Компьютерная математика

УДК 004

Р.А. ЮЩЕНКО, А.К. ЮЩЕНКО

ОСОБЕННОСТИ АРИФМЕТИКИ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ В СОВРЕМЕННЫХ КОМПЬЮТЕРАХ

Представлен краткий обзор чисел с плавающей запятой и соответствующей арифметики. Подчеркнуты особенности, которые отличают ее от арифметики действительных чисел и часто являются причиной существенных проблем при численном решении задач на компьютере. Эти проблемы усиливаются на больших объемах данных и параллельных вычислениях. Несмотря на все это, минимизировать и даже гарантировать точность решений возможно даже без существенных потерь в производительности.

Введение. В 60-е и 70-е годы прошлого века компьютеры в основном использовались для решения сложных вычислительных задач моделирования. В таких расчетах используются действительные числа и соответствующая арифметика. Но представление любого действительного числа в общем случае требует неограниченной памяти, что на компьютерах недостижимо. Это стимулировало исследования методов представлений действительных чисел, используя конечное число двоичных разрядов. Наиболее распространенным для вычислительных задач и по сегодняшний день является представление с плавающей запятой, в котором обеспечивается компромисс диапазона и точности. От способа представления числа зависят и свойства арифметики, поэтому разнообразие различных представлений в эти годы создало проблемы совместимости программ и аппаратуры. Одни и те же программы на компьютерах разных производителей возвращали разный результат, иногда между решениями не было ничего общего, иногда ни одно из них вообще не имело физического смысла. Поэтому в 70-е и 80-е годы была проведена огромная работа по стандартизации чисел с плавающей запятой, в результате которой разработан и до сих пор существует практически без изменений стандарт IEEE 754. В результате, с середины 80-х годов работа с числами с плавающей запятой существенно упростилась, но для гарантии точности или даже осмысленности решений, по-прежнему необходимо проводить специальные исследования в аксиоматике машинной арифметики [1].

Свойства компьютерной арифметики особенно ярко стали проявляться в последние годы. Это происходит по нескольким причинам. Во-первых, экспоненциальный рост производительности и объемов памяти компьютеров (закон Мура) привел к тому, что масштабы обрабатываемых данных возросли на несколько порядков. В результате, погрешность округления, не существенная для маленьких задач, снова стала значительной при обработке больших объемов данных [2]. Во-вторых, производители высокопроизводительной техники и компиляторов часто отступают от стандартов для обеспечения скорости. Это характерно для режимов «быстрой математики» компиляторов и для графических ускорителей [3]. В-третьих, и здесь мы имеем дело с фундаментальной проблемой, арифметика с плавающей запятой отличается от арифметики действительных чисел: действительные числа формируют закрытое множество, числа с плавающей запятой – нет, свойства ассоциативности и дистрибутивности в арифметике с плавающей запятой не выполняются и т. д. [4]. Следствия таких побочных эффектов многократно усиливаются в параллельных вычислениях, поскольку параллелизм как таковой подразумевает реорганизацию вычислений за счет применения эквивалентных преобразований к алгоритмам. Эквивалентные преобразования в обычной арифметике далеко не всегда являются эквивалентными в компьютерной арифметике.

Основные понятия. Множество целых чисел бесконечно, но всегда можно подобрать такое число разрядов (бит), чтобы представить любое целое число, возникающее при решении конкретной задачи. Множество действительных чисел не только бесконечно, но еще и непрерывно, поэтому, независимо от числа разрядов, неизбежно возникнут числа, которые не имеют точного представления. Числа с плавающей запятой – один из возможных способов представления действительных чисел и состоит из набора отдельных разрядов, условно разделенных на знак, порядок и мантиссу. Порядок и мантисса – целые числа, которые вместе со знаком дают представление числа с плавающей запятой, как показано на рис. 1.

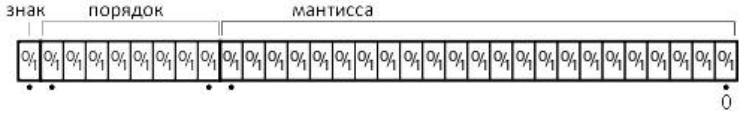


РИС. 1. Представление числа с плавающей запятой

Математически это записывается так: $(-1)^s \times M \times B^E$, где s – знак, B – основание, E – порядок, а M – мантисса.

