * x_2 * x_3 < 1$ also guarantees that $p'_1 + B^{-1} < 1$ does not overflow.

Similar operations can be performed at each stage, beginning with the penultimate sumands $q^{(j-2)}_{j-1} + p^{(j-2)}_{j-1}$ and ending with the last carry-over to $p^{(j-2)}_1$.

Adding the summands in floating-point arithmetic would merely reproduce the floating-point product without increasing its accuracy [Higham 1996, p. 93]. Nevertheless, the foregoing results will now increase the accuracy of *sums* of products by adding all their summands through doubly compensated summation.

6. PROVABLY ACCURATE SUMS OF MATRIX PRODUCTS

Real and complex dot products consist of sums of products. Each product can be computed exactly as two floating-point numbers through one floating-point product and one fused multiply-add instruction, as in Theorem 2. (To avoid overfow and underflow, in this section replace `fma` by `scaled_fma` as in Algorithm 2.)

The sum of all the resulting floating-point numbers can then be computed accurately to within the penultimate digit, for instance, through Schmookler's macros to simulate a long adder [Schmookler 1980], or through the doubly compensated summation by Algorithm 4 [Higham 1996, p. 97; Priest 1992, pp. 64–66]:

Algorithm 4 (Priest's Doubly Compensated Summation).

```
DATA: A positive integer $N$, and floating-point numbers $x_1, \ldots, x_N$.
START:
Order $x_1, \ldots, x_N$ in nonincreasing magnitudes, so that $|x_1| \geq \cdots \geq |x_N|$.
Set $s_1 := x_1$ ($s_1$ is the first partial sum);
set $c_1 := 0$ ($c_1$ is the first rounding error).
For $k = 2, \ldots, N$ do the following floating-point operations:
    $y_k := c_{k-1} \oplus x_k$,
    $u_k := x_k \ominus (y_k \ominus c_{k-1})$,
    $t_k := y_k \oplus s_{k-1}$,
    $v_k := y_k \ominus (t_k \ominus s_{k-1})$,
    $z_k := u_k \oplus v_k$,
    $s_k := t_k \oplus z_k$,
    $c_k := z_k \ominus (s_k \ominus t_k)$;
next $k$.
STOP.
```

Result: Let $S := \sum_{k=1}^{N} x_k$; then the exact sum S and the computed sum s_N differ from each other by at most one unit in their penultimate digits:

$$|s_N - S| \leq B^{1-m}|S|. \tag{60}$$

Doubly compensated summation now can add accurately products split by fused multiply-add instructions.

6.1 Real Dot Products

For any two arrays of n floating-point numbers each,

$$\vec{\mathbf{r}} := (r_1, \ldots, r_n) \in \mathrm{F}^n, \tag{61}$$

$$\vec{\mathbf{s}} := (s_1, \ldots, s_n) \in \mathrm{F}^n, \tag{62}$$

compute an array $\vec{\mathbf{p}} \in \mathrm{F}^{2n}$ of $2n$ floating-point numbers as follows:

$$p_k := r_k \circledast s_k \text{ for every } k \in \{1, \ldots, n\}, \tag{63}$$

$$p_{n+k} := q_k := \mathtt{fma}(r_k, s_k; -p_k) \text{ for every } k \in \{1, \ldots, n\}. \tag{64}$$

By Theorem 2, it follows that $r_k * s_k = p_k + q_k = p_k + p_{k+n}$, whence

$$\vec{\mathbf{r}} \bullet \vec{\mathbf{s}} := \sum_{k=1}^{n} (r_k * s_k) = \sum_{k=1}^{n} (p_k + p_{n+k}). \tag{65}$$

Hence, sort the array $\vec{\mathbf{p}}$ in non-increasing order of magnitudes, and add its entries by doubly compensated summation (Algorithm 4), which produces the dot product $\vec{\mathbf{r}} \bullet \vec{\mathbf{s}}$ accurately to one unit in its penultimate digit.

In contrast, without the fused multiply-add instruction, there was hitherto no guarantee of any relative accuracy in dot products computed through conventional (IEEE) floating-point arithmetic, with which all the computed digits could be wrong [Higham 1996, p. 69]. See also Example 3 in Section 7.

