RC 21586 (97348) 01NOV99 Computer Science/Mathematics

Research Report

The Fused Multiply-Add Instruction Leads to Algorithms for Extended-Precision Floating Point: Applications to Java and High-Performance Computing

Fred G. Gustavson, José E. Moreira
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598-0218

Robert F. Enenkel
IBM Centre for Advanced Studies
Toronto, Canada

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

The Fused Multiply-Add Instruction Leads to Algorithms for Extended-Precision Floating Point: Applications to Java and High-Performance Computing

Fred G. Gustavson * José E. Moreira † Robert F. Enenkel ‡

Dedicated to Cleve Moler on His 60th Birthday

Abstract

Portability in Java requires that all architectures produce the same result for a particular floating-point calculation. Conceptually, the best definition for this result is the correctly rounded value of the infinitely precise result. Although this is likely an unattainable goal in general, it should be striven for when attainable with reasonable performance. Although many architectures implement IEEE double-precision arithmetic in hardware, providing the correctly rounded result for the basic operations $(+, -, *, /)$ with good performance, most do not provide hardware support for extended precision arithmetic. The performance and accuracy of the resulting software implementations can be poor.

Many machines provide a double-precision fused multiply-add (FMA) instruction that computes $d = a*b+c$ with d correctly rounded. Using the FMA instruction, higher performance algorithms for extended-precision arithmetic can be devised. We present examples of such algorithms for the basic operations, $+, -, *, /$. The algorithms always compute the exact result for each of the operations. This contributes to the Java goal of portability by providing a large spectrum of architectures on which it is possible to efficiently implement correctly rounded extended precision floating point. The FMA is also valuable in large-scale computation of linear algebra problems, where the accumulation of dot products is a basic operation. Performing this accumulation in extended precision greatly increases the size of problem that may be solved before roundoff error becomes a limiting factor. It also produces a much faster way, over just using IEEE arithmetic, of doing the more accurate computation.

Given two integers a and b, there exist unique integers q and r such that $a = bq + r$ with $0 \leq r < b$. In number theory, this lemma is the computational basis for proving the Fundamental Theorem of Arithmetic.

Note that $a = bq + r$ has the same form as the FMA. The exact nature of the FMA and this lemma allow us to define multiplication and division of IEEE numbers exactly. Furthermore, the FMA instruction provides a much faster way to perform these exact computations. Thus, we will demonstrate a clear need for Java to accept the FMA as a standard instruction.

Finally, we demonstrate the accuracy of the algorithms on example problems (matrix multiplication, 2×2 determinant, complex multiplication, and triangle area calculation), for which existing computer arithmetic gives completely inaccurate results in certain instances.

1 Introduction

The increasing interest on using Java [1] as a programming language for numerical codes has brought general attention to a problem well-known to numerical analysts: floating-point computation is hard and full of surprises! Fixed-length binary floating-point numbers in general, and IEEE 754 double-precision in particular, do not obey the usual laws of arithmetic. For example, if a, b, and c are three double-precision floating-point numbers, then $(a + b) + c$ does not necessarily produce the same result as $a + (b + c)$.

The IEEE 754 standard [14] provides a common environment for performing binary floating-point computations. Processors from different vendors that adhere to the standard will produce the same results for the same individual operations. Nevertheless, two Fortran programs compiled for these two machines may still produce different results

*IBM T. J. Watson Research Center, gustav@watson.ibm.com
†IBM T. J. Watson Research Center, jmoreira@us.ibm.com
‡IBM Centre for Advanced Studies, enenkel@ca.ibm.com

[1] Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

for the same input of a computation, depending on what kind of code optimizations are performed. For example, a common compiler optimization is to transform the reduction code,

```
sum = 0
do i=1,n,1
   sum = sum + a(i)*b(i)
end do
```

to an unrolled version with more instruction-level parallelism (let n be a multiple of 4, so that no fix-up code is necessary):

```
s0=0, s1=0, s2=0, s3=0
do i=1,n-3,4
   s0 = s0 + a(i+0)*b(i+0)
   s1 = s1 + a(i+1)*b(i+1)
   s2 = s2 + a(i+2)*b(i+2)
   s3 = s3 + a(i+3)*b(i+3)
end do
sum = s0+s1+s2+s3
```

This transformation is legal according to real arithmetic, since addition of real numbers is associative. However, associativity does not hold for IEEE 754 floating-point numbers. Depending on the unrolling factor, the additions of $a[i] * b[i]$ will be performed in a different order. An unrolling of 4 performs the additions in different order than an unrolling of 8, and both results may be different from the result obtained from the original unrolled code.

Another important issue with floating-point numbers is the accuracy of results. Let x be the result of a computation performed using IEEE double-precision floating-point arithmetic. Let x' be the exact result for the same computation using real arithmetic. The error in ulps of x is defined as

$$\text{ulperr}(x, x') = (x - x')/\text{ulp}(x'),$$

where $\text{ulp}(x) = \min\{\text{succ}(|x|) - |x|, |x| - \text{pred}(|x|)\}$, and $\text{succ}(x)$ and $\text{pred}(x)$ are the the next larger floating-point number after x, and the previous smaller floating-point number before x, respectively.

IEEE 754 specifies that, in round-to-nearest mode, the maximum error for any of the basic operations $(+, -, *, /)$ is 0.5 ulp. It is surprising, however, how fast this error can propagate with even a few operations. Consider the computation of $x = a * d - b * c$. This operation occurs, for example, when computing 2×2 determinants and multiplying complex numbers. We performed a statistical test with 100 million computations of x for different values of a, b, c, and d. The 4 inputs were generated randomly using a uniform distribution between 0 and 1. Even though the maximum error for each of the multiplications and the subtraction was just 0.5 ulps, the maximum error observed for the computation of x was 6.6×10^{15} ulps!

