УДК 004

Р.А. Ющенко

АКТУАЛЬНЫЕ ПРОБЛЕМЫ СОВРЕМЕННОЙ АРИФМЕТИКИ С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

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 для больших объемов данных и параллельных вычислений. 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 физических и экономических процессов. 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 амплитуды (абсолютной величины) 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 их них вообще не имело физического смысла. Поэтому в 70-е годы была проведена огромная работа по стандартизации чисел с плавающей запятой, в результате которой разработан и до настоящего времени существует практически без изменений стандарт IEEE 754. В результате с 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].

Арифметика с плавающей запятой считается экзотической областью компьютерных наук, не смотря на то, что соответствующие типы данных присутствуют в каждом языке программирования. И благодаря стандартам, в большинстве тривиальных случаев особенностью такой арифметики можно пренебречь. Но сейчас, спустя сорок лет, проблема представления чисел вновь стала актуальной сразу по нескольким причинам. Во-первых, экспоненциальное развитие суперкомпьютеров (закон Мура) привело к тому, что объемы 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 форсирования скорости. Это характерно, например, для графических процессоров (гонка «мачо-флопсов») [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.

© Р.А. Ющенко, 2012

1. 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, который является компромиссом между точностью и амплитудой принимаемых значений.

Число с плавающей запятой состоит из набора отдельных разрядов, условно разделенных на знак, порядок и мантиссу. Порядок и мантисса – наборы разрядов, которые вместе со знаком дают представление числа с плавающей запятой как показано на рис. 1.

Рис. 1. Преставление 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, которое представляет 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

$1.01e{+}1 = 1\times2^1 + 0\times2^0 + 1\times2^{-1} = 2 + 0{,}5 = 2{,}5$.

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

2 = 10 (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 уже в самых первых машинах числа представляли в так называемом *нормализованном виде*, когда первый разряд мантиссы всегда подразумевался равным единице, как показано на рис. 2.

Рис. 2. Нормализованное представление числа с плавающей запятой

Строго говоря, нормализованное число имеет следующий вид:

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

Нормализация экономит один разряд (так как неявную единицу не нужно хранить в памяти) и обеспечивает уникальность представления числа. В нашем примере «2» имеет единственное представление («1.000e+1»), а мантисса хранится в памяти как «000», так как старшая единица подразумевается неявно. Но в нормализованном представлении чисел возникает новая проблема – в такой форме невозможно представить ноль. Решений, обладающих своими преимуществами и недостатками, потенциально существует множество. Выбор, как и для многих парадигма-

тических концепций в компьютерных науках, продиктован исторически сложившейся ситуацией.

2. Немного истории

В 60-е и 70-е годы не было ни единого стандарта представления чисел с плавающей запятой, ни способов округления и арифметических операций. В результате программы были крайне не мобильны. Но еще большей проблемой было то, что у разных компьютеров были свои «странности» и их нужно было знать и учитывать в программе. Например, разница двух не равных чисел могла возвращать ноль. В результате выражения «$X = Y$» и «$X - Y = 0$» вступали в противоречие. Умельцы обходили эту проблему хитрыми трюками, например, делали присваивание «$X = (X - X) + X$» перед операциями умножения и деления, чтобы избежать проблем.

Инициатива создать единый стандарт для представления чисел с плавающей запятой совпала с попытками в 1976 году компанией Intel разработать «лучшую» арифметику для новых сопроцессоров к 8086 и i432. За разработку взялись ученые киты в этой области, проф. Джон Палмер и Уильям Кэхэн. Последний в своем интервью высказал мнение, что серьезность, с которой Intel разрабатывала свою арифметику, заставила другие компании объединиться и начать процесс стандартизации [6].

Все были настроены серьезно, так как стандартизация существующей архитектуры давала огромное конкурентное преимущество компании, в которой такая архитектура уже реализована. Свои предложения представили компании DEC, National Superconductor, Zilog, Motorola. Производители мейнфреймов Cray и IBM наблюдали со стороны. Компания Intel, разумеется, тоже представила свою новую арифметику. Авторами предложенной спецификации стали Уильям Кэхэн, Джероми Кунен и Гарольд Стоун и их предложение сразу прозвали «K-C-S».