6.2 Real Cross Products

With $n := 3$, the cross product $\vec{\mathbf{r}} \times \vec{\mathbf{s}}$ can be computed through three dot products in $\mathtt{F}^2$, each one of the form $(r_k, -r_\ell) \bullet (s_\ell, s_k)$:

$$\vec{\mathbf{r}} \times \vec{\mathbf{s}} := \begin{pmatrix} r_2 * s_3 - r_3 * s_2 \\ r_3 * s_1 - r_1 * s_3 \\ r_1 * s_2 - r_2 * s_1 \end{pmatrix} = \begin{bmatrix} (r_2, -r_3) \bullet (s_3, s_2) \\ (r_3, -r_1) \bullet (s_1, s_3) \\ (r_1, -r_2) \bullet (s_2, s_1) \end{bmatrix}. \tag{66}$$

Applying Algorithm 4 to each dot product (as in Section 6.1) computes $\vec{\mathbf{r}} \times \vec{\mathbf{s}}$ accurately to one unit in the penultimate digit of each coordinate.

6.3 Complex Multiplication

The complex product $w * z$ of two complex numbers $w := (u, v)$ and $z := (x, y)$ can be computed through two real dot products. Indeed, with complex numbers written in columns to track their real and imaginary parts more easily through the equations,

$$\begin{pmatrix} u \\ v \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} u * x - v * y \\ u * y + v * x \end{pmatrix} \tag{67}$$

$$= \begin{bmatrix} (u, -v) \bullet (x, y) \\ (u, -v) \bullet (y, -x) \end{bmatrix}. \tag{68}$$

These dot products correspond physically to the dot products of the complex conjugate of the fluid velocity $(u, -v)$ with the tangent vector (x, y) and the normal vector $(y, -x)$ along any curve [Chorin and Marsden 1993, p. 52]. Applying Algorithm 4 to each coordinate computes $w * z$ accurately to one unit in the penultimate digit in the real part *and* in the imaginary part. Hitherto—without fused multiply-add instructions—conventional (IEEE) floating-point arithmetic could lead to completely erronerous results in either the real or the imaginary part of the product [Hewlett-Packard Co. 1984, p. 73; Higham 1996, p. 80]. See also Example 3 in Section 7.

6.4 Complex Affine Operations

For any three complex numbers $w := (u, v)$, $z := (x, y)$, and $c := (r, s)$, the complex affine operation $(w * z) + c$ can be computed to one unit in the penultimate digit of each coordinate, by splitting the complex product as just demonstrated, and then adding the separate parts and the constant with doubly compensated summation (Algorithm 4) in each coordinate:

$$(w * z) + c = \tag{69}$$

$$\left[\begin{pmatrix} u \\ v \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix}\right] + \begin{pmatrix} r \\ s \end{pmatrix} = \tag{70}$$

$$\begin{bmatrix} (u \circledast x) + \mathtt{fma}[u, x; -(u \circledast x)] + (-v \circledast y) + \mathtt{fma}[-v, y; -(-v \circledast y)] + r \\ (u \circledast y) + \mathtt{fma}[u, y; -(u \circledast y)] + (v \circledast x) + \mathtt{fma}[v, x; -(v \circledast x)] + s \end{bmatrix}. \tag{71}$$

6.5 Complex Dot Products

For any two arrays of complex floating-point numbers

$$\vec{\mathbf{w}} := (w_1, \dots, w_n) = [(u_1, v_1), \dots, (u_n, v_n)], \tag{72}$$

$$\vec{\mathbf{z}} := (z_1, \dots, z_n) = [(x_1, y_1), \dots, (x_n, y_n)], \tag{73}$$

their complex dot product $\vec{\mathbf{w}} \bullet \vec{\mathbf{z}}$ can be computed through two real dot products:

$$\begin{aligned} \vec{\mathbf{w}} \bullet \vec{\mathbf{z}} &:= \sum_{k=1}^{n} w_k * \overline{z_k} && (74) \\ &= \sum_{k=1}^{n} \begin{pmatrix} u_k \\ v_k \end{pmatrix} * \begin{pmatrix} x_k \\ -y_k \end{pmatrix} && (75) \\ &= \sum_{k=1}^{n} \begin{bmatrix} (u_k * x_k) + (v_k * y_k) \\ (v_k * x_k) - (u_k * y_k) \end{bmatrix} && (76) \\ &= \begin{bmatrix} \sum_{k=1}^{n} (\vec{\mathbf{u}}; \vec{\mathbf{v}}) \bullet (\vec{\mathbf{x}}; \vec{\mathbf{y}}) \\ \sum_{k=1}^{n} (\vec{\mathbf{u}}; \vec{\mathbf{v}}) \bullet (-\vec{\mathbf{y}}; \vec{\mathbf{x}}) \end{bmatrix}, && (77) \end{aligned}$$

