Quadruple-precision summation in openQCD

Martin Lüscher February 2018; last revised October 2019

1. Introduction

On very large lattices, the numerical evaluation of scalar products of quark fields (and of other sums over all lattice sites) is potentially affected by huge rounding effects. If the summands are 32-bit floating-point numbers, the problem is, in practice, avoided by calculating the sum using 64-bit data and arithmetic. Quadruple-precision arithmetic is however not natively supported on the current generation of HPC systems. A summation of 64-bit floating-point numbers with quadruple precision (i.e. with relative errors per addition on the order of 10^{-32}) therefore requires a software implementation of quadruple-precision data and arithmetic.

While there exist several software libraries supporting quadruple-precision arithmetic, their use in openQCD would, at present, imply a loss of portability. Moreover, rather than a full-fledged quadruple-precision math library only an accurate evaluation of lattice sums is required.

In this note the implementation chosen in openQCD is described. It is based on a representation of quadruple-precision numbers through pairs of double-precision numbers and the algorithms published by Dekker [1] many years ago. Much of the material covered here can also be found in chapter 4 of a book of Knuth [2] as well as in a more recent publication by Shewchuk [3]. The implementation assumes the IEEE 754 standard for double-precision floating-point data and arithmetic, but is otherwise entirely portable.

2. Double-precision floating-point numbers

In the following some basic knowledge of the IEEE 754 standard for double-precision floating-point arithmetic is assumed. See ref. [4], for example, for a very readable presentation of the relevant conventions.

2.1 Representable numbers

According to the standard, a double-precision floating-point number occupies 8 bytes (64 bits) in the memory of the computer. Only a finite set of numbers can thus be represented, among them all numbers x of the form

$$x = \pm 2^p \times \left\{1 + d_1 2^{-1} + d_2 2^{-2} + \ldots + d_{52} 2^{-52}\right\}, \tag{2.1}$$

where p is an integer in the range

$$-1022 \leq p \leq 1023 \tag{2.2}$$

and d_k, $k = 1, \ldots, 52$, an arbitrary sequence of 0's and 1's. These numbers have magnitude less than or equal to

$$2^{1024}\left\{1 - 2^{-53}\right\} \simeq 1.8 \times 10^{308}, \tag{2.3}$$

while the smallest positive number is

$$2^{-1022} \simeq 2.2 \times 10^{-308}. \tag{2.4}$$

In particular, all non-zero integers n satisfying $|n| \leq 2^{53}$ are of the form (2.1) and are thus representable.

Any representable number x is stored in memory as a string $b_1 b_2 \ldots b_{64}$ of bits b_k. If x is of the form (2.1) the bits are determined as follows:

(a) The first bit b_1 is 0 for positive numbers and 1 for negative ones.

(b) The next eleven bits, $b_2 \ldots b_{12}$, represent the exponent p. Namely, one first constructs an integer

$$e = b_2 2^{10} + b_3 2^9 + \ldots + b_{12} 2^0, \tag{2.5}$$

the *biased exponent*, and then sets

$$p = e - 1023. \tag{2.6}$$

Note that e ranges from 0 to 2047. The numbers of the form (2.1) (i.e. the so-called normalized ones) have $1 \leq e \leq 2046$ and p is thus restricted to the range (2.2).

(c) The remaining 52 bits $b_{13}b_{11}\ldots b_{64}$ are exactly equal to $d_1d_2\ldots d_{52}$.

The number 0 is not of the form (2.1) and thus requires special attention. According to the IEEE standard, both, $000\ldots0$ and $100\ldots0$, are valid bit-string representations of 0. One may think of these as ± 0.

A further set of representable numbers are the so-called subnormal numbers. They are of the form

$$x = \pm 2^{-1022} \times \left\{ d_1 2^{-1} + d_2 2^{-2} + \ldots + d_{52} 2^{-52} \right\}, \tag{2.7}$$

where not all digits d_k are equal to zero. The corresponding bit string has biased exponent $e = 0$, while the digits d_k are stored as described above.

The numbers considered so far exhaust all possible bit strings with biased exponent $e < 2047$. Bit strings with $e = 2047$ do not represent real numbers. They are reserved for special purposes and will not be discussed here.

2.2 Arithmetic operations

The arithmetic operations $+, -, *, /$ performed by the computer are denoted by the round symbols $\oplus, \ominus, \otimes, \oslash$. In the case of the addition of two representable numbers u and v, for example, $u \oplus v$ is the representable number delivered by the processor as the sum of u and v. Evidently $u \oplus v$ cannot in general be equal to the exact sum $u + v$.