Основание определяет систему счисления разрядов. Математически доказано [5], что числа с плавающей запятой с базой $B = 2$ (двоичное представление) наиболее устойчивы к ошибкам округления, поэтому на практике встречаются только базы 2 и, реже, 10. Для простоты, не теряя общности, для всех примеров будем полагать $B = 2$. Таким образом, формула числа с плавающей запятой будет иметь вид:

$$(-1)^s \times M \times 2^E.$$

Мантисса – это целое число фиксированной длины, представляющее старшие разряды действительного числа. Допустим, мантисса состоит из трех бит ($|M| = 3$). Возьмем, например, число «5», которое в двоичной системе будет равно 101_2. Старший бит соответствует $2^2 = 4$, средний (который в нашем примере равен нулю) $2^1 = 2$, а младший $2^0 = 1$. Порядок – это степень базы (двойки) старшего разряда. В нашем случае $E = 2$. Такие числа удобно записывать в так называемой стандартной форме, например, «$1.01e + 2$». Из такой записи видно, что мантисса состоит из трех знаков, а порядок равен двум.

Допустим необходимо получить дробное число, используя те же 3 бита мантиссы. Это можно сделать, например, для $E = 1$. Тогда искомое число будет равно

$$1.01e + 1 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 2 + 0{,}5 = 2{,}5.$$

Очевидно, что таким образом одно и то же число можно представить поразному. Рассмотрим пример с длиной мантиссы $|M| = 4$. Число «2» можно представить в следующем виде:

$$2 = 10 \text{ (в двоичной системе)} = 1.000e + 1 = 0.100e + 2 = 0.010e + 3.$$

Существование нескольких представлений одного и того же числа существенно усложняют сравнение этих чисел в процессоре. Поэтому современные числа представлены в так называемом нормализованном виде (рис. 2), когда первый бит мантиссы всегда подразумевался равным единице, экономя один разряд. Эти и другие идеи воплотились в стандарт IEEE 754, последняя редакция которого – IEEE 754-2008 [6].

РИС. 2. Нормализованное представление числа с плавающей запятой

Числа с плавающей запятой согласно IEEE-754 представлены в виде знака (s), мантиссы (M) и порядка (E) следующим образом:

$$(-1)^s \times 1.M \times 2^E.$$

Чтобы не загромождать читателя чрезмерной информацией, рассмотрим только один тип данных, с одинарной точностью (single). Числа с половинной (half), двойной (double) и расширенной (extended) точностью обладают теми же особенностями, но отличаются числом разрядов порядка и мантиссы. В числах одинарной точности (float/single) порядок состоит из 8 бит, а мантисса – из 23. Эффективный порядок определяется как E-127.

Специальные числа. В IEEE 754 число «0» представляется значением с порядком, равным $E = E_{min} - 1$ (для single это – 127) и нулевой мантиссой. Введение нуля как самостоятельного числа (так как в нормализованном представлении нельзя представить ноль) позволило избежать многих странностей в арифметике. И хоть операции с нулем нужно обрабатывать отдельно, обычно они выполняются быстрее, чем с обычными числами.

Также в IEEE 754 предусмотрено представление для специальных чисел, работа с которыми вызывает исключение. К таким числам относится бесконечность (± ∞) и неопределенность (*NaN*). Бесконечности представлены как числа с порядком $E = E_{max} + 1$ и нулевой мантиссой. Получить бесконечность можно при переполнении и при делении ненулевого числа на ноль. Бесконечность при делении разработчики определили, исходя из существования пределов, когда делимое и делитель стремиться к какому-то числу. Соответственно, $c/0 = \pm \infty$ (например, 3/0 = + ∞, а – 3/0 = – ∞), так как если делимое стремиться к константе, а делитель к нулю, предел равен бесконечности. При 0/0 предел не существует, поэтому результатом будет неопределенность.