Enter Java, a language designed to be portable on all kinds of platforms. A primary directive of Java is that the execution of a sequence of code with a specific set of inputs must produce exactly the same output on all platforms. (We are ignoring the effects of multithreading. What multithreading adds is a level of nondeterminism. That is, because of possible races, two executions of a Java multithreaded program can actually produce different results. However, multithreading does not not impact the behavior of individual floating-point expressions at the level we are considering.) From a numerical perspective this means that floating-point computations in Java have to produce exactly the same result for a specific input, on all platforms. Java accomplished this by (i) mandating the use of IEEE 754 floating-point and (ii) forcing computations to be performed exactly in the order specified by the source code. Because of this definition, Java does not currently allow the operation $a * b + c$ to be implemented with a fused multiply-add (FMA) instruction, as the result could be slightly different from what is obtained from an IEEE 754 multiply followed by an IEEE 754 add. This operational definition for floating-point computations in Java also prevents many optimizations that are typically performed by optimizing compilers, including the parallelization of reduction previously shown. The inability to reorder computations has a significant impact on Java performance, as compared to Fortran and C [12]. In addition, the operational definition does not address the issue of accuracy. The computation of $a * d - b * c$ in Java will produce the same result everywhere, but it may be the same very wrong result everywhere!

There is a more subtle but very negative effect of using operational definitions of arithmetic computations: operational definitions can severely hamper innovation. Suppose that Java defines a "sort" method that is implemented as an unstable bubble sort. Then it becomes impossible for a vendor to replace it with a much faster implementation (*e.g.*, quicksort) because it could produce slightly different results. Back to more numerical examples, suppose that Java defines a standard matrix multiply method that is implemented as the straightforward three-dimensional inner-product loop. Then it becomes impossible to replace that implementation with, for instance, a Strassen implementation, even if that implementation is much faster and just as accurate. Another example is the current definition of the semantics of elementary math functions (sin, cos, log, ...) in Java. These functions are defined operationally as having to produce the same results as a Java implementation of fdlibm [15]. Although fdlibm is a good portable library, its error properties are less than ideal (*i.e.*, sometimes the error in the result exceeds 0.5 ulps). In 1984, a fast scalar and vector math library that satisfied a 1-ulp error criterion was produced for the IBM System/370 [2] machine [1]. The existence of an FMA instruction for that machine would have been quite useful in enhancing performance. New math libraries [16] have appeared recently that can compute elementary math functions exactly (always with no more than 0.5 ulp in error) and very efficiently. Again, the use of the FMA instruction for efficiency was crucial. Nevertheless, these libraries cannot be used with Java since they can produce results that are different from fdlibm. In summary, Java is riding behind the technology curve because of operational definitions that prevent the use of the latest technology.

A conceptually more appealing definition would be that Java code has to produce, for every floating-point calculation, the correctly rounded value of the infinitely precise result. This is the result that would be obtained by performing all intermediate computations using real arithmetic and then rounding the final result to the nearest floating-point representation. This is no longer an operational definition. All sorts of optimizations can be performed, as long as the correct final answer is obtained. The accuracy issue is also resolved with this approach.

Unfortunately, the goal of computing the infinitely precise result is, in general, unattainable. Starting with double-precision floating point inputs, represented as 64-bit values, each operation performed may require additional bits to represent the precise result. The cost in storage and processing can grow very fast. However, we can still strive for the precise results in some particular cases. The Java Grande Forum is considering a proposal that specifies the transcendental functions in Java as returning the correctly rounded value of the infinitely precise result [7].

When it is impractical to attain the infinitely precise result we can specify a maximum error in ulps, *maxerr*, for the result. Any computed result x of a floating point computation would then have to satisfy

$$\text{ulperr}(x, x') \leq maxerr,$$

where x' is the infinitely precise answer. Computations typically still need to be performed in extended precision to guarantee that we are within the error bounds. Specifying that results of a Java computation must fall within certain error bounds is a compromise that addresses the issues of accuracy, enables certain compiler optimizations, and maintains predictability in the language. The disadvantage is that exact bit-for-bit reproducibility of results is no longer guaranteed. However, for most numerical computations, the field of numerical analysis has shown that it is preferable to get any answer within error bounds of the exact answer, rather than the same reproducible wrong answer.

At this point we advocate a single change to the current Java standard: Java should allow the use of the FMA instruction, as it is a fundamental tool not only for performance but also for building higher accuracy algorithms. A detailed justification will be given in the main body of this paper. Our ultimate goal is to develop a space- and time-efficient technique for performing exact floating-point. That is, we want to maintain enough bits of information and perform the necessary operations at each step to always represent the infinitely precise result. The exact floating-point result can be used then to produce the correctly rounded value at the end of a computation, or to provide an estimate of errors. As a first step in that direction, we will describe in this paper how to produce the exact floating-point result for each of the four basic arithmetic operations ($+, -, *, /$). Our technique is based on a canonical representation that uses multiple double-precision fields to represent an extended precision floating-point number, as described in [4, 8, 11]. These exact floating-point computations can be performed using just the 4 basic IEEE operations and, in some cases, the FMA.

The rest of this paper is organized as follows. Section 2 defines the semantics we adopt for exact floating-point computations. It also describes how the FMA can be used to compute the exact result for multiplication and division. Section 3 discusses the three case studies where the FMA is very useful: (i) matrix-multiplication, (ii) computation of a 2×2 determinant (or complex number multiplication), and (iii) computation of the area of a triangle. Finally, Section 4 presents our conclusions for this work.

[2] System/370 is a trademark of IBM.