The IEEE 754 standard specifies that

$$u \oplus v = \text{round}(u + v), \tag{2.8}$$

where round(x) is a representable number near x. In particular, round$(x) = x$ if x is representable. Five different rounding rules are tolerated by the standard. The default one is to round to the nearest representable number, and if x happens to be exactly halfway between two successive representable numbers, round(x) is chosen to be the one with the last binary digit d_{52} equal to zero. This rule is often referred to as "round to nearest with ties to even".

Rounding away from zero in the case of ties as well as rounding to 0, $+\infty$ or $-\infty$ is also permitted by the standard. While these alternative rules may be supported by the current processors via special flags or instructions, their use normally requires some particular action to be taken by the programmer or the user.

The same rounding rules are also adopted for the other arithmetic operations. They do not apply, of course, if an exception such as exponent overflow occurs during the computations. Note that an exponent underflow cannot occur when adding and subtracting representable numbers. All such numbers are integer multiples of 2^{-1074} and the same must therefore be true for the (exact) sum of any two of them. Now if the sum is greater or equal to 2^{-1022}, rounding yields a representable number of the form (2.1). And if the sum is less than 2^{-1022}, it must be equal to zero or to a subnormal number, where rounding has no effect.

From the above it follows that

$$u \oplus v = (u + v)(1 + \eta), \qquad |\eta| \leq \epsilon, \tag{2.9}$$

where

$$\epsilon = 2^{-53} \simeq 1.1 \times 10^{-16} \tag{2.10}$$

if the default rounding is used.

3. Quadruple-precision summation

In the following, the double-precision floating-point data and arithmetic are assumed to comply with the IEEE 754 standard, to the extent described above and with the default rounding rules.

3.1 Quadruple-precision numbers

A *quadruple-precision number* is, in this note, an ordered pair (x, y) of representable double-precision numbers such that $x = x \oplus y$. The *value* of (x, y) is the exact sum $x + y$.

It follows from this definition that apart from $(0, 0)$ all other quadruple-precision numbers have $x \neq 0$. y must be equal to zero if x is a subnormal number, while in all other cases the bound

$$|y| \leq 2^{p-53} \tag{3.1}$$

holds, where p is the exponent of x [cf. eq. (2.1)]. In other words, y represents the less significant bits of the number. The value of (x, y) determines x and y uniquely, since $x = \mathrm{round}(x + y)$.

Any number of the form

$$\pm 2^p \times \left\{1 + d_1 2^{-1} + d_2 2^{-2} + \ldots + d_{105} 2^{-105}\right\} \tag{3.2}$$

with exponent in the range $-969 \leq p \leq 1023$ and arbitrary binary digits $d_1, \ldots, d_{105}$ is equal to the value of a quadruple-precision number. All these numbers are thus machine-representable in the way described here.

3.2 Exact addition of double-precision numbers

The exact sum $u + v$ of two double-precision numbers u, v may be represented by a quadruple-precision number. By definition

$$w = u \oplus v \tag{3.3}$$

is representable and (as shown in refs. [2] and [3], for example) the exact remainder

$$r = u + v - w \tag{3.4}$$

is representable too. Furthermore, since $w = \mathrm{round}(u + v) = \mathrm{round}(w + r) = w \oplus r$, it follows that (w, r) is a quadruple-precision number with value $u + v$. Symbolically one may write

$$(w, r) = u + v, \tag{3.5}$$

even though this notation is an abuse of language. No confusion will however arise in the following.

The remainder r can be calculated through

$$r = (u \ominus u') \oplus (v \ominus v''), \tag{3.6}$$

where

$$u' = w \ominus v, \tag{3.7}$$

$$v'' = w \ominus u'. \tag{3.8}$$

Alternatively, if $u = 0$ or $v = 0$ or if the exponent of u is greater or equal to the exponent of v, the simpler formula

$$r = v \ominus (w \ominus u) \tag{3.9}$$

may be employed. Evidently these relations are only meaningful if no exponent overflow occurs during the computations. Since additions of representable numbers do not underflow, there is in fact no other instance where eqs. (3.6) and (3.9) would not apply.

3.3 Addition of quadruple-precision numbers

Let (u, r) and (v, s) be two quadruple-precision numbers. In the following lines an algorithm is described that yields a quadruple-precision number (w, t), whose value coincides with the sum of the values of (u, r) and (v, s) up to a relative error of order ϵ^2.