where $(\vec{\mathbf{p}}; \vec{\mathbf{q}})$ denotes the concatenation of two arrays:

$$(\vec{\mathbf{p}}; \vec{\mathbf{q}}) = (p_1, \dots, p_n; q_1, \dots, q_n), \tag{78}$$

and where both real dot products can be computed accurately as in Section 6.1.

6.6 Determinants of Small Matrices

The determinant of a real or complex matrix A with two rows and two columns can be computed through one real or complex dot product:

$$\begin{aligned} \det \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} &= (a_{1,1} * a_{2,2}) - (a_{2,1} * a_{1,2}) && (79) \\ &= (a_{1,1}, a_{2,1}) \bullet \overline{(a_{2,2}, -a_{1,2})}. && (80) \end{aligned}$$

An algorithm attributed to Kahan uses one multiplication, one subtraction, and two fused multiply-add instructions to compute $\det(A)$ "with high relative accuracy" [Higham 1996, p. 65], but for some data the error can be of the order of a unit in the $2m$th digit of $|a_{1,2} * a_{2,1}|$, which can also be of the order of the determinant to be computed [Higham 1996, p. 537]. The method proposed here uses two fused multiply-add operations (one to split $a_{1,1} * a_{2,2}$ and one to split $a_{2,1} * a_{1,2}$) followed by a doubly compensated summation to yield $\det(A)$ accurately to the penultimate digit. In principle, though perhaps gradually impractically as the size increases, Corollary 1 extends the foregoing result to larger matrices. For instance, the computation of elements of volume can proceed through the computation of determinants, which can overflow without warning the user in integer arithmetic or can be inaccurate in floating-point arithmetic [O'Rourke 1993, p. 156]. Still,

$$\det\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} = a_{1,1} * a_{2,2} * a_{3,3} + a_{1,2} * a_{2,1} * a_{3,3} \tag{81}$$

$$+a_{1,3} * a_{2,2} * a_{3,1} + a_{1,1} * a_{2,3} * a_{3,2} \tag{82}$$

$$+a_{1,2} * a_{2,3} * a_{3,1} + a_{1,3} * a_{2,1} * a_{3,2}, \tag{83}$$

where Corollary 1 splits each product $a_{1,i} * a_{2,j} * a_{3,k}$ into $2^{3-1} = 4$ floating-point numbers, for a total of $4 * 6 = 24$ floating-point numbers, which doubly compensated summation can then add accurately to the penultimate digit.

6.7 Real or Complex Affine Matrix Operations

For all real or complex matrices W, Z, C with compatible dimensions, the affine operation $(W * Z) + C$ can be expressed as one dot product in each coordinate, which can be computed accurately to one unit in the penultimate digit, as in Sections 6.1 and 6.5.

6.8 Discriminants of Real or Complex Quadratic, Cubic, or Quartic Equations

The discriminant d^2 of a quadratic equation with real or complex coefficients a, b, c can be computed through one real or complex dot product:

$$d^2 := b^2 - 4ac \tag{84}$$

$$= (b, a, a, a, a) \bullet \overline{(b, -c, -c, -c, -c)}, \tag{85}$$

which can be computed accurately to the penultimate digit, as in Sections 6.1 and 6.5. In principle, though perhaps gradually impractically as the degree increases, Corollary 1 then extends the foregoing result to the accurate computation of the discriminants

$$18c_0 * c_1 * c_2 * c_3 + (c_1 * c_2)^2 - \left[27 * (c_0 * c_3)^2 + 4 * \left(c_0 * c_2^3 + c_3 * c_1^3\right)\right] \tag{86}$$

of cubic equations $c_3 * z^3 + c_2 * z^2 + c_1 * z + c_0 = 0$ [Birkhoff and MacLane 1977, p. 120; Chandrasekharan 1985, p. 40; Dickson 1914, p. 33], and the discriminants

$$\left[c_4 * c_0 - (c_3 * c_1/4) + 3 * (c_2/6)^2\right]^3 \tag{87}$$

$$-27 * \left[(c_4 * c_2 * c_0/6) + (c_3 * c_2 * c_1/48) - \left(c_3 * c_1^2/64\right) - (c_2/6)^3 - c_0 * (c_4/4)^2\right]^2$$

of quartic equations $c_4 * z^4 + c_3 * z^3 + c_2 * z^2 + c_1 * z + c_0 = 0$ [Chandrasekharan 1985, p. 41].

