Available online at www.sciencedirect.com
SCIENCE DIRECT®
Nuclear Instruments and Methods in Physics Research A 559 (2006) 207–210
NUCLEAR INSTRUMENTS & METHODS IN PHYSICS RESEARCH Section A
www.elsevier.com/locate/nima

High precision numerical accuracy in physics research

Florent de Dinechin*, Gilles Villard

École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

Available online 12 December 2005

Abstract

Concerns arise that the current standard of double-precision floating-point may no longer be sufficient for today's large-scale numerical simulations. One approach to solve this problem will be to switch to a wider floating-point format: the upcoming quadruple-precision standard is introduced and compared to currently available software-based approaches. Another complimentary approach is to use mathematical and algorithmic techniques to improve the accuracy of large floating-point programs and the confidence in the quality of the result.

PACS: 02.70.−c; 07.05.Tp

Keywords: Numerical simulation; Accuracy; Precision; Floating-point; Quadruple-precision

1. Introduction

Floating-point (FP) is the most used representation of the reals. The standard for hardware FP in current workstations and mainframes is double-precision, a format with 53 bits of mantissa (translating to roughly 16 decimal digits) and 11 bits of exponent (or a range between 10^{-307} and 10^{307}). This is more than enough to represent most values manipulated by physicists. However, current computers are able to perform billions of FP operations each second, and some physical simulations will require trillions of FP operations. As each operation may introduce a tiny rounding error, there is increasing concern that the accumulation of these errors will render the final result meaningless [1].

Do such huge computations now require a switch to quadruple precision (or quad), a new format which would bring 32 decimal digits? Bailey [2] and Briggs,[1] authors of software packages for quadruple-like precision, have applied them to the study of many physical phenomena, from climate and supernova to muon decay. Thus software approaches are available now, and hardware quad can be expected in the near future. They are reviewed in Section 3, with an evaluation of their performance overhead.

Confidence in a floating-point program, however, should not depend only on the ability to increase the number of digits. Many other techniques exist to evaluate, improve or validate numerical code. They are surveyed in Section 4.

Before addressing these main topics, the next section reviews the floating-point features available in current computing systems.

2. Floating-point in 2005

The IEEE-754/IEC 60559 standard for floating-point arithmetic, adopted in 1985, has solved many of the reliability and consistency problems that could be experienced with earlier floating-point implementations. This standard defines the arithmetic: it specifies which behaviours are valid for one isolated FP operation in a computer. However, the translation of a numerical code to a succession of such operations depends on a language semantics, implemented by a compiler. Then, the choice among the various IEEE-754 valid FP behaviours is largely dependent on the operating system. This section reviews these questions.

*Corresponding author. Tel.: +33 4 72 72 85 03.
E-mail address: Florent.de.Dinechin@ens-lyon.fr (F. de Dinechin).
[1] http://members.lycos.co.uk/keithmbriggs/doubledouble.html

0168-9002/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.nima.2005.11.140

2.1. The IEEE-754 standard

This standard defines defines the usual floating-point formats (single, double, and a family of double-extended precisions) and precisely specifies the valid behaviours of the basic operators $+, -, \times, \div$ and $\sqrt{}$. An example of behaviour mandated by the standard is for the operators to return the FP number uniquely defined as the result of the exact operation applied to the arguments, then rounded to the nearest FP number. With such well-specified operators, one may prove properties of FP computations [3]. This standard was also key to enable FP portability: a computation may be performed on different computers and produce exactly the same results.

2.2. Floating point hardware in 2005

Virtually all recent computers are able to support the IEEE-754 standard efficiently through a combination of hardware and software. Current PC and workstation processors offer hardware double-precision operators for $+$, $-$, $\times$, and at least hardware assistance for $\div$ and $\sqrt{}$. Peak performance is typically between 2 and 4 double-precision FP operations per clock cycle for $+$, $-$ and $\times$, with much slower $\div$ and $\sqrt{}$ [4]. However, most processors go beyond this common denominator and offer faster or more accurate operators. Two examples follow.