Неопределенность или *NaN* (от «not a number») – это представление, введенное для того, чтобы арифметическая операция могла всегда вернуть какое-то не бессмысленное значение. В IEEE 754 *NaN* представлен как число, в котором $E = E_{max} + 1$, а мантисса не нулевая. Любая операция с *NaN* возвращает *NaN*. При желании в мантиссу можно записывать информацию, которую программа сможет интерпретировать. Стандартом это не оговорено, и мантисса чаще всего игнорируется. По определению *NaN* ≠ *NaN*, поэтому, для проверки значения переменной нужно просто сравнить его с самим собой.

Значение *NaN* возникает при выполнении любой из следующих операций: “∞ + (– ∞)”, “0 × ∞”, “0/0, ∞/∞”, “$\sqrt{x}$ ” (где $x < 0$).

Число «0». Поскольку в IEEE754 число «0» представлено фиксированным значением порядка и мантиссы, знак может быть произвольным. В результате, число «0» имеет два представления, отличающиеся знаком. Так, 3•(+ 0) = + 0, а 3•(– 0)= – 0. В стандарте знак сохранили умышленно, чтобы выражения, которые в результате переполнения или потери значимости превращаются в бесконечность или в ноль, при умножении и делении все же могли представить наиболее приближенный к корректному результат. Например, если бы у нуля не было знака, выражение $1/(1/x) = x$ не выполнялось бы верно при $x = \pm \infty$, так как 1/∞ и 1/ – ∞ равны 0. Однако по определению принято, что « + 0 = – 0». Это позволяет избежать дополнительных затруднений и странностей.

Еще один пример: «(+ ∞/0) + ∞ = + ∞», тогда как «(+ ∞/ – 0) + ∞ = *NaN*». Бесконечность в данном случае лучше, чем *NaN*, поскольку если в арифметическом выражении появился *NaN*, результатом всего выражения всегда будет *NaN*. Если же в выражении встретилась бесконечность, то результатом может быть ноль, бесконечность или обычное число с плавающей запятой. Например, «1/∞ = 0». Это позволяет вычислить следующую дробь для $x = 1, 2, 3, 4$:

$$F(x) = 7 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}}.$$

Денормализованные числа. Пусть имеем нормализованное представление с длиной мантиссы $|M| = 2$ бита плюс один бит нормализации (рис. 3) и диапазоном значений порядка $-1 \le E \le 2$. В этом случае получим 16 чисел.

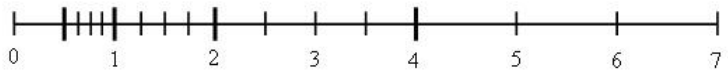


РИС. 3. Нормализованное представление числа с плавающей запятой

Крупными штрихами показаны числа с мантиссой, равной 1,00. Видно, что расстояние от нуля до ближайшего числа (0 – 0,5) больше, чем от этого числа к следующему (0,5 – 0,625). Это значит, что разница двух любых чисел от 0,5 до 1 даст 0, даже если эти числа не равны. Что еще хуже, в пропасть между 0,5 и 0 попадает разница чисел, больших 1. Например, «1,5 – 1,25 = 0» (см. рис. 3).

Для решения этой проблемы разработчиками стандарта IEEE 754 сделано следующее. Мы знаем, что при $E = E_{min} - 1$ (для float это «– 127») и нулевой мантиссе число считается равным нулю. Если же мантисса не нулевая, то число считается не нулевым, его порядок полагается $E = E_{min}$, причем неявный старший бит мантиссы полагается равным нулю. Такие числа называются денормализованными.

Строго говоря, числа с плавающей запятой теперь имеют вид:

$$(-1)^s \times 1.M \times 2^E, \text{ если } E\min \le E \le E\max \text{ (нормализованные числа);}$$

$$(-1)^s \times 0.M \times 2^{Emin}, \text{ если } E = E_{min} - 1 \text{ (денормализованные числа).}$$

Вернемся к примеру. Наш $E_{min} = -1$. Введем новое значение порядка, $E = -2$, при котором числа являются денормализованными. В результате получаем новое представление чисел (рис. 4).