6.9 Real or Complex Polynomial Evaluations

More generally than for determinants and discriminants, for any real or complex polynomial, although perhaps practically only for low degrees, fused multiply-add instructions can split all the products and powers into a sequence of summands according to Corollary 1, which doubly compensated summation can then add with Algorithm 4 to evaluate the polynomial accurately to the penultimate digit.

7. DECIMAL CALCULATORS

Doubly compensated summation and Theorem 2 hold for every base, but the full fused multiply-add instruction is not available on all decimal calculators.

Nevertheless, typical decimal hand-held calculators already include extended (though not double) precision for fused multiply-add and other instructions for matrices, even though such instructions are not required by the IEEE 854-1987 Standard [Institute of Electrical and Electronic Engineers 1987]. For instance, Hewlett-Packard's HP-15, HP-28, and HP-48 series calculators include a "residual" instruction `RSD`, so that `B A X RSD` computes $B-(A*X)$ using three additional decimal digits, as do the instructions for dot products and determinants [Hewlett-Packard Co. 1984, p. 104; 1986, p. 112; 1987, p. 79; 1990, p. 362; 1994].

Example 3. This example shows how a properly rounding decimal floating-point arithmetic can produce not only all wrong digits but also the wrong sign for a complex or real dot product; the same example also shows how a properly rounding decimal floating-point arithmetic can fail to preserve the positivity of the discriminant $d^2 := b^2 - 4*a*c$ for the following quadratic equation $a*x^2+b*x+c=0$:

$$0.3\,124\,999\,999 * x^2 - 0.707\,106\,781\,1 * x + 0.4 = 0. \tag{88}$$

Exact integer calculations with *Mathematica* [Wolfram 1996] reveal that

$$d^2 = 3.760\,331\,721 * 10^{-11} > 0; \tag{89}$$

consequently, the proposed quadratic equation has *two* distinct real roots. Yet a ten-digit decimal calculator (here a Hewlett-Packard HP-15C) computes an approximation $\tilde{d}^2$ of d^2 as follows:

$$\begin{aligned}
b \circledast b &= (-0.707\,106\,781\,1) \circledast (-0.707\,106\,781\,1) && (90)\\
&= 0.499\,999\,999\,9; && (91)\\
a \circledast c &= 0.3\,124\,999\,999 \circledast 0.4 && (92)\\
&= 0.125\,000\,000\,0; && (93)\\
4 \circledast (a \circledast c) &= 4 \circledast 0.125\,000\,000\,0 && (94)\\
&= 0.500\,000\,000\,0; && (95)\\
\tilde{d}^2 &:= (b \circledast b) \ominus [4 \circledast (a \circledast c)] && (96)\\
&= 0.499\,999\,999\,9 \ominus 0.500\,000\,000\,0 && (97)\\
&= -0.000\,000\,000\,1 && (98)\\
&< 0. && (99)
\end{aligned}$$

This negative value of the computed discriminant $\tilde{d}^2$ would indicate that the proposed quadratic equation has no real solution. As already mentioned about the error analysis for conventional floating-point arithmetic, the size of the discrepancy

$$d^2 - \tilde{d}^2 = 3.760\,331\,721 * 10^{-11} - (-0.000\,000\,000\,1) = 13.760\,331\,721 * 10^{-11} \tag{100}$$

is indeed of the order of one Unit in the Last Place (`ulp`) of the Operand with the Largest Magnitude (`OLM`), which is here

$$\mathtt{ulp}(\mathtt{OLM}) = \mathtt{ulp}(|b|) = \mathtt{ulp}(0.707\,106\,781\,1) = 10^{-10}. \tag{101}$$

In other words, the Operand with the Largest Magnitude is so large that one unit in its last place is still larger than the result to be computed. Hence, not only is the magnitude of the computed result (10^{-10}) thrice as large as the magnitude of the exact result ($3.8 * 10^{-11}$), but also the *sign* of the computed result is wrong.