Power/PowerPC and Itanium processors have fused multiply and add (FMA) operators: one instruction performs the operation $a \times b + c$, with only one rounding error with respect to the exact result. This is usually both faster and more accurate than a multiplication followed by an addition.

Pentium-compatible processors by Intel and AMD, as well as the HP/Intel Itanium processors, also provide hardware double-extended precision formats with 64 bits of mantissa (about 20 decimal digits) and an extended exponent range. The FP operators of these processors can be instructed to round to single, to double, or to double-extended.

2.3. Languages, compilers and processors

Due to this hardware variety, even with a computer supporting IEEE-754, being in control of the details of the FP computations of one's program (for instance to ensure portability) still requires some efforts.

Firstly, one has to know his *language*. To take just an example, in Fortran, an expression written $a/b*c/d$ may be computed either as $(a/b)*(c/d)$, or as $(a*c)/(b*d)$. A Fortran compiler may choose the parenthesing it judges the more efficient. These two expressions are mathematically equivalent, but do not lead to the same succession of rounding errors, and therefore possibly to different FP results.[2] However, the Fortran2003 standard (ISO/IEC 1539-1:2004(E)) also states that if parentheses are given in the source code, they should be respected: the programmer is in control of this question.

Secondly, one has to know his compiler, which is in charge of translating the program into a succession of operations. As an example, to comply with IEEE-754, a compiler should not use FMA operators,[3] except to perform additions $a \times 1 + b$ and multiplications $a \times b + 0$. Of course, the default behaviour of most compilers will be to try to fuse additions and multiplications which, again, usually improves both speed and accuracy. If one wants portability between, for instance, a PC (without FMA) and a PowerMac (with FMAs), one has to find the compiler options—they always exist—that prevent fusing $\times$ and $+$.

Finally, one has to know his operating system, which is in charge of setting some aspects of the processor behaviour, in particular rounding. For instance, the same C program, compiled by the same compiler in a way compliant with the C standard (ISO/IEC 9899:1999) may lead to different results on the same PC hardware under Solaris and Linux, because by default the first allows promoting double computations to double-extended (who will complain since it is more accurate and no slower?), and the second does not (probably for consistency with Solaris systems without double-extended hardware). Again, the programmer is in control: a call to a standard C99 function (which is a call to the OS) will set the system to a common behaviour.

These examples show that standard compliance is good for portability, but usually bad for performance and sometimes accuracy. The important thing is that, thanks to existing standards, it is possible for the programmer to control to the last bit the behaviour of every last FP operation of his program.

3. Software and hardware for quadruple precision

The quadruple precision format, (hereafter quad), is already part of the IEEE Standard for Shared Data Formats 1596.5-1993. This standard does not specify the operations, which should be in the upcoming revision of IEEE-754. Quoting the draft of this revision as of August 2005,[4] quad is "a 128-bit quadruple precision with 112 fraction bits and implicit integer bit" whose rationale is to "match existing hardware and software implementations, and discourage undesirable alternatives such as double-double" (this last point is explained below). These existing software implementations are integrated in compilers from companies like Sun, HP and Intel. To our knowledge, the only hardware implementation can be found in some IBM mainframes, and very little documentation exists on it.

[2] In the development of the LHC@Home project, such an expression, appearing identically in two points of a program, was compiled differently. This unexpected behaviour would very rarely break the distributed simulation.

[3] The next revision of the IEEE-754 standard should include the FMA.

[4] Available at http://754r.ucbtest.org/

Table 1
Cost of double-FP operations

Double-FP operation	Cost in FP operations
+, −	8 +, one <
×	9×, 15 + (or 7 FMA)
/	2/, 8×, 17+ (or much less with FMAs)

3.1. Quad versus double-double

An alternative to this format is the double-double format, in which the unevaluated sum of two FP numbers is used to represent a number with twice the precision, as $1.234 \times 10^5 + 5.678 \times 10^1$ could be used to represent the number 1.2345678×10^5. Algorithms for computing on such double-FP numbers were given by Dekker [5] and have been improved since then [6,7]. Some Fortran `REAL*16` implementations have used this format internally. The cost of the basic operations is summarized in Table 1 (the actual cost in cycles is very machine dependent, and many tricks can be used to improve it in special cases).