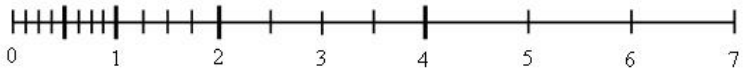


РИС. 4. Денормализованные числа

Интервал от 0 до 0,5 заполняют денормализованные числа, что дает возможность не проваливаться в ноль в вышерассмотренных примерах (0,5 – 0,25 и 1,5 – 1,25). Это сделало представление более устойчивым к ошибкам округления для чисел, близких к нулю.

Влияние денормализованных чисел. Денормализованные числа требуют особой обработки и, как показали эксперименты [7], это сказывается на производительности программ. Причины и масштабы падения производительности зависят от конкретной реализации арифметики на данном оборудовании. Наиболее пагубные последствия происходят в случае возникновения исключений и программной нормализации.

Поскольку в работе [7] эксперименты проведены на весьма устаревших процессорах, авторы решили повторить его на современном оборудовании и проиллюстрировать в данном обзоре. Суть эксперимента в следующем: пусть дана матрица, на краях которой заданы некоторые значения, а все остальные элементы равны нулю. Эксперимент состоит из множества итераций, на каждой из которых строиться новая матрица, каждый элемент которой представляет среднее значение от соседних элементов исходной матрицы. Такой класс алгоритмов является характерным для конечно-разностных методов.

На рис. 5 схематически показано, как меняются элементы от итерации к итерации. Вначале все элементы матрицы, кроме крайних, равны нулю (рис 5, а). Далее, через несколько десятков итераций соседние элементы матрицы становятся настолько малыми, что их невозможно представить в нормализованном виде. Штрихом на рис. 5, б показаны обычные значения, а жирной линией – денормализованные числа. Некоторое время количество денормализованных чисел растет (рис. 5, в), после чего, с уменьшением площади внутренней границы, со временем исчезают и денормализованные числа. После определенной итерации в матрице таких чисел не остается совсем (рис. 5, г).

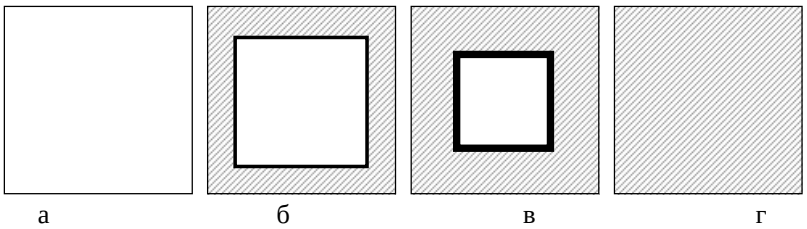
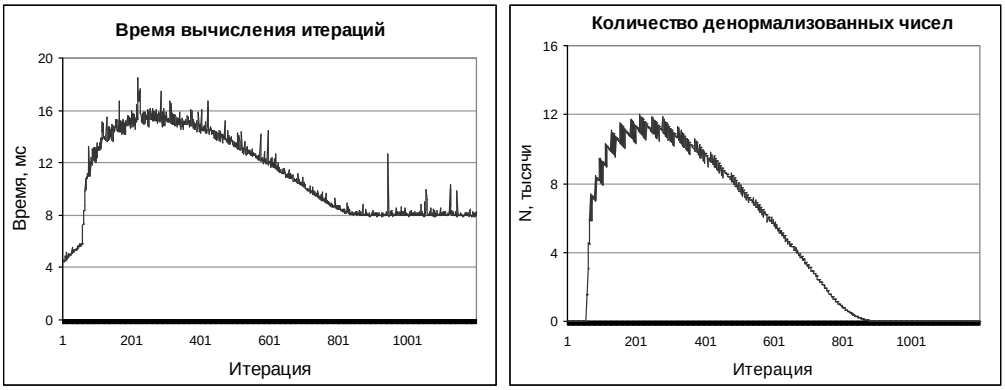


РИС. 5. Заполнение матрицы денормализованными числами: а – первая итерация – денормализованных чисел еще нет; б – денормализванные числа начинают появляться на границе; в – максимальное количество денормализованных чисел в матрице; г – денормализованных чисел в матрице снова нет

Эксперимент проведен на компьютере Intel Core 2 Duo 2.14 ГГц, использована матрица размером 500 x 500, и значение на краях $V = 100$, тип данных одинарной точности. Отметим, что в эксперименте с двойной точностью наблюдается аналогичная картина.