Практически сразу же были отброшены все предложения, кроме двух: VAX от DEC и «K-C-S» от Intel. Спецификация VAX была значительно проще, уже была реализована в компьютерах PDP-11, и было понятно, как на ней получить максимальную производительность. С другой стороны в «K-C-S» содержалось много полезной функциональности, такой как «специальные» и «денормализованные» числа (рассмотрены далее).

В «K-C-S» все арифметические алгоритмы заданы строго и требуется, чтобы в реализации результат с ними совпадал. Это позволяет выводить строгие выкладки в рамках этой спецификации. Если раньше математик решал задачу численными методами и доказывал свойства решения, не было никакой гарантии, что эти свойства сохранятся в программе. Строгость арифметики «K-C-S» сделала возможным доказательство теорем, опираясь на арифметику с плавающей запятой.

Компания DEC сделала все, чтобы ее спецификацию сделали стандартом. Она даже заручилась поддержкой некоторых авторитетных ученых в том, что арифметика «K-C-S» в принципе не может достигнуть такой же производительности, как у DEC. Ирония в том, что Intel знала, как сделать свою спецификацию такой же производительной, но эти хитрости были коммерческой тайной. Если бы Intel не уступила и не открыла часть секретов, она бы не смогла сдержать натиск DEC.

3. Представление чисел с плавающей запятой сегодня

Разработчики «K-C-S» победили и теперь их детище воплотилось в стандарт IEEE 754, последняя редакция которого – IEEE 754-2008 [7]. Числа с плавающей запятой в нем представлены в виде знака (s), мантиссы (M) и порядка (E) следующим образом:

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

Вообще говоря, в стандарте IEE 754-2008 кроме чисел с основанием 2 присутствуют числа с основанием 10, так называемые десятичные (decimal) числа с плавающей запятой. В данной статье для простоты такое представление не рассматривается.

Чтобы не загромождать читателя чрезмерной информацией, рассмотрим только один тип данных, с одинарной точ-

ностью (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 порядок состоит из 8 бит, а мантисса – из 23. Эффективный порядок определяется как «$E - 127$». Например, число 0,15625 будет записано в памяти как

Рис. 3. Пример числа с плавающей запятой в IEEE754 (из Википедии)

В этом примере: знак $s = 0$ (положительное число), порядок

$E = 01111100_2 - 127_{10} = -3$,

мантисса $M = 1.01_2$ (первая единица неявная). В результате число $F = 1.01_2$e-3 = $= 2^{-3} + 2^{-5} = 0{,}125 + 0{,}03125 = 0{,}15625$

Специальные числа. 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*). Эти числа позволяет вернуть адекватное значение при возникновении исключительных ситуаций. 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, $3/0 = +\infty$, а $-3/0 = -\infty$), 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.

Значение *NaN* возникает при выполнении любой из следующих операций:

- $\infty + (-\infty)$,
- $0 \times \infty$,
- $0/0$, ∞/∞,
- sqrt(x), где $x < 0$.

По определению $NaN \neq NaN$, поэтому, для проверки значения переменной нужно просто сравнить его с самим собой.

Ноль со знаком. 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 максимально корректный результат. Например, если бы у нуля не было знака, выражение «1 / (1 /x) = x» не выполнялось бы верно при «$x = \pm\infty$», так как «$1 / \infty$» и «$1 / -\infty$» равны «0». Однако по определению принято, что «+0 = –0». Это позволяет избежать дополнительных затруднений и странностей. Еще один пример:

$$(+\infty / 0) + \infty = +\infty, \text{ тогда как}$$
$$(+\infty / -0) + \infty = NaN.$$

Чем бесконечность в данном случае лучше, чем «*NaN*»? Тем, что 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,

$$1 / \infty = 0.$$

Денормализованные числа. Пусть имеем нормализованное представление с длиной мантиссы «$|M| = 2$» бита (плюс один бит нормализации) и диапазоном значений порядка «$-1 \le E \le 2$». В этом случае получим 16 чисел (рис. 4).