This idea can be extended to arbitrary multiple precision [8,9], but with current hardware, the simpler approach where the mantissa is stored in an array of integers (see for instance MPFR[5] based on GMP) is more efficient for precisions larger than quad.

Performance of software quad is also much lower than that of hardware double, but it is difficult to compare quad and double-double: the available quad implementations are not libraries, but directly integrated in compilers. A library approach suffers from the overhead of function calls (several tens of cycles on modern systems, or the equivalent of 5–10 FP operations), and prevents many optimizations which a compiler may perform.

Both software quad or software double-doubles have their pros and cons. Conversions between double and double-double are trivial, whereas conversion between double and quad has a cost. Quad has a larger exponent range that double-double, and 113 bits of significant versus $53 + 53 = 106$ for double-double. A more fundamental problem with double-double is that blindly replacing all the double operations of a working program with double-double may lead to a non-working program (here "working" means for instance without overflows and underflows). This does not happen when going from double to quadruple (of course a non-working program may be converted to a working one). Such questions are important enough for the IEEE-754 revision committee to describe the double-double format as "indesirable".

However, well controlled use of double-double techniques allows for very efficient implementation. An example is the quad elementary functions libraries by HP and Intel, which internally use double-double-extended arithmetic (and not the quad operations developed by these same companies). The published performance of these functions [10] is roughly ten times slower than the corresponding double-precision function. This is consistent with the previous table (taking into account that a quad function must use a more accurate algorithm than a double function).

As a conclusion, one should expect the performance of optimized software quad or double-double to be in the order of 10 times slower than that of double precision.

3.2. Tradeoffs for hardware quad

Let us now consider hardware quad implementations. The algorithmic space and time complexities of current implementations of FP operators [4] are summarized in Table 2 (the reader should be aware that the constraints of current deep submicron, multi-gigahertz VLSI technology add many other dimensions of complexity).

One would conclude from this table that going from double to quad means roughly doubling the silicon cost of the adder, quadrupling that of the multiplier, and probably adding one cycle to the latency of each operation. With the silicon budget of current high-end microprocessors, and considering that the FP unit rarely occupies more than $\frac{1}{20}$th of the area, this could be implemented with current technology, and therefore will happen as soon as the market justifies it.

However, a hardware quad operation will still be more expensive than a double one. Consider the evolution of FP hardware in the x86 family. The initial double (extended) operators were supplemented with the SSE unit, which was (roughly speaking) able to perform either one double-precision operation, or two single-precision operations in the same time. Then the SSE2 unit was added with a 128-bit data path which may perform either two double-precision operations, or four single-precision operations. The next step could be for this unit to support quad operation. Indeed Akkas and Schulte have designed a unit capable either of quad multiplication in three cycles, or of two parallel double multiplications in two cycles [11]. Here the raw performance ratio of quad to double is equal to 3. They report a cost of 146,000 gates (versus 70,000 for a double-precision multiplier), and a delay of 6.11 ns (versus 4.68 ns). All this is consistent with Table 2. Note that the FP unit is only one aspect of hardware quad in a processor: another challenge is to feed it with data, requiring 128-bit

Table 2
Space and time complexity of FP operations (n is the mantissa size)

Operation	Space	Time
+, −	$n\log(n)$	$\log(n)$
×	n^2	$\log(n)$

[5] http://www.mpfr.org/

registers and buses running at full speed. However, this challenge is already met e.g. in the SSE2 extensions.

The conclusion here is that it is very probable that hardware quad, when it comes, will be at least twice slower than double precision. Another desirable—but currently inaccessible [12]—tradeoff would be a quad unit also able of operations on double-precision complex numbers and intervals (see below).

4. Improving confidence in FP programs

Blindly replacing all the double-precision operations with quad-precision operations may solve some accuracy problems, but not all. It may even increase the confidence in a nevertheless wrong result, if for example the discretization of the problem is inadequate and introduces methodological errors much larger than the rounding errors.