Из рис. 6, а, б видно, что существует явная корреляция времени вычисления итерации и количества денормализованных чисел в матрице в этот момент. Это показывает наличие проблем с производительностью при работе с денормализованными числами даже на современных процессорах. На пике графика ~ 6 % чисел являются денормализованными, при этом производительность падает в два раза. Поэтому можно оценить, что скорость работы с денормалиованными числами будет в 30 раз медленнее, чем с обычными.

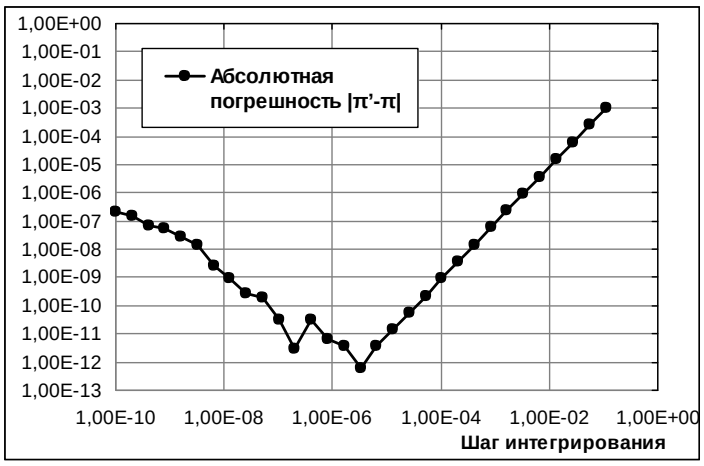
Одним из побочных эффектов эксперимента можно считать то, что на начальных итерациях время вычислений в два раза ниже, чем на 1000-й. Таким образом, работа со специальными числами (по крайней мере с нулем) в два раза быстрее, чем с обычными числами с плавающей запятой.



а б

РИС. 6. Графики: а – время выполнения итераций; б – количество денормализованных чисел на итерации

Другим примером является увеличение абсолютной погрешности при уменьшении шага численного интегрирования, проиллюстрированной на примере вычисления числа π по формуле (1). На рис. 7 показана величина абсолютной погрешности в зависимости от шага численного интегрирования.



$$\int_0^1 \frac{4}{1=x^2} = \pi(1)$$

РИС. 7. Погрешность численного интегрирования

Порядок чисел. Одна из особенностей представления чисел в формате IEEE754 состоит в том, что порядок и мантисса расположены друг за другом таким образом, что они вместе образуют последовательность целых чисел $\{n\}$ для которых выполняется:

$$n < n + 1 \Rightarrow F(n) < F(n + 1),$$

где $F(n)$ – число с плавающей запятой, образованное от целого n, разбиением его битов на порядок и мантиссу.

Поэтому, если взять положительное число с плавающей запятой, преобразовать его к целому и прибавить «1», мы получим следующее число, которое представимо в этой арифметике. На Си это можно сделать так:

```
float a=0.5;
int n = *((int*) &a);
float b = *((float*) &(++n));
printf("После %e следующее число: %e, разница (%e)\n", a, b, b-a);
```

Округление. Анализ погрешностей округления охватывает слишком широкую область знаний, даже при попытке выделить наиболее важные аспекты, связанные со спецификой чисел с плавающей запятой. Для подробного ознакомления смотрите книгу Хигхема [8].

Ошибочные результаты при численных расчетах в арифметике с плавающей запятой возникают по трем причинам: погрешности исходных данных, погрешности представления, погрешности, вносимые арифметическими операциями. Последние два вида погрешностей характерны для чисел с плавающей запятой. Погрешность представления возникает из-за того, что действительные числа практически всегда невозможно представить в виде числа с конечным числом разрядов. Но такая погрешность очень мала, и ее можно регулировать, выбирая тип данных. Погрешность, вносимая арифметическими операциями, может быть очень существенной, из-за аддитивного и мультипликативного эффектов, возникающих, когда результаты одних операций используются в качестве аргументов для других. При увеличении числа итераций погрешность растет и очень быстро выходит за допустимые пределы.