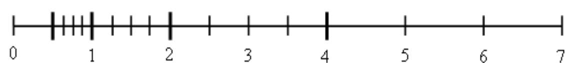


Рис. 4. Нормализованное представление числа с плавающей запятой

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,

$$1{,}5 - 1{,}25 = 0 \text{ (см. рис. 4).}$$

В «околонулевую яму» подпадает не каждая программа. Согласно статистике 70-х годов в среднем каждый компьютер сталкивался с такой проблемой один раз в месяц. Учитывая, что компьютеры приобретали массовость, разработчики «K-C-S» посчитали эту проблему достаточно серьезной, чтобы решать ее на аппаратном уровне. Предложенное ими решение состояло в следующем. Мы знаем, что при «$E = E_{min} - 1$» (для *single* 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:

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

(нормализованные числа)

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

(денормализованные числа)

Вернемся к примеру, в котором «$E_{min} = -1$». Введем новое значение порядка, «$E = -2$», при котором числа являются денормализованными. В результате получаем новое представление чисел (рис. 5).

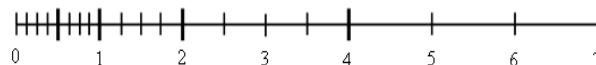


Рис. 5. Денормализованные числа

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

Но роскошь использования денормализованного представления чисел в процессоре не дается бесплатно. Из-за того, что такие числа нужно обрабатывать по-другому во всех арифметических операциях, трудно сделать работу такой арифметики эффективной. Это накладывает дополнительные сложности при реализации АЛУ в процессоре. Кроме того, несмотря на свою полезность, денормализованные числа не являются панацеей и за округлением до нуля все равно нужно следить. Поэтому эта функциональность стала камнем преткновения при разработке стандарта и встретила самое сильное сопротивление разработчиков оборудования.

Влияние денормализованных чисел на производительность. Денормализованные числа требуют особой обработки и, как показали эксперименты [8], это сказывается на производительности программ. Причины и масштабы падения производительности зависят от конкретной реализации арифметики на данном оборудовании. Наиболее пагубные последствия возникают в случае возникновения исключений и программной нормализации.

Поскольку в работе [8] эксперименты проведены на весьма устаревших процессорах, автор решил 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 итера-

ций, 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 численного решения дифференциальных уравнений в частных производных.

На рис. 6 схематически показано, как меняются элементы от итерации к итерации. Вначале все элементы матрицы, кроме крайних, равны нулю (рис. 6, а). Далее, через несколько десятков итераций соседние элементы матрицы становятся насколько малыми, что их невозможно представить в нормализованном виде. Штрихом на рис. 6, б показаны обычные значения, а жирной линией – денормализованные числа. Некоторое время количество денормализованных чисел растет (рис. 6, в), после чего уменьшение площади внутренней границы уменьшается и с ними, со временем, исчезают и денормализованные числа. После определенной итерации в матрице таких чисел не остается совсем (рис. 6, г).