In contrast, the use of fused multiply-add instructions—or any similar `pseudo_fma` instruction with extended precision as in Section 3—and doubly compensated summation restores the correct magnitude and sign for the computed result. For instance, the HP-15C allows for dot products in extended thirteen-digit precision through its `MATRIX 5` instruction [Hewlett-Packard Co. 1984, p. 104 & p. 208]. To use this extended precision, enter the arrays

$$\begin{aligned} \mathtt{A} &:= (b, a, a, a, a), \\ \mathtt{B} &:= (b, -c, -c, -c, -c); \end{aligned}$$

having defined `C` as the destination of the result, recall both arrays on the stack, compute their dot product, and recall the result:

```
RCL MATRIX A
RCL MATRIX B
f MATRIX 5
RCL MATRIX C
3.760000 E − 11
```

which agrees with the exact result d^2 to four significant digits. Therefore, there are two distinct real solutions (which happen to be 1.131 361 039 and 1.131 380 661).

The same computations with scaled up coefficients $a := 0.3\,124\,999\,999 * 10^{99}$, $b := -0.707\,106\,781\,1 * 10^{99}$, $c := 0.4 * 10^{99}$, would have required a `scaled_pseudo_fma` operation as in Algorithm 3 instead of `f MATRIX 5` to avoid overflows.

This example has also demonstrated how a properly rounding decimal floating-point arithmetic can produce all wrong digits and the wrong sign in the dot product $\mathtt{A} \bullet \mathtt{B}$, which is d^2 but is computed as $\tilde{d}^2$, and in the real part of the complex product $w * z$ with $w := (b, 4a)$ and $z := (b, c)$, which is also d^2 but is also computed as $\tilde{d}^2$.

Remark 2. The situation in Example 3 does *not* occur with binary arithmetic. Indeed, if $B = 2$, then multiplications and divisions by 2 are exact, and multiplications by 2 distribute over additions and subractions, so that

$$(b \circledast b) \ominus [4 \circledast (a \circledast c)] \;=\; 2 * 2 * \left\{ \left[\left(\frac{b}{2}\right) \circledast \left(\frac{b}{2}\right) \right] \ominus (a \circledast c) \right\}. \qquad (102)$$

Moreover, the monotonicity of a properly rounding (or always truncating, or always upward rounding) floating-point multiplication guarantees that if $(b/2) * (b/2) \geq a * c$, then $(b/2) \circledast (b/2) \geq (a \circledast c)$, whence $[(b/2) \circledast (b/2)] - (a \circledast c) \geq 0$, and hence $[(b/2) \circledast (b/2)] \ominus (a \circledast c) \geq 0$, by monotonicity of $\ominus$ [Higham 1996, p. 60]. Consequently, binary floating-point arithmetic can round a positive or a negative discriminant to zero, but it can neither round a positive discriminant to a negative discriminant, nor round a negative discriminant to a positive discriminant.

8. CONCLUSIONS

Combined with doubly compensated summation, the scalar fused multiply-add instruction $\texttt{fma}(a, x; y) = \texttt{float}[(a * x) + y]$ allows for real or complex matrix affine (BLAS) operations $(A * X) + Y$ accurate to within one unit in the penultimate place of each coordinate, without resorting to slower multiple-precision software. Because of the ubiquity of such matrix affine operations in industrial and scientific computing, inclulding computer graphics, and because hitherto the highest accuracy provable for single floating-point operations amounted to one half of one unit on the last place (`ulp`) not of the result but of the potentially much larger Operand with the Largest Magnitude (`OLM`), the result just proved would seem to warrant serious considerations for the inclusion of such fused multiply-add instructions in high-level computer languages for example, `C`, `FORTRAN`, and `PACSAL`, and in computing standards, in particular, in the next edition of the IEEE 754-1985 [Institute of Electrical and Electronic Engineers 1985] and IEEE 854-1987 [Institute of Electrical and Electronic Engineers 1987] Standards.

In the context just examined for matrix arithmetic, the fused multiply-add instruction serves mainly to produce exact multiplications by splitting each product into two floating-point numbers. Thereafter, Priest's [1992] doubly compensated summation or Schmookler's [1980] macros serve mainly to add split products accurately by simulating an adder and accumulator of length $2(|E_{\max} - E_{\min}| + m)$. Therefore, the ultimate step for accurate matrix arithmetic would be to provide such an accumulator besides the fused multiply-add instruction.