2 Performing exact floating-point computations

We start by defining some notation. IEEE double-precision floating point values will be denoted by lower case letters $(a, b, c, \ldots)$. We note that each double-precision floating-point value defines a specific real value. Real arithmetic add, subtract, multiply and divide are denoted by the $+, -, *, /$ operators. These operations are defined over the field of real numbers and they always produce an exact value. That is, $a * b$ represents the exact value obtained by multiplying the IEEE double-precision number a by the IEEE double-precision number b. This value may or may not be representable as an IEEE double-precision floating-point number. IEEE double-precision floating-point operations are denoted by $\oplus, \ominus, \otimes, \oslash$. That is, $a \otimes b$ represents the floating-point number obtained through IEEE multiplication of floating-point numbers a and b. We know that $\text{ulperr}(a \otimes b, a * b) \leq 0.5$, when round-to-nearest mode is used. The same is true of the ulp error in the other three basic operations. The fused multiply-add (FMA) instruction computes the floating-point number

$$d = \text{fl}(a * b + c) \tag{1}$$

where a, b, c, and d are double-precision numbers and $\text{fl}(x)$ is the correctly rounded double-precision representation of the real number x. In other words, d is always accurate within 0.5 ulp.

We want to develop a canonical form for extended-precision floating-point numbers that allows us to represent the exact values of $a + b$, $a - b$, $a * b$, and a/b. It may seem strange that we can always represent such real numbers with a finite number of bits, but the reasoning is as follows. IEEE 754 double-precision floating-point numbers represent a large but finite set of rational numbers. Any IEEE floating-point number can be transformed into an integer by scaling it by a power of 2. Intuitively, when we are dealing with IEEE floating-point numbers we are essentially dealing with integers. From number theory, we know that given any two integers a and b there exist two integers q (the integer quotient) and r (the remainder) such that

$$a = q * b + r. \tag{2}$$

This fundamental relation has the same form as the fused multiply-add operation, and it can be used to define all four basic arithmetic operations:

- Setting $q = 1$ in (2) defines addition.
- Setting $q = -1$ in (2) defines subtraction.
- Setting $r = 0$ in (2) defines multiplication.
- Solving (2) for q and r defines division.

If (2) is evaluated in the set of integer numbers we know we can get exact answers. We can do the same in the set of IEEE floating-point numbers, with one exception. When the result overflows or underflows the range of numbers that can be represented by IEEE floating-point format then (1) will not hold. This is a limitation of our approach and we do not discuss it any further in this text.

We now proceed by using the fundamental relation denoted in (2), together with a canonical form for extended-precision floating-point numbers, to define exact floating-point arithmetic for double-precision numbers.

2.1 The canonical form

We represent an extended-precision number (ch, cl) as the unevaluated sum of two IEEE double-precision numbers ch and cl whose fraction bits do not overlap. That is,

$$(ch, cl) = ch + cl, \text{ where } ch \oplus cl = ch. \tag{3}$$

Equation (3) says that the value of the extended-precision number is equal to the exact (*i.e.*, performed with real arithmetic) sum of the two IEEE numbers ch and cl. However, the result of the IEEE arithmetic sum $ch \oplus cl$ is identically equal to ch. In other words, the absolute value of cl is too small to make a difference when added to ch according to the rules of IEEE 754 double-precision arithmetic.

When we discuss the four basic operations later we will show that the exact result for each of those operations can be represented with an extended precision number in canonical form. We want to show that, given two IEEE double-precision numbers a and b, and one of the basic operations $(+, -, *, /)$ we can compute an extended precision number (ch, cl) such that the appropriate one of the following holds:

$$\begin{aligned} ch + cl &= a + b, \\ ch + cl &= a - b, \\ ch + cl &= a * b, \\ a &= ch * b + cl. \end{aligned}$$

We want to emphasize that the above equalities are exact in the field of real numbers. This is not the same as what is usually called quadruple-precision (128-bit) computation. Although the exact result of the multiplication of two double-precision numbers can be represented as a quadruple-precision number, the same is not necessarily true for addition. If b is much smaller than a (say $b \approx a * 2^{-100}$), then computing $a + b$ in quadruple precision will capture some but not all the bits of b. Meanwhile, the extended-precision number (ch, cl) that we compute is mathematically identical to the exact result of $a + b$.

We note that these semantics for exact floating-point that we adopt were described in [4, 8, 11]. In fact, extended precision format in the IBM XLF compiler [8] (equivalent to the Fortran `real*16` data type) uses the canonical form described here, as opposed to the conventional 128-bit quadruple precision.

2.2 Algorithms for performing exact $+, -, *$ and $/$

We now present algorithms for computing the extended-precision numbers that represent the exact result for each of the four basic arithmetic operations. These are fairly trivial algorithms that demonstrate the power and benefit of allowing the FMA instruction.

Let a and b be double-precision floating-point numbers. We can then compute $(ch, cl) = a$ op b, where op is one of $+, -, *, /$, using only IEEE operations and the FMA instruction as follows:

- $a + b$: This is computed as:

  $ch = a \oplus b$
  if $|a| \geq |b|$ then
  $\quad cl = b \oplus (a \ominus ch)$
  else
  $\quad cl = a \oplus (b \ominus ch)$
  end if

- $a - b$: This is computed the same way as $a + b$, just replacing b with $-b$.

- $a * b$: This can be easily computed using the standard FMA instruction:

  $$\begin{aligned} ch &= a \otimes b \\ cl &= \mathrm{fl}(a * b - ch) \text{ (FMA)} \end{aligned}$$

- a/b: Again, we compute (ch, cl) such that $a = b * ch + cl$. This is done with the FMA as follows:

  $$\begin{aligned} ch &= a \oslash b \\ cl &= \mathrm{fl}(a - b * ch) \text{ (FMA)} \end{aligned}$$

It is important to note that the fused multiply-add instruction is the key to performing exact floating-point multiply and divide. When we add to that the pervasiveness of fused multiply-adds in linear algebra operations, we start to appreciate the relevance of this instruction.

3 Case studies

The extreme usefulness of exact floating-point operations in applications employing repeated additions and/or subtractions, such as iterative refinement methods, dot-products, and matrix-vector computations, is attested to in [4]. We now give numerical results of our methods applied to some example problems.