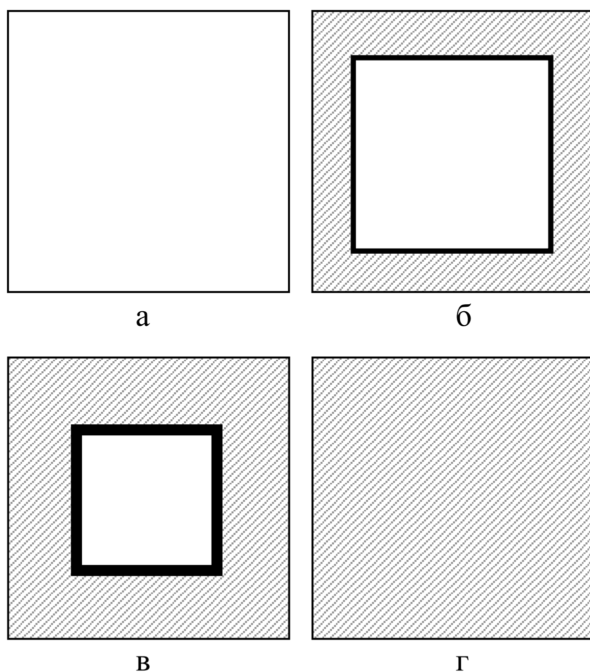
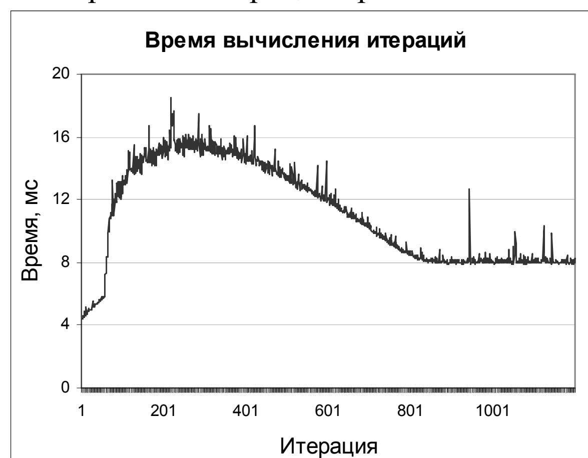
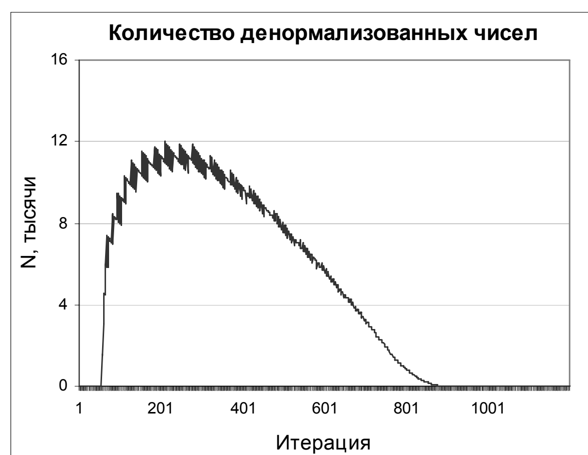


Рис. 6. Заполнение матрицы денормализованными числами

Из рис. 7 видно, что существует явная корреляция времени вычисления итерации и количества денормализованных чисел в матрице в этот момент, что показывает наличие проблем с производительностью у денормализованных чисел даже на современных процессорах.



а



б

Рис. 7. Графики: а – времени вычисления итераций; б – количество денормализованных чисел на итерации

Эксперимент проведен на компьютере Intel Core 2 Duo 2.14 ГГц, использована матрица размером 500x500, и значение на краях $V = 100$, тип данных одинарной точности. Отметим, что в эксперименте с двойной точностью наблюдается аналогичная картина. На пике графика ~6% чисел являются денормализованными, при этом производительность падает в два раза, поэтому вычисления следует ожидать потерю производительности в 30 раз, если все числа являются денормализованными.

Одним из побочных эффектов эксперимента можно считать то, что на начальных итерациях время вычислений в два раза ниже, чем на 1000-ной. Таким образом, работа со специальными числами (по

крайней мере с нулем), в два раза быстрее, чем с обычными числами с плавающей запятой.

Очередность чисел в IEEE754. 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:

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

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, прибавить «1», мы получим следующее число, которое представимо в этой арифметике. На Си это можно сделать так (Этот код будет выполняться корректно только на архитектуре с 32-битным типом «*int*»):

```
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 [9].

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, если 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. 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 вычитание не окажет катастрофического воздействия на результат. В работах [5, 9] приведено множество формул, рекомендуемых при расчете в арифметике с плавающей запятой, а также теоремы, доказывающие границы погрешности.

Минимизировать погрешность арифметических операций помогает стандарт. В правилах округления в IEEE754 сказано, что результат любой арифметической операции должен быть таким, как если бы он был выполнен над точными значениями и округлен до ближайшего числа, представимого в этом формате. Уменьшить ошибки округления помогают также денормализованные числа. Но такой подход требует от АЛУ наличия двух дополнительных скрытых разряда в регистрах, а также отдельную ветку обработки денормализованных чисел. Некоторые опции компилятора (такие как «–ffast-math» в gcc) могут отключить полную поддержку IEEE754, из-за чего погрешность округления может существенно возрасти.

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

$$(10^{20} + 1) - 10^{20} = 0 \neq$$
$$\neq (10^{20} - 10^{20}) + 1 = 1.$$

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 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 минимума из двух значений. Допустим из двух значений нам нужно выбрать минимальное. В Си это можно сделать одним из следующих способов:

- `x < y? x: y;`
- `x <= y? x: y;`
- `x > y? y: x;`
- `x >= y? y: x.`

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 (таблица).

Сравнение чисел. Очень распространенная ошибка при работе с числами с плавающей запятой 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(); //
     fValue=fExpected?
```