To start with, rounding is not always the culprit of a loss of accuracy. In the subtraction $1.23456 - 1.23455 = 1.00000 \times 10^{-5}$, the zeroes of the mantissa of the result correspond to digits absent from the two inputs, and hence hold no valid information: the result has only one valid digit left! However, this subtraction was exact, without any rounding error. This phenomenon is called cancellation. This example and others can be found in a FP tutorial by Goldberg [3] (versions of which are freely available on the web) or on the web page of W. Kahan.[6]

Physical knowledge may come to the rescue here: as most FP variables can be related to physical or geometrical quantities, the programmer may check whether a cancellation has no effect on the simulation (because it corresponds to a very small value of some physical quantity, whose physical influence will be negligible anyway), or in contrary if it should be considered seriously (if for instance it affects the angle of a vector describing the motion of a particle, so that its effect will be amplified by the subsequent motion of the particle). Physical knowledge will also help decide where parentheses should go in a Fortran arithmetic expression to maximize the accuracy of its evaluation, etc.

More general error analysis techniques help quantify in a mathematical way how equations are sensitive to roundoff errors [13]. Other mathematical techniques can be implemented as automatic tools. According to a survey by Kahan,[7] the safest of these approaches is interval arithmetic [14], which replaces all the FP variables with intervals guaranteed to hold the corresponding real. It is possible to almost automatically convert a FP program to an interval one. However, this will likely expose places where accuracy is lost, but not repair the program: this still requires a programmer's knowledge, and maybe algorithmic changes: for the same computation, different algorithms will have different numerical behaviours. A typical example is the sum of a large array of numbers: its accuracy may be improved by sorting the numbers first, then summing them in increasing order of magnitude. Another solution is the use of accurate sum algorithms [15].

All these techniques and tools are complimentary (see for instance a comparison in [16]). At the very least, awareness of some of these techniques teaches the limits of floating-point and how to get around them. They will also help the performance-conscious programmer to use slower quadruple-precision computing only when needed.

References

[1] U. Kulisch, Does the computer result really solve the problem? Computer Zeitung, September 6, 2004 (in German).
[2] D.H. Bailey, Comput. Sci. Eng. 7 (3) (2005) 54.
[3] D. Goldberg, ACM Comput. Surveys 23 (1) (1991) 5.
[4] M.D. Ercegovac, T. Lang, Digital Arithmetic, Morgan Kaufmann, Los Altos, CA, 2003.
[5] T.J. Dekker, Numer. Math. 18 (3) (1971) 224.
[6] D. Knuth, The Art of Computer Programming, vol. 2, Addison-Wesley, Reading, MA, 1973.
[7] S. Linnainmaa, ACM Trans. Math. Software 7 (3) (1981) 272.
[8] R.P. Brent, ACM Trans. Math. Software 4 (1) (1978) 57.
[9] Y. Hida, X.S. Li, D.H. Bailey, Algorithms for quad-double precision floating-point arithmetic, in: 15th IEEE Symposium on Computer Arithmetic, June 2001, pp. 155–162.
[10] Peter Markstein, A fast quad precision elementary function library for Itanium, in: Real Numbers and Computers, 2003, pp. 5–12.
[11] A. Akkas, M.J. Schulte, A quadruple precision and dual double precision floating-point multiplier, in: Euromicro Digital System Design, IEEE Computer Society, Silver Spring, MD, 2003.
[12] N. Holmes, W. Kahan, D. Zuras, Computer (2005) 7.
[13] N.J. Higham, Accuracy and Stability of Numerical Algorithms, SIAM, Philadelphia, PA, 1996.
[14] R.E. Moore, Interval Analysis, Prentice-Hall, Englewood Cliffs, NJ, 1966.
[15] Y. He, C.H.Q. Ding, J. Supercomputing 18 (2001) 259.
[16] H. Hasegawa, Utilizing the quadruple-precision floating-point arithmetic operation for the Krylov subspace methods, in: SIAM Linear Algebra, 2003.

[6] http://www.cs.berkeley.edu/~wkahan/

[7] www.cs.berkeley.edu/~wkahan/Mindless.pdf