3.1 Matrix multiplication

The BLAS-3 operation DGEMM is the building block for efficient numerical linear algebra. It computes

$$C = \alpha A \times B + \beta C,$$

where α and β are scalars and A, B, and C are matrices. At the core of DGEMM are computations of the dot-product of rows of A by columns of B. Much literature is devoted to the problem of accurate computation of dot-products [2, 6, 10, 13]. In its simplest form, element (i, j) of $A \times B$ is computed as in

```
s = 0;
for (k=0; k<n; k++) {
   s = s + a[i][k]*b[k][j];
}
```

Each time a term is added to the sum, error accumulates. Using the FMA instruction to implement the loop body helps a little but we still face the problem that at each iteration we have to perform a rounding operation that chops off quite a few bits of the exact result. Instead, we can use exact floating-point arithmetic to greatly improve the accuracy of our result:

```
(sh,sl) = (0,0);
for (k=0; k<n; k++) {
   (sh,sl) = (sh,sl) +
               a[i][k]*b[k][j];
}
s = sh;
```

That is, we first compute $(xh, xl) = a_{ik}b_{kj}$ exactly, using the algorithm in Section 2.2. We then accumulate the (xh, xl) result into an extended precision accumulator (sh, sl). In general, the exact result of this accumulation would require four double-precision numbers to represent, and grow with each additional term. To prevent this fast growth of storage we limit the accumulator to just two double-precision numbers, and round the result to the closest representable extended precision value.

Because we still perform rounding operations in the body of the loop, we cannot guarantee that, at the end of the computation, s is the correctly rounded value of the exact result. Nevertheless, the error in s is certainly many orders of magnitude smaller than before. Note that this "extended-precision" DGEMM does not require any changes to the interface. Input and output matrices continue to be of double-precision floating-point elements. However, the output is much closer to the exact value for the multiplication, as performed over the real field.

Figure 1 shows some results from an experimental evaluation of matrix multiply. We start with two 100×100 matrices A and B, whose elements are random numbers uniformly distributed within a ([low, high)) interval. (The notation $[x, y)$ denotes an interval from x to y, including x but excluding y.) We then compute the product $A \times B$ using (i) just IEEE 754 operations, and (ii) the extended precision method described above. We measure the error in ulps for each element of the result (there are 10,000 resulting elements), with respect to the correctly rounded value of the infinitely precise result. The plots in Figure 1 show the number of occurrences for each value of the error. Each plot is for a particular range of values for the input matrices elements, as denoted in the title of the plot. When extended precision is used, every result is exactly equal to the correctly rounded value. This is illustrated with a dark bar extending all the way to 10,000 occurrences. When just the IEEE operations $\otimes$ and $\oplus$ are used, 75 to 80% of the results differ from the correctly rounded value. The largest observed error in our experiment was 9 ulps.

3.2 Computation of $a * d - b * c$

The computation of $a * d - b * c$ is what got us started into looking at exact floating point. This computation occurs when evaluating the determinant of a 2×2 matrix,