ACKNOWLEDGMENTS

I thank the two anonymous referees for suggestions that greatly improved the manuscript.

REFERENCES

ABBOTT, P. H., BRASH, D. G., CLARK, C. W. III, CRONE, C. J., EHRMAN, J. R., EWART, G. W., GOODRICH, C. A., HACK, M., KAPERNICK, J. S., MINCHAU, B. J., SHEPARD, W. C., R. M. SMITH, S., TALLMAN, R., WALKOWIAK, S., WATANABE, A., AND WHITE, W. R. 1999. Architecture and software support in IBM

S/390 parallel enterprise servers for IEEE floating-point arithmetic. *IBM J. Res. Develop. 43,* 5/6 (Sept./Nov.), 723–760.

Apple Computer, Inc. 2002. Altivec instruction cross-reference. `http://developer.apple.com/hardware/ve/instructions/vec_madd.html`.

Birkhoff, G. and MacLane, S. 1977. *A Survey of Modern Algebra*, 4th ed. Macmillan, New York, NY.

Bleher, J. H. 2001. Elements of scientific computing. In *Perspectives on Enclosure Methods*, U. Kulisch, R. Lohner, and A. Facius, Eds. Springer-Verlag, New York, 99–103.

Chandrasekharan, K. 1985. *Elliptic Functions*. Springer-Verlag, Berlin-Heidelberg-New York.

Chorin, A. J. and Marsden, J. E. 1993. *A Mathematical Introduction to Fluid Mechanics*, 3rd ed. Texts in Applied Mathematics, vol. 4. Springer-Verlag, New York, N.Y.

Coan, J. S. 1977. *Advanced Basic: Applications and Problems*. Hayden, Rochelle Park, N.J.

Coleman, T. F. and Van Loan, C. 1988. *Handbook for Matrix Computations*. Society for Industrial and Applied Mathematics, Philadelphia, Pa.

Dekker, T. J. 1971. A floating-point technique for extending the available precision. *Num. Math. 18*, 224–242.

Demmel, J. and Kahan, W. 1990. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput. 11*, 5 (Sept.), 873–912.

Dickson, L. E. 1914. *Elementary Theory of Equations*. Wiley, London, UK.

Duff, I. S. 1996. Foreword. In *Lectures on Finite Precisikon Computations*. Society for Industrial and Applied Mathematics, Philadelphia, Pa, xiii–xiv.

Forsythe, G. E. 1970. Pitfalls in computation, or why a math book isn't enough. *Amer. Math. Monthly 77,* 9 (Nov.), 931–956.

Gohlender, G. and Grüner, K. 1983. Realization of an optimal computer arithmetic. In *A New Approach to Scientific Computing*, U. W. Kulisch and W. L. Miranker, Eds. Notes and Reports in Computer Science and Applied Mathematics, vol. 7. Academic Press, New York, N.Y. 247–268.

Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23*, 1 (Mar.), 5–48.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations*, 2nd ed. Johns Hopkins University Press, Baltimore, Md.

Grandine, T. A. 2000. Applications of contouring. *SIAM Rev. 42*, 2 (June), 297–316.

Hammer, R., Neaga, M., and Ratz, D. 1993. PASCAL-XSC: New concepts for scientific computation and numerical data processing. In *Scientific Computing with Automatic Result Verification*, E. Adams and U. Kulisch, Eds. Mathematics in Science and Engineering, vol. 189. Academic Press, Orlando, Fla., 15–44.

Henrici, P. 1982. *Essentials of Numerical Analysis with Pocket Calculator Demonstrations*. Wiley, New York.

Hewlett-Packard Co. 1984. *HP-15C Advanced Functions Handbook*. Hewlett-Packard Co., Corvallis Division, 1000 NE Circle Blvd., Corvallis, OR 97330, USA.

Hewlett-Packard Co. 1986. *HP-28C Reference Manual*, 1st ed. Hewlett-Packard Co., Corvallis Division, 1000 NE Circle Blvd., Corvallis, OR 97330, USA.

Hewlett-Packard Co. 1987. *HP-28S Reference Manual*, 1st ed. Hewlett-Packard Co., Corvallis Division, 1000 NE Circle Blvd., Corvallis, OR 97330, USA.

Hewlett-Packard Co. 1990. *HP 48SX Scientific Expandable Calculator Owner's Manual*, 1st ed. Hewlett-Packard Co., Corvallis Division, 1000 NE Circle Blvd., Corvallis, OR 97330, USA.