First the exact sums

$$(a, b) = u + v, \tag{3.10}$$

$$(c, d) = r + s, \tag{3.11}$$

are computed. (w, t) is then obtained through

$$(e, f) = a + (b \oplus c), \tag{3.12}$$

$$(w, t) = e + (f \oplus d). \tag{3.13}$$

A rounding error may occur when calculating $b \oplus c$ and $f \oplus d$, but the effect of these errors is bounded by

$$z = (x + y)(1 + \eta), \qquad |\eta| \leq 5\epsilon^2, \tag{3.14}$$

where x, y and z denote the values of (u, r), (v, s) and (w, t), respectively. Moreover, eq. (3.9) may be used to compute the remainders f and t so that

$$f = (b \oplus c) \ominus (e \ominus a), \tag{3.15}$$

$$t = (f \oplus d) \ominus (w \ominus e). \tag{3.16}$$

In total one thus needs 20 additions to compute (w, t). And if $r = 0$ (so that (u, r) represents the double-precision value u), the quadruple-precision addition requires 10 double-precision additions.

4. Exact multiplication of double-precision numbers

Similarly to the sum of two double-precision numbers, their product can be exactly represented by a quadruple-precision number. One can then easily devise algorithms for products of quadruple-precision numbers with relative errors of order ϵ^2.

4.1 Splitting algorithm

Let u be a representable double-precision number. The splitting algorithm decomposes u into an exact sum

$$u = u_1 + u_2 \tag{4.1}$$

of two double-precision numbers with 26-bit mantissa (i.e. numbers of the form (2.1) with vanishing bits $d_{26}, \ldots, d_{52}$). The algorithm

$$\begin{aligned} a &= (2^{27} + 1) \otimes u, \\ b &= a \ominus u, \\ u_1 &= a \ominus b, \\ u_2 &= u \ominus u_1, \end{aligned} \tag{4.2}$$

was invented by Dekker and a relatively simple proof appeared in ref. [3]. Moreover, the splitting can be shown to be such that $|u_1| \geq |u_2|$.

4.2 Product rule

Now let u and v be two double-precision numbers and

$$u = u_1 + u_2, \quad v = v_1 + v_2, \tag{4.3}$$

their decomposition in 26-bit numbers according to the splitting algorithm. Clearly, the products $u_i v_j$ are all representable and therefore exactly equal to $u_i \otimes v_j$. In order to obtain the representation

$$uv = x + y \tag{4.4}$$

by the quadruple-precision number (x, y), it thus remains to sum these four products exactly.

As it turns out, x and y can be computed through

$$x = u \otimes v,$$

$$c = (u_1 \otimes v_1) \ominus x,$$

$$d = (u_2 \otimes v_1) \oplus c,$$

$$e = (u_1 \otimes v_2) \oplus d,$$

$$y = (u_2 \otimes v_2) \oplus e \tag{4.5}$$

(see sect. 2.5 of ref. [3] for a proof of the correctness of these rules).

5. Programs

5.1 Elementary functions

The program file `utils/qsum.c` in the `modules` directory provides the functions

```
void acc_qflt(double u,double *qr),
void add_qflt(double *qu,double *qv,double *qr),
void global_qsum(int n,double **qu,double **qr),
void scl_qflt(double u,double *qr),
void mul_qflt(double *qu,double *qv,double *qr).
```

The first one adds the double-precision number `u` to the quadruple-precision number

```
(qr[0],qr[1]),
```

while the second adds the quadruple-precision numbers `qu,qv` and assigns the result to `qr`. The third function sums each of the quadruple-precision numbers

```
(qu[k][0],qu[k][1]), k=0,...,n-1,
```

over all MPI processes and assigns the sums to the quadruple-precision numbers

```
(qr[k][0],qr[k][1]), k=0,...,n-1.
```

All additions are performed in quadruple precision using the formulae in sect. 3.

Calling the function `scl_qflt(u,qr)` multiplies ("scales") the quadruple-precision number `qr` by `u`, while the function `mul_qflt(qu,qv,qr)` assigns the product of `qu`

and `qv` to `qr`. All these operations are performed in quadruple precision, the relative error being at most $6\epsilon^2$.

In these elementary functions, the quadruple-precision numbers are represented by `double` arrays of length 2 (or more). The programs take it for granted that the arrays have these many elements and that the input quadruple-precision numbers are in the proper condition. Typically a large sum is accumulate by setting

```
qr[0]=0.0;
qr[1]=0.0;
```

before the summation loop and adding the summands in the loop using `acc_qflt()` or `add_qflt()` as appropriate. If so desired, the results can then be summed over all MPI processes using `global_qsum()`.

5.2 Quadruple-precision types

In `su3.h` the data types

```
typedef struct
{
  double q[2];
} qflt;
```

and

```
typedef struct
{
  qflt re,im;
} complex_qflt;
```

for quadruple-precision real and complex numbers are defined. Programs for scalar products, actions, etc. use these types to return quadruple-precision results.