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a * d - b * c,$$

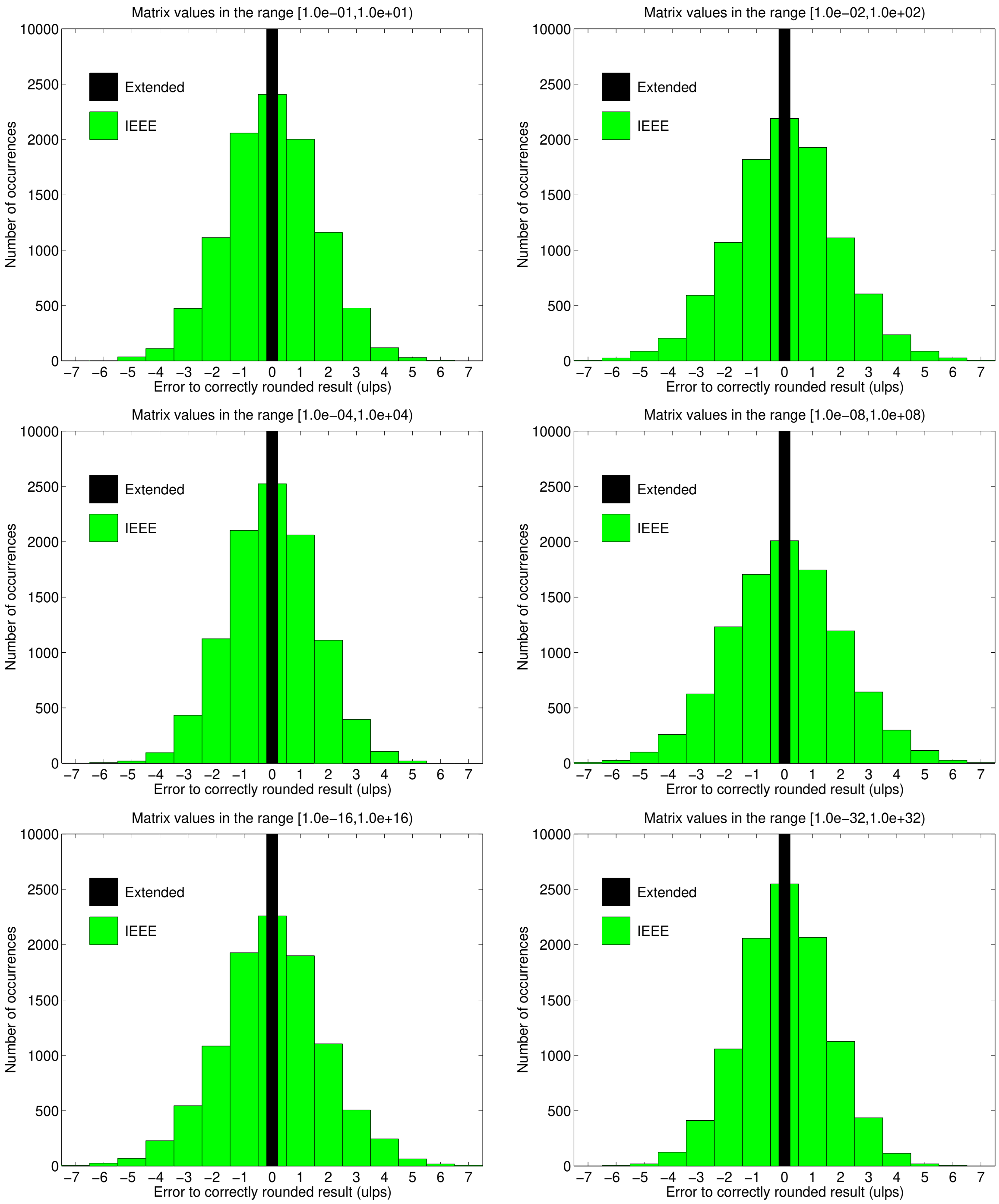

Figure 1: Summary of results for the matrix multiply case study.

approach	range of errors (ulps)
IEEE	$(-367417, +191719)$
FMA	$(-128858, +64225)$
Extended	$(-0.5000, +0.5000)$

Table 1: Error in computation of $a * d - b * c$.

or multiplying complex numbers,

$$(a + ib) * (d + ic) = (a * d - b * c) + i(a * c + b * d).$$

Computing $a * d - b * c$ straightforwardly using only double-precision multiplies and adds can result in very large relative errors. Consider, for example,

$$\begin{aligned} a &= 1.23456789012340000000 \times 10^{13} \\ b &= 2.34567890123450000000 \times 10^{13} \\ c &= 1.23456789012430000000 \times 10^{13} \\ d &= 2.34567890123540000000 \times 10^{13}. \end{aligned}$$

Computing $a * d - b * c$ with IEEE double-precision multiplies and subtracts, that is,

$$\begin{aligned} ad &= a \otimes d, \\ bc &= b \otimes c, \\ x &= ad \ominus bc, \end{aligned}$$

yields a result,

$$x = -9.9986838650880000 \times 10^{13}.$$

However, the correct double-precision result for this operation (*i.e.*, the nearest rounding of the exact result) is

$$x' = -9.9999990999999000 \times 10^{13}.$$

The relative error $|((x - x')/x')|$ is 1.3×10^{-4}. The error can also be expressed as 8×10^{11} ulps!

The problem is that $a * d$ and $b * c$ are both very similar numbers which cannot be expressed exactly with just 64 bits. To obtain their difference as an accurate double-precision number we actually have to perform the computations in extended precision. It is easy to obtain the correctly rounded value for the exact result of $a * d - b * c$ using extended precision:

1. First, we compute $(bch, bcl) = b * c$.
2. Then, we compute $(adh, adl) = a * d$.
3. Finally, we compute $(xh, xl) = (adh, adl) - (bch, bcl)$.

The result we want is in xh.

We performed a statistical test of the accuracy of this method. We used 1 million (a, b, c, d) sets, generated randomly with a uniform distribution between 0 and 1. We computed $a * d - b * c$ in three different ways: (i) using just the IEEE operations $\otimes$ and $\ominus$, (ii) computing $b \otimes c$ with an IEEE multiply and then $\mathrm{fl}(a * d - (b \otimes c))$ with an FMA, and (iii) using the extended-precision method described above. The range of errors, with respect to the exact result, for each approach are shown in Table 1.

As expected, the error for computation with extended precision is always between -0.5 ulp and $+0.5$ ulp. In contrast, the error using straightforward IEEE operations can be very large. The error from computing $a * d - b * c$ naively with FMAs is statistically smaller than just using multiplies and adds, but it can still be very large. (When we extended our test suite to 100 million sets we observed errors as large as 6.6×10^{15} ulps with the IEEE approach, and 2.3×10^{15} ulps with the FMA approach. A theoretical limit for the error is 9×10^{15} ulps.)

It is true that the numbers in Table 1 represent extreme cases and most of the time the error from computing $a * d - b * c$ with IEEE arithmetic is not that bad. However, the existence of extremely large errors is problematic because, for example, it prevents programmers from determining tight error bounds for their codes. Figure 2 presents some more results from our experiments. For each of the approaches (IEEE, FMA, and Extended) we present both a histogram of the number of occurrences of errors and a plot of the probability density function for the error. The histograms show, for a particular error value e in ulps, the number of results with an error in between $e - 0.5$ and $e + 0.5$ ulps. In the probability density case we plot $f(e)$, the normalized probability that the error lies in a small interval around e.

3.3 Computing the area of a needle-like triangle

Kahan [9] and Darcy [5] point out that the well-known Heron's formula,

$$A = \sqrt{s(s-a)(s-b)(s-c)},\ s = \frac{a+b+c}{2}, \tag{4}$$

for the area of a triangle (where A is the area and a, b, and c are the lengths of the sides) gives numerically inaccurate results when the triangle is 'needle-like' (*i.e.* $c \approx b \gg a$). In this section, we derive a numerically accurate formula using extended-precision arithmetic.