Hewlett-Packard Co. 1994. *HP 48G Series User's Guide*, 7th ed. Hewlett-Packard Co., Corvallis Division, 1000 NE Circle Blvd., Corvallis, OR 97330, USA.

Higham, N. J. 1996. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pa.

Hokenek, E. and Montoye, R. K. 1990. Loading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit. *IBM J. Res. Devel. 34*, 1 (Jan.), 71–77.

Institute of Electrical and Electronic Engineers. 1985. *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*. Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, NY 10017.

INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. 1987. *IEEE Standard for Radix-Independent Floating-Point Arithmetic (ANSI/IEEE Std 854-1987)*. Institute of Electrical and Electronic Engineers, 345 East 47th Street, New York, NY 10017.

INTEL CORPORATION. 2001. Introducing the Intel Itanium architecture. In 2001: `http://developer.intel.com/software/products/itc/architec/itanium/arch_mod/index.htm`; in 2002: `http://www.intel.com/design/Itanium/index.htm`.

KAHAN, W. M. 1972. A survey of error analysis. In *Information Processing 71: Proceedings of IFIP Congress 71, Volume 2—Applications*, C. V. Freiman, Ed. 1214–1239.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language*, 2nd ed. Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, N.J.

KNUTH, D. E. 1998. *The Art of Computer Programming*, 2nd ed. *Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass.

KULISCH, U. W. AND MIRANKER, W. L. 1986. The arithmetic of the digital computer: A new approach. *SIAM Rev. 28*, 1 (Mar.), 1–40.

LAWO, C. 1993. C-XSC: A programming environment for verified scientific computating and numerical data processing. In *Scientific Computing with Automatic Result Verification*, E. Adams and U. Kulisch, Eds. Mathematics in Science and Engineering, vol. 189. Academic Press, Orlando, Fla., 71–86.

MARKSTEIN, P. W. 1990. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM J. Res. Devel. 34*, 1 (Jan.), 111–119.

MONTOYE, R. K., HOKENEK, E., AND RUNYON, S. L. 1990. Design of the IBM RISC System/6000 floating-point execution unit. *IBM J. Res. Devel. 34*, 1 (Jan.), 59–70.

O'CONNELL, F. P. AND WHITE, S. W. 2000. POWER3: The next generation of PowerPC processors. *IBM J. Res. Devel. 44*, 6 (Nov.), 873–884.

OEHLER, R. R. AND GROVES, R. D. 1990. IBM RISC System/6000 processor architecture. *IBM J. Res. Devel. 34*, 1 (Jan.), 23–36.

O'ROURKE, J. 1993. *Computational Geometry in C*. Cambridge University Press, Cambridge, U.K.

OVERTON, M. L. 2001. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, Philadelphia, Pa.

PRIEST, D. M. 1992. On properties of floating point arithmetics: Numerical stability and the cost of accurate computations. Ph.D. dissertation, University of California at Berkeley, Berkeley, CA. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

SCHMOOKLER, M. S. 1980. Design of large ALUs using multiple PLA macros. *IBM J. Res. Devel. 24*, 1 (Jan.), 2–14.

SCOTT, A. P., BURKHART, K. P., KUMAR, A., BLUMBERG, R. M., AND RANSEN, G. L. 1997. Four-way superscalar PA-RISC processors. *Hewlett-Packard J. 48*, 2 (Aug.), 8–15.

STERBENZ, P. H. 1974. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, N.J.

WALTER, W. V. 1993. ACRITH-XSC: A Fortran-like language for verified scientific computating. In *Scientific Computing with Automatic Result Verification*, E. Adams and U. Kulisch, Eds. Mathematics in Science and Engineering, vol. 189. Academic Press, Orlando, Fla., 45–70.

WARREN, H. S., JR. 1990. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. Devel. 34*, 1 (Jan.), 85–92.

WILKINSON, J. H. 1963. *Rounding Errors in Algebraic Processes*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J.

WILSON, I. R. AND ADDYMAN, A. M. 1978. *A Practical Introduction to Pascal*. Springer-Verlag, New York.

WOLFRAM, S. 1996. *The Mathematica Book*, 3rd ed. Wolfram Media, Champaign, Ill.

Received August 2001; revised February 2002; accepted October 2002