Наиболее опасная с точки зрения погрешностей операция – вычитание разнозначных чисел. Такая операция уничтожает значимые старшие разряды и усиливает младшие разряды, которые представлены с погрешностью. Такая ситуация называется аннулированием. Используя алгебраические преобразования можно существенно снизить погрешность. Для многих широко распространенных математических формул разработаны специальные формы, которые позволяют значительно уменьшить погрешность. Например, формулу «$x^2 - y^2$» лучше вычислять следующим образом: «$(x - y)(x+ y)$». В этом случае аннулирование не окажет катастрофического воздействия на результат. В работах [5, 8] приведено множество формул, рекомендуемых при расчете в арифметике с плавающей запятой, а также теоремы, доказывающие границы погрешности.

Неассоциативность арифметических операций. Неассоциативность арифметических операций над числами с плавающей запятой очевидна на следующем примере:

$$(1020 + 1) - 1020 = 0 \neq (1020 - 1020) + 1 = 1.$$

Допустим, имеем программу суммирования чисел.

```
double s = 0.0;
for (int i=0; i<n; i++) s = s + t[i];
```

Некоторые компиляторы по умолчанию могут переписать код для использования нескольких арифметико-логических устройств (АЛУ) одновременно (будем считать, что n делится на 2):

```
double sa[2], s;
sa[0]=sa[1]=0.0;
for (int i=0; i<n/2; i++) {
    sa[0]=sa[0]+t[i*2+0];
    sa[1]=sa[1]+t[i*2+1];
}
s=sa[0]+sa[1];
```

Поскольку операции суммирования не ассоциативны, эти две программы могут выдать различный результат.

Выбор минимального значения. Допустим, из двух значений нам нужно выбрать минимальное. В Си это можно сделать используя тернарный оператор «?» одним из способов, приведенных в таблице.

ТАБЛИЦА. Вычисление минимума и максимума для специальных чисел

x	y	$x < y\ ?\ x : y$	$x <= y\ ?\ x : y$	$x > y\ ?\ y : x$	$x > = y\ ?\ y : x$
+ 0	– 0	– 0	+ 0	+ 0	– 0
NaN	1	1	1	*NaN*	*NaN*

Часто компилятор считает их эквивалентными и всегда использует первый вариант, так как он выполняется за одну инструкцию процессора. Но если мы учтем ± 0 и *NaN*, эти операции никак не эквивалентны.

Сравнение чисел. Очень распространенная ошибка при работе с float-ами возникает при проверке на равенство. Например,

```
float fValue = 0.2;
if (fValue == 0.2) DoStuff();
```

Ошибка здесь, во-первых, в том, что 0,2 не имеет точного двоичного представления, а во-вторых 0,2 – это константа двойной точности, а переменная fValue – одинарной, и никакой гарантии о поведении этого сравнения нет.

Немного лучший, но все равно ошибочный способ, это сравнивать разницу с допустимой абсолютной погрешностью:

```
if (fabs(fValue – fExpected) < 0.0001) DoStuff();
```

Недостаток такого подхода в том, что погрешность представления числа увеличивается с ростом самого этого числа. Так, если программа ожидает «10000», то приведенное равенство не будет выполняться для ближайшего соседнего числа (10000,000977). Это особенно актуально, если в программе имеется преобразование из одинарной точности в двойную и/или наоборот.

Проблема выбора правильной процедуры сравнения хорошо описана в работе Брюса Доусона [9]. В ней предлагается сравнивать числа с плавающей запятой преобразованием к целочисленной переменной. Это – лучший способ, хотя и работает не на любом оборудовании:

```
bool AlmostEqual (float A, float B, int maxUlps)
{
    // maxUlps не должен быть отрицательным и не
    // слишком большим, чтобы
    // NaN не был равен ни одному числу
    assert(maxUlps > 0 && maxUlps < 4 * 1024 * 1024);
    int aInt = *(int*)&A;
    // Уберем знак в aInt, если он есть,
    // чтобы получить правильно
    // упорядоченную последовательность
    if (aInt < 0) aInt = 0x80000000 - aInt;
    // Аналогично для bInt
    int bInt = *(int*)&B;
    if (bInt < 0) bInt = 0x80000000 - bInt;
    unsigned int intDiff = abs(aInt - bInt);
    if (intDiff <= maxUlps)
        return true;
    return false;
}
```