We take a direct approach. Lay out the triangle as in Fig. 3, where we know $c \geq b \geq a$. Then $A = \frac{1}{2}ch$, so that

$$16A^2 = (2ch)^2 = (2cb + 2cx)(2cb - 2cx).$$

Note that $2cx = c^2 + b^2 - a^2$, so that

$$\begin{aligned} A &= \frac{1}{4}\sqrt{((c+b)^2 - a^2)\,(a^2 - (c-b)^2))} \qquad (5)\\ &= \frac{1}{4}\sqrt{a_1 a_2 a_3 a_4}, \qquad (6) \end{aligned}$$

where

$$\begin{aligned} a_1 &= c + b + a \qquad (7)\\ a_2 &= c + b - a \qquad (8)\\ a_3 &= a + c - b \qquad (9)\\ a_4 &= a - (c - b). \qquad (10) \end{aligned}$$

There are two numerically distinct ways to associate each of these terms, leading to $2^4 = 16$ equivalent expressions for (6). These equivalent expressions, however, can produce different results when evaluated using floating-point arithmetic. Kahan [9] and Darcy [5] select the numerically stable variant,

$$A \approx \frac{1}{4}\sqrt{\bar{a}_1 \otimes \bar{a}_2 \otimes \bar{a}_3 \otimes \bar{a}_4}, \tag{11}$$

where

$$\begin{aligned} \bar{a}_1 &= c \oplus (b \oplus a), \qquad (12)\\ \bar{a}_2 &= c \oplus (b \ominus a), \qquad (13)\\ \bar{a}_3 &= a \oplus (c \ominus b), \qquad (14)\\ \bar{a}_4 &= a \ominus (c \ominus b). \qquad (15) \end{aligned}$$

Note that (11) is approximate, and while it gives the correct answer to within a few ulps [9], there are no given error bounds.

We now derive another approximate formula for A using the results we obtained for complex multiplication and 2×2 determinants. Let $u = c - b$, $v = c + b$, $x = a^2 - u^2$, $y = v^2 - a^2$, so that

$$\begin{aligned} A &= \frac{1}{4}\sqrt{(a^2 - u^2)(v^2 - a^2)}\\ &= \frac{1}{4}\sqrt{x^2 y^2}. \qquad (16) \end{aligned}$$

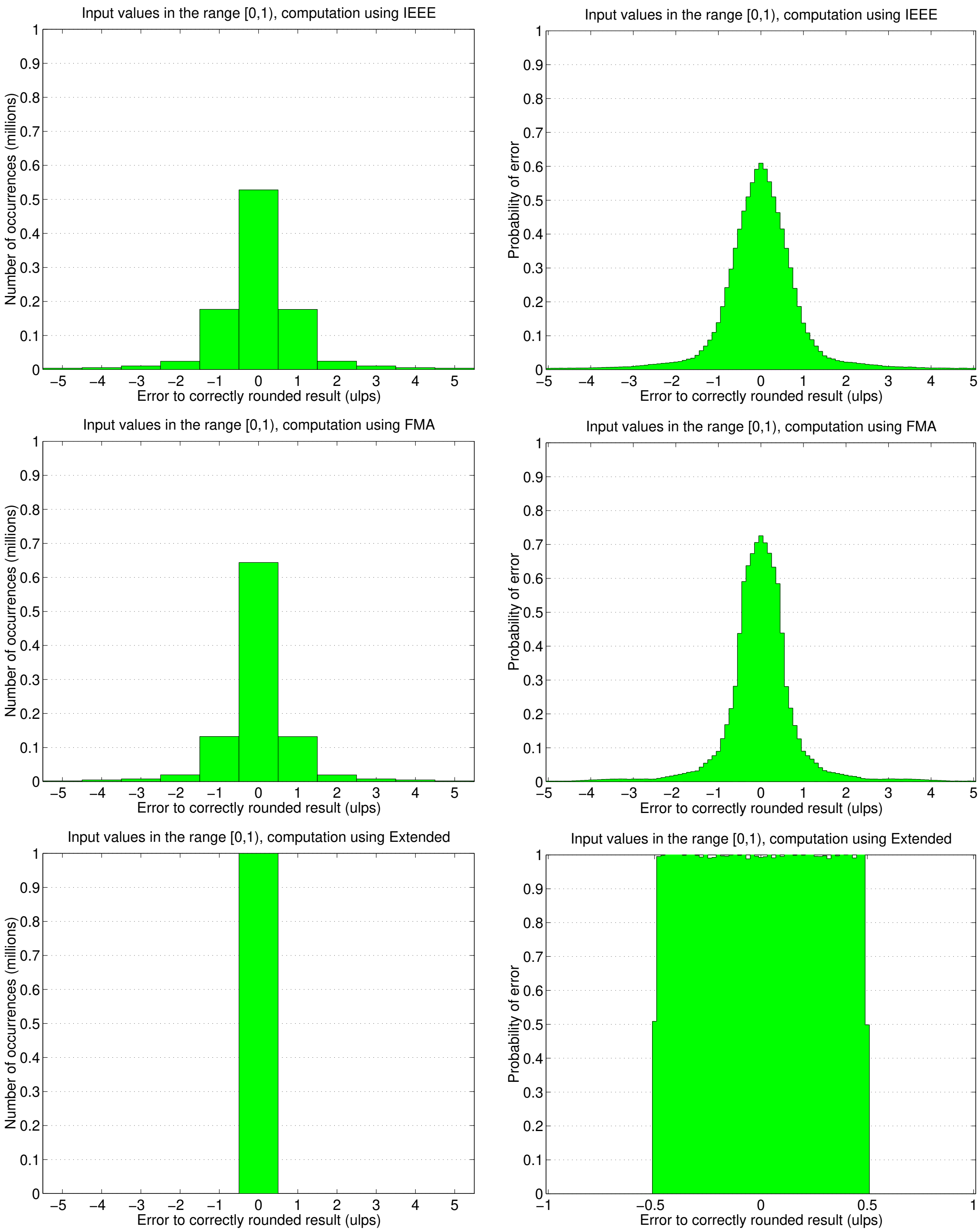

Figure 2: Summary of results for the $a * d - b * c$ case study.