5.3 Ensuring IEEE 754 compliance

On a given machine, and with the chosen compiler and compiler options, the IEEE 754 compliance may not be guaranteed. Computers supporting the x86 instruction set are a particularly delicate case, because these machines support 80-bit floating-point arithmetic in addition to the compliant arithmetic.

If the compiler options `-Dx64` or `-DAVX` are set on x86 machines, openQCD executes the quadruple-precision arithmetic using SSE instructions. The non-compliant x86 instructions are then safely avoided. Compliance can usually also be enforced

by choosing a combination of compiler flags. The GCC compiler, for example, supports the option `-mfpmath=sse`, which guarantees that all floating-point arithmetic is executed on the SSE registers (this is anyway the default on x86-64 machines).

As a safety measure, the openQCD main programs check the IEEE 754 compliance at the beginning of the program by calling the function

```
void check_machine(void)
```

(see `modules/utils/mutils.c`). The correct functioning of the quadruple-precision programs can also be thoroughly checked by executing the program `check3.c` in the `devel/utils` directory.

6. Error propagation in large sums

Let $x_1, \ldots, x_n$ be a series of double-precision numbers, whose sum is to be computed in quadruple precision. In practice each MPI process first sums the locally available values, using the function `acc_qflt()`, and the complete sum is then obtained by calling the function `global_qsum()`. Only the arithmetic (rounding) errors accumulated when the sum is calculated are discussed in this section, i.e. inaccuracies in the data are not taken account.

6.1 Local summation

If $y_1, \ldots, y_m$ are the local values, the partial sums $s_1, \ldots, s_m$ determined by the MPI process are given recursively through

$$\begin{aligned} &s_1 = y_1, \\ &s_{k+1} = (s_k + y_k)(1 + \eta_k), \quad k = 1, \ldots, m-1, \end{aligned} \tag{6.1}$$

where the errors satisfy

$$|\eta_k| \le 5\epsilon^2. \tag{6.2}$$

The recursion (6.1) implies

$$s_m = \sum_{k=1}^{m} (1 + \delta_k) y_k, \qquad \delta_k = \prod_{j=k}^{m} (1 + \eta_j) - 1, \tag{6.3}$$

and s_m thus approximates the exact sum up to the deviation

$$\delta s = \sum_{k=1}^{m} \delta_k y_k. \tag{6.4}$$

Note that the errors

$$|\delta_k| \leq (1+5\epsilon^2)^{m-k+1} - 1 = 5(m-k+1)\epsilon^2 + \ldots \tag{6.5}$$

are of order $m\epsilon^2$ if $m\epsilon^2 \ll 1$, a condition which, in practice, is usually satisfied by a wide margin.

If the summands y_k are all non-negative, it follows from these equations that the sum is obtained with a relative error of at most $5m\epsilon^2$. In general, the absolute error is bounded by

$$|\delta s| \leq 5m\epsilon^2 \sum_{k=1}^{m} |y_k| \tag{6.6}$$

(up to terms of order $m^2\epsilon^4$) and no statement can be made on the relative error. The rigorous bound (6.6) is however often far too pessimistic, because rounding errors tend to have varying sign and therefore partially cancel in large sums.

6.2 Global summation

Suppose now there are n_p MPI processes and that the local sums $z_1, \ldots, z_{n_p}$ have already been computed. The program `global_qsum()` sums these numbers using the `MPI_Reduce()` library function. Usually this implies that the sum is performed in a hiearchical manner by summing pairs of numbers, then pairs of these sums, and so on. In the first step, for example, z_{2k} is added to z_{2k-1} for all $k = 1, \ldots, n_p/2$. In each step the summation is preceded by a communication of the numbers obtained in the previous step to the MPI processes, where the pair sums are to be performed.

One needs $\lceil \log_2(n_p) \rceil$ steps of this kind to complete the calculation. The accumulated rounding errors are then at most $5\epsilon^2$ times the number of steps. In practice these errors thus tend to be totally negligible with respect to the errors accumulated in the course of the calculation of the local sums.

References

[1] T. J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math. 18 (1971) 224

[2] D. E. Knuth, Semi-Numerical Algorithms, *in*: The Art of Computer Programming, vol. 2, 2nd ed. (Addison-Wesley, Reading MA, 1981)

[3] J. R. Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete & Computational Geometry 18 (1997) 305

[4] Numerical Computation Guide, SPARCompiler Fortran 2.0, (Sun Microsystems, Mountain View, 1992)