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 [10]. В ней предлагается сравнивать числа с плавающей запятой преобразованием к целочисленной переменной. Это – лучший способ, хотя и работает не на любом оборудовании. В этой программе maxUlps (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, «maxUlps=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 бу дет равна «0,0146», а при сравнении с «0.001», погрешность будет менее 0.00000001 (10 – 8). Алгоритм проиллюстрирован на рис. 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. Хорошим примером является эксперимент, сделанный Т. Мэттсоном [11], в котором сум-

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*

```
bool AlmostEqual2sComplement(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 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;
}
```

Рис. 8. Алгоритм Б. Доусона сравнения чисел с плавающей запятой

мирование случайных чисел в массиве из 20 тысяч элементов приводит, в зависимости от 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 [9] 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» [12], например, показанным на рис. 9.

```
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;
    }
}
```

Рис. 9. Алгоритм Кэхэна суммирования с компенсацией

Несмотря на приведенные трудности, понимание свойств арифметики с плавающей запятой позволяет использовать их для обеспечения точных результатов даже на пониженной разрядности. Так, в своей параллельной библиотеке PLASMA Д. Донгарра и др. [13] комбинируют арифметику одинарной и двойной точности для оптимизации скорости программы без ущерба точности конечного результата.

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. В работах Николаевской и Чистяковой проведены исследования таких задач и проиллюстриро-

вано практическое применение эмулируемой арифметики [14].

Проверка полноты поддержки IEEE 754. Для того, чтобы арифметика с плавающей запятой удовлетворяла стандарту IEEE 754 необходима поддержка со стороны аппаратуры и компилятора. Иногда, даже если компилятор поддерживает арифметику IEEE 754, по умолчанию эта поддержка выключена для «оптимизации скорости». Уильям Кэхэн предложил программу на Си и Фортране, которая позволяет проверить соответствие связки «архитектура – компилятор – опции» стандарту IEEE754 [15]. Аналогичная программа доступна и для графических процессоров [3]. Так, например, компилятор Intel (icc) по умолчанию использует «расслабленную» модель IEEE754, и в результате не все тесты выполняются. Опция «–fp–model precise» позволяет компилировать программу с точным соответствием стандарту. В компиляторе GCC использование опции «–ffast–math» приводит к несоответствию IEEE 754.

Заключение

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. Наличие нуля, денормализованных чисел, специальных чисел (бесконечностей и неопределенности), устойчивых алгоритмов округления и требований к погрешности арифметических операций позволяет аналитическими методами гарантировать точность решений в рамках этой арифметики. 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.

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» // Кибернетика и системный анализ. – 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. – N 23, Vol. 1. – P. 5 – 48.
6. *Severance C.* An Interview with the Old Man of Floating-Point // IEEE Computer. – 1998. – N 1. – P. 114 – 115.
7. IEEE 754-2008 Standard for Floating-Point Arithmetic.
8. *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.
9. *Higham N.* Accuracy and Stability of Numerical Algorithms.
10. http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm.
11. http://software.intel.com/en-us/videos/tim-mattson-floating-points-arent-real/.
12. *Kahan W.* Implementation of algorithms. Technical Report 20, Department of Computer Science, University of California, Berkeley, CA, USA, 1973.
13. *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.
14. *Николаевская Е.А., Чистякова Т.В.* Программно-алгоритмические методы повышения точности компьютерных решений // Кибернетика и системный анализ. – 2009. – № 6. – С. 172 – 176.
15. http://www.cs.unc.edu/~ibr/projects/paranoia/.

Получено 21.06.2011

Об авторе:

Ющенко Руслан Андреевич,
кандидат физико-математических наук,
научный сотрудник.

***Место работы автора*:**

Институт кибернетики
им. В.М. Глушкова НАН Украины,
03187, Киев,
проспект Академика Глушкова, 40.
Тел.: +38044 526 3603,
e-mail: pgt@ukr.net