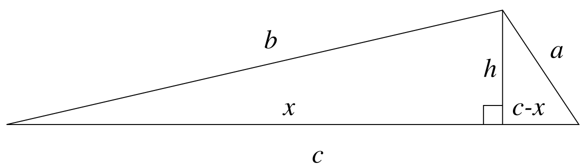

Figure 3: A needle-like triangle.

a	b	c	Our method (16)	Heron (4)	Kahan (11)
100	10^{16}	10^{16}	$\mathbf{9.8 \times 10^{-14}}$	$\mathbf{9.8 \times 10^{-14}}$	$\mathbf{9.8 \times 10^{-14}}$
2	10^{16}	10^{16}	$\mathbf{2.5 \times 10^{-17}}$	5.0×10^{15}	$\mathbf{2.5 \times 10^{-17}}$
3	10^{16}	10^{16}	$\mathbf{6.7 \times 10^{-17}}$	2.5×10^{15}	1.0
51	10^{16}	$10^{16} + 50$	**.12**	6.3×10^{15}	.88
11	10^{16}	$10^{16} + 10$	**.13**	5.7×10^{15}	1.1

Table 2: Ulp error in triangle-area calculations (smallest errors in **bold**).

With our approach, there is no need to choose one of various possible associations. The method of Section 3.2 is used to compute x and y. It can be shown that u is always computed exactly by IEEE arithmetic, but v is not always exact. Thus x is correctly rounded, while y is accurate to within a few ulps. This results in a value for A comparable in accuracy to Kahan's formula (11), and vastly better than Heron's formula (see Table 2).

The numerical accuracy of Kahan's formula (11) depends on the fact that $\bar{a}_4$ in (15) is computed exactly by IEEE arithmetic, and $\bar{a}_1$, $\bar{a}_2$, and $\bar{a}_3$ in (12-14) are accurate to within an ulp or two. This is shown by a novel roundoff error analysis [9].

If full extended-precision arithmetic is available, however, we can obtain an fully accurate answer for A. One can show that each of the a_i's can be represented exactly by a pair of double-precision numbers in canonical form. That is,

$$a_1 = a_{1h} + a_{1l}, \tag{17}$$
$$a_2 = a_{2h} + a_{2l}, \tag{18}$$
$$a_3 = a_{3h} + a_{3l}. \tag{19}$$

(It is trivially true for a_4 since (10) is exactly computed by IEEE floating point. The proof of (17-19) is based on simple inequalities relating to relationships satisfied by the sides of a triangle, such as $2c < a + b + c < 3c$.) In (19),

$$a_{3h} = a \oplus u,$$
$$a_{3l} = u \oplus (a \ominus a_{3h}).$$

One still needs to compute a_{1h}, a_{1l}, a_{2h}, and a_{2l} in (17 – 18). We will not give explicit formulas for these quantities, but note that a few double-precision IEEE operations are needed to compute them. Assuming they have been computed we have

$$A = \frac{1}{4}\sqrt{(a_{1h} + a_{1l})(a_{2h} + a_{2l})(a_{3h} + a_{3l})a_4}. \tag{20}$$

Note that equation (20) has no error; *i.e.* it is exact. The area can be computed in extended precision using equation (20) and then correctly rounded.

Before proceeding further, consider these special cases:

1. $c = b > a$: an isosceles triangle,
2. $c > b = a$: an isosceles triangle,
3. $c = b = a$: an equilateral triangle,

4. $c^2 = b^2 + a^2$: a right triangle.

In Case 1,

$$\begin{aligned} a_1 &= 2c + a, \\ a_2 &= 2c - a, \\ a_3 &= a, \\ a_4 &= a, \\ A &= \frac{1}{4} a \sqrt{a_1 a_2}. \end{aligned} \tag{21}$$

Note that $2c$ is exact but $a_1 = 2c + a$ and $a_2 = 2c - a$ are not. Here we have

$$\begin{aligned} a_{1h} &= 2c \oplus a, \\ a_{1l} &= a \oplus (2c \ominus a_{1h}), \\ a_{2h} &= 2c \ominus a, \\ a_{2l} &= a \oplus (2c \ominus a_{2h}). \end{aligned}$$

In Case 2,

$$\begin{aligned} a_1 &= c + 2b = 2b + c, \\ a_2 &= c, \\ a_3 &= c, \\ a_4 &= 2b - c, \\ A &= \frac{1}{4} c \sqrt{a_1 a_4}. \end{aligned} \tag{22}$$

Note that $2b$ is exact but $a_1 = 2b + c$ is not. Here we have

$$\begin{aligned} a_{1h} &= 2b \oplus c, \\ a_{1l} &= c \oplus (2b \ominus a_{1h}). \end{aligned}$$

In Case 3,

$$\begin{aligned} a_1 &= 3 * a, \\ a_2 &= a, \\ a_3 &= a, \\ a_4 &= a, \\ A &= \frac{1}{4} \sqrt{3} a^2. \end{aligned} \tag{23}$$

In Case 4, the area is computed exactly as

$$A = \frac{1}{2} ab. \tag{24}$$

However, while it is trivial to tell whether one of Cases 1–3 holds, in Case 4, we must accurately determine whether $c^2 = b^2 + a^2$. This can be done as follows. Since $u = c - b$ and $v = c + b$, $vu = c^2 - b^2$. Therefore $c^2 = b^2 + a^2$ iff $vu = a^2$. Recall that $u = c - b$ is exact in IEEE arithmetic, although $v = c + b$ may not be. Compute $(v_h, v_l) = c + b$. Then $v_h + v_l = c + b = v$, so that $vu = a^2$ iff $v_l u = a^2 - v_h u$. Now compute the left side with Section 2.2 and the right side with Section 3.2. Comparing these determines whether the triangle is right-angled.

Note that equations (21 – 24), like (20), have no error; *i.e.* they are exact.

Using the canonical form we can compute as accurately as we please, and in fact can compute the correctly rounded result. We think these equations (20 – 24) are the correct definitions of A if there must be an exact one. Also note that many FMAs and double-precision IEEE operations must be done to compute (20 – 24), and then to round them correctly.