В этой программе max*Ulps* (от «*Units-In-Last-Place*») – это максимальное количество чисел с плавающей запятой, которое может лежать между проверяемым и ожидаемым значением. Другой смысл этой переменной – это количество двоичных разрядов (начиная с младшего), в сравниваемых числах разрешается упустить. Например, max*Ulps* = 16, означает, что младшие 4 бита ($\log_2 16$) могут не совпадать, а числа все равно будут считаться равными. При этом, при сравнении с числом 10000 абсолютная погрешность будет равна 0,0146, а при сравнении с 0.001, погрешность будет менее 0.00000001 (10 – 8).

Параллельные вычисления. В параллельных вычислениях использование чисел с плавающей запятой иногда приводит к тому, что результат вычислений зависит от количества процессов, использованных в конкретном случае. Это связано с различием в алгебраических свойствах арифметики действительных чисел и арифметики чисел с плавающей запятой, которые существенно сильнее проявляются в параллельном вычислении, поскольку в таких вычислениях для разработки и оптимизации алгоритма широко применяются алгебраические свойства операций (например, ассоциативность). Хорошим примером является эксперимент, сделанный Т. Мэттсоном [10], в котором суммирование случайных чисел в массиве из 20 тысяч элементов приводит, в зависимости от использованного числа процессов, к значительной разнице в результате. Суть проблемы параллельной агрегации большого числа значений в том, что при разработке параллельного алгоритма применяются алгебраические преобразования, исходящие из ассоциативности арифметических операций. Так, формулу $S = \sum_{i=1}^{n} x_i$ можно переписать как $S = S_1 + S_2 = \sum_{i=1}^{n/2} x_i + \sum_{i=n/2+1}^{n} x_i$. При этом частные суммы

можно рассчитать независимо на отдельных процессорах. Повторяя разбиение можно получить эффективный каскадный параллельный алгоритм, который с точки зрения арифметики с плавающей запятой содержит неоднозначности, так как конечный результат зависит от порядка суммирования (см. п. 3.7). Это приводит к недетерминированности результата.

Хигхем [8] рекомендует использовать для агрегации переменную более высокой точности, чем исходные значения – это наиболее простой способ сократить погрешность округления, который, к тому же, автоматически работает в параллельных вычислениях. Если же такой подход по какой-то причине не приемлем (например, значения и так представлены в максимально возможной точности), можно воспользоваться одним из рекомендованных алгоритмов «суммирования с компенсацией» [11], например,

```
// Алгоритм Кэхэна суммирования с компенсацией
double CompensatedSum(double *x, int count)
{
    s=0; e=0; temp=0; y=0;
    for (i=0; i<count; i++) {
        temp = s;
        y = x[i] + e;
        s = temp + y;
        e = (temp – s) + y;
    }
}
```

Несмотря на приведенные трудности, понимание свойств арифметики с плавающей запятой позволяет использовать их для обеспечения точных результатов даже на пониженной разрядности. В своей параллельной библиотеке PLASMA Д. Донгарра и его команда используют смешанную арифметику одинарной и двойной точности для оптимизации скорости программы без ущерба точности конечного результата [12].

Но не для всех задач фиксированная разрядность чисел с плавающей запятой подходит для достижения корректного результата. В этом случае применяют эмулируемую арифметику с произвольной разрядностью. Скорость решения в этом случае падает в десятки раз, зато точность решения гарантирована (если исходные данные точны). В работах Николаевской и Чистяковой проведены исследования таких задач и проиллюстрировано практическое применение эмулируемой арифметики [13].

Выводы. Арифметика с плавающей IEEE 754, используемая практически во всех современных компьютерах, спроектирована таким образом, который позволяет избавиться от многих странностей, возникающих при попытке представления действительных чисел конечным числом разрядов. Не смотря на это, так называемый «наивный подход», при котором не учитывается аксиоматика машинной арифметики, может привести к неожиданным результатам.

Особенно следует отметить проявление недетерминированности арифметики с плавающей запятой в параллельных вычислениях при использовании каскадных алгоритмов агрегации, поскольку такие алгоритмы исходят из предположения ассоциативности, которые в этом случае не выполняются. Проблема точности решений в параллельных компьютерах усиливается тем, что на них как правило решаются задачи большого объема, а погрешность представления чисел увеличивается при росте числа итераций.

Единого решения, которое смогло бы удовлетворить одновременно требованиям точности и скорости не сегодняшний день не существует, и в каждом конкретном случае приходится искать компромиссы.

Р.А. Ющенко, О.К. Ющенко

ОСОБЛИВОСТІ АРИФМЕТИКИ З ПЛАВАЮЧОЮ КОМОЮ
НА СУЧАСНИХ КОМП'ЮТЕРАХ

Представлено короткий огляд чисел із плаваючою комою і відповідної арифметики. Підкреслено особливості, які відрізняють її від арифметики дійсних чисел і часто є причиною істотних проблем під час числового розв'язання задач на комп'ютері. Ці проблеми підсилюються на даних великого обсягу і в паралельних обчисленнях. Незважаючи на все це, мінімізувати й навіть гарантувати точність ровз'язку можливо навіть без суттєвих втрат обчислювальної продуктивності.

R. Iushchenko, O. Iushchenko

SPECIFICITIES OF FLOATING POINT ARITHMETIC ON MODERN COMPUTERS

An overview of modern floating point arithmetic is presented. The aspects, which are different from real numbers arithmetic, and, thus, cause a lot of numeric problems are outlined. These problems become substantial on large amounts of data and in parallel computations. Nevertheless, it is possible to guarantee the required accuracy of the results without considerable loss of performance.

1. *Молчанов И.Н.* Машинная математика. Проблемы и перспективы // Кибернетика и системный анализ. – 2004. – № 6. – С. 65 – 72.
2. *Молчанов И.Н., Перевозчикова О.Л., Химич А.Н.* Опыт разработки семейства кластерных комплексов Инпарком // Там же. – 2009. – № 6. – С. 88 – 96.
3. *http*://www.cs.unc.edu/~ibr/projects/paranoia/.
4. *http*://software.intel.com/en-us/videos/tim-mattson-floating-points-arent-real/.
5. *Goldberg D.* What Every Computer Scientist Should Know About Floating-Point Arithmetic // ACM Computing Surveys. – 1991. – Vol. 1, N 23. – P. 5 – 48.
6. *IEEE* 754-2008 Standard for Floating-Point Arithmetic.
7. *Bjørndalen J.M., Anshus O.J.* Trusting floating point benchmarks - are your benchmarks really data independent?, Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing, June 18 – 21, 2006, Umeå, Sweden.
8. *Higham N.* Accuracy and Stability of Numerical Algorithms. – Philadelphia: 2002, SIAM. – P. 680.

9. *Bruce Dawson*,
http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm
10. *Tim Mattson*,
http://software.intel.com/en-us/videos/tim-mattson-floating-points-arent-real/.
11. *Kahan W.* Implementation of algorithms. Technical Report 20, Department of Computer Science, University of California, Berkeley, CA, USA, 1973.
12. *Buttari A., Dongarra J., Kurzak J., Luszczek P., Tomov S*. "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy," ACM Transactions on Mathematical Software. – 2008. – Vol. 34, N 4. – P. 17 – 22.
13. *Николаевская Е.А., Чистякова Т.В.* Программно-алгоритмические методы повышения точности компьютерных решений // Кибернетика и системный анализ. – 2009. – № 6. – С. 172 – 176.

Получено 15.03.2016

Об авторах:

Ющенко Руслан Андреевич,
кандидат физико-математических наук, старший научный сотрудник
Института кибернетики имени В.М. Глушкова НАН Украины,

Ющенко Александра Константиновна,
младший научный сотрудник
Института кибернетики имени В.М. Глушкова НАН Украины.