4 Conclusions

We have described a canonical extended-precision floating-point representation and algorithms that allow us to compute and represent the exact result for the four basic floating-point operations $(+, -, *, /)$. Exact computation of the basic operations, and of dot-products, was the driving force behind the IBM ACRITH package [3]. Our desire to perform exact floating-point arithmetic arises from the Java requirement of producing the same answer in all machines. Using operational definitions of computations to enforce this requirement places significant restrictions on the optimizations allowed for Java programs. A conceptually more useful definition would be to require the exactly rounded answer for operations, or to establish limits for errors with respect to the exact answer.

Many modern processors implement a fused multiply-add instruction (FMA) which computes the double-precision rounded value for $a * b + c$. We have shown that exact results for the basic arithmetic operation $(+, -, *, /)$ can be computed with the help of the FMA instruction. Also, the FMA instruction greatly increases the performance of computing the exact $*$ and $/$ operations. Finally, equation (2) from number theory can be used to prove the Fundamental Theorem of Arithmetic. These are three strong justifications for Java to allow the use of the FMA instruction.

Errors in floating-point computations propagate very quickly. The computation of $a * d - b * c$ can produce errors in the range of 10^{15} ulps, even though the errors in the basic operations are less than or equal to 0.5 ulps. We need to use exact floating-point with extended precision just to get the correct double-precision answer.

About the authors

Fred G. Gustavson, IBM Research Division, Thomas J. Watson Research Center Fred Gustavson manages the Algorithms and Architectures group in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute, joining IBM Research in 1963. One of Dr. Gustavson's primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. He has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2 for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Outstanding Invention Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a fellow of the IEEE.

José E. Moreira IBM Research Division, Thomas J. Watson Research Center José Moreira received B.S. degrees in physics and electrical engineering in 1987 and an M.S. degree in electrical engineering in 1990, all from the University of Sao Paulo, Brazil. He received his Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1995. Dr. Moreira is a Research Staff Member in the Scalable Parallel Systems Department at the IBM Thomas J. Watson Research Center. Since joining IBM Watson in 1995, he has worked on various topics related to the design and execution of parallel applications. His current research activities include performance evaluation and optimization of Java programs, and scheduling mechanisms for the ASCI Blue project.

Robert F. Enenkel, Centre for Advanced Studies, IBM Toronto Laboratory Robert Enenkel is a Research Associate at the IBM Centre for Advanced Studies (CAS). Prior to joining CAS, he worked at the IBM Toronto Lab on the development of a C compiler and its math library, and developed parallel methods for random number generation for the Fortran and High Performance Fortran compilers. He received his B.Sc., M.Sc., and Ph.D. degrees from the University of Toronto, with thesis research in the area of numerical methods for the parallel solution of initial value problems for ordinary differential equations. His current interests include machine arithmetic, mathematical function libraries, and computational finance. Dr. Enenkel is a member of SIAM, the Society for Industrial and Applied Mathematics.

References

[1] Ramesh C. Agarwal, James W. Cooley, Fred G. Gustavson, James B. Shearer, Gordon Slishman, and Bryant Tuckerman. New scalar and vector elementary functions for the IBM System/370. *IBM Journal of Research and Development*, 30(2):126–144, March 1986.

[2] C. Baumhof. A new VLSI vector arithmetic coprocessor for the PC. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 210–215, Bath, England, July 19-21 1995.

[3] J. H. Bleher, A. E. Roeder, and S. M. Rump. ACRITH: High-accuracy arithmetic, an advanced tool for numerical computation. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, pages 318–321, Urbana, Illinois, June 4-6 1985.

[4] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. Semantics for exact floating point operations. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 22–26, Grenoble, France, June 26-28 1991.

[5] J. D. Darcy. Java's evolving floating-point support: The good, the bad, and the ugly. In Robert F. Enenkel, editor, *CASCON 1998 Workshop Report, Numerical Computing: Compiler and Library Support*, chapter 3.1. IBM Centre for Advanced Studies, 1999. Technical report TR-74.165-n.

[6] M. Daumas and D. W. Matula. Validated roundings of dot products by sticky accumulation. *IEEE Transactions on Computers*, 46(5):623–629, May 1997.

[7] Java Grande Forum. Making Java work for high-end computing. Technical Report TR-1, Java Grande Forum, 1998. URL: `http://www.javagrande.org/reports.htm`.

[8] IBM Corporation. *XL Fortran for AIX, User's Guide, Version 5 Release 1*, 1st edition, November 1997.

[9] W. Kahan. Miscalculating area and angles of a needle-like triangle. URL: `http://www.cs.berkeley.edu/~wkahan/Triangle.pdf`.

[10] A. Knöfel. Fast hardware units for the computation of accurate dot products. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 70–74, Grenoble, France, June 26-28 1991.

[11] Victoria Markstein, Peter Markstein, Tung Nguyen, and Steve Poole. Wide format floating-point math libraries. In *Proceedings of the Supercomputing '91*, pages 130–138, Albuquerque, New Mexico, November 18-22 1991.

[12] J. E. Moreira, S. P. Midikiff, and M. Gupta. A comparison of Fortran, C/C++, and Java for numerical computing. *IEEE Antennas and Propagation Magazine*, 40(5):102–105, October 1998.

[13] M. Müller, C. Rüb, and W. Rülling. Exact accumulation of floating-point numbers. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 64–69, Grenoble, France, June 26-28 1991.

[14] Institute of Electrical and Electronic Engineers. IEEE 754: Standard for binary floating-point arithmetic. Technical Report 754-1985, 1985.

[15] Sun Microsystems Inc. FDLIBM (freely distributable LIBM). URL: `http://www.netlib.org/fdlibm/readme`.

[16] Abraham Ziv and Moshe Olshansky. Math library for Java. URL: `http://www.alphaWorks.ibm.com/tech/mathlibrary4java`.