引用格式:周泉,杨靓,何卫强. 平方根倒数速算法的精度优化[J]. 微电子学与计算机,2019,36(5):64-69. [ZHOU Q, YANG L, HE W Q. Accuracy optimizing for fast inverse square root algorithm[J]. Microelectronics & Computer, 2019, 36(5): 64-69.]

平方根倒数速算法的精度优化

周 泉,杨 靓,何卫强

(西安微电子技术研究所,陕西 西安 710065)

摘 要: 平方根倒数速算法是计算机数值计算中的一种常用方法. 文中在对平方根倒数速算法的原理进行分析的基础上,提出一个新的算法常数来提升计算结果的统计精度. 实验结果表明,本文提供的算法常数相比于原有的算法常数能将批量数据处理的平均精度提升大约30%~40%,而针对单个数据的计算,本文方法的意义在于能从一个更大的概率(平均约77%)上获得比原有方法更加精确的结果.

关键词: 平方根倒数;0x5f3759df;0x5f34ff59;卡马克常数;精度优化

中图分类号: TP301.6 **文献标识码**: A **文章编号**: 1000-7180(2019)05-0064-06

DOI:10.19304/j.cnki.issn1000-7180.2019.05.014

Accuracy optimizing for fast inverse square root algorithm

ZHOU Quan, YANG Liang, HE Wei-qiang

(Xi'an Microelectronics Technology Institute, Xi'an 710065, China)

Abstract: The fast inverse square root algorithm is a common method in computer numerical calculation. Based on the principle analysis of the fast inverse square root algorithm, a new algorithm constant is proposed to improve the statistical accuracy of the calculation results. The experimental results show that the algorithm constant provided by this paper can increase the average precision of batch data processing by about 30%~40% compared with the original algorithm constant. For the calculation of single data, the significance of the method in this paper is to obtain a more accurate result than the original method on a larger probability (average of about 77%).

Key words: inverse square root; 0x5f3759df; 0x5f34ff59; carmack constant; accuracy optimizing

1 引言

平方根倒数速算法(Fast Inverse Square Root, FISqrt)是计算形如$1/\sqrt{x}$的取值的一种快速算法,其中x为正浮点数,且按照IEEE754标准[1]格式表示. FISqrt在数值计算中广泛存在,其典型应用是计算正规化向量. 该算法最初发现于一个被称为"Quake-3"的游戏源代码中[2],但是该算法的原作者到目前为止尚未真正明确. 现在已经追溯到该算法最早的应用存在于美国SGI公司上世纪九十年代初的Indigo多媒体工作站中,但是彼时的FISqrt算法只是由Gary Tarilli进行了实现,且Gary Tarilli也坦诚,他只是对算法中的关键常数作了一定改进,并非原作者[2]. FISqrt算法的C语言描述如图1所示,程序的计算结果为x的平方根倒数值,其中的常数"0x5f3759df"被称为"卡马克常数"或"魔术数字". 该算法的简洁高效是毫无疑问的,它与Visual C++中1.0/sqrt(x)相比,计算速度快了4倍[3].

平方根倒数的计算普遍存在于图像视频等多媒体处理中,因此Intel在x86架构面向多媒体的SSE指令集中引入了平方根倒数计算指令[4-5],从硬件上支持平方根倒数的计算以获得更快的计算速度. 目前平方根倒数的硬件实现方法具有较多的研究[6-8],

收稿日期: 2018-08-22; 修回日期: 2018-09-10

其中 Zafar[8] 等人根据下列所示的软件算法的思路，直接进行硬化设计，并对设计进行评估，证明了其可行性. 在目前的大多数 RISC 处理器中，为了避免过多的硬件开销，一般不会提供类似平方根倒数计算的硬件支持，这时主要采用 FISqrt 算法来实现浮点数平方根倒数的快速计算.

```
float InvSqrt (float x)
{
   float xhalf= 0.5f * x;
   int i = *(int*)&x;                  //以整数方式读取x，并赋值给i
   i = 0x5f3759df - (i>>1);            //计算以整数表示的牛顿迭代初值
   x = *(float*)&i;                    //以浮点方式读取i，并赋值给x
   x = x*(1.5f -(x half* x * x));      // 一次牛顿迭代以提高精度
   return x;
}
```

在图像视频处理中，往往需要对批量数据进行相同操作，平方根倒数的计算也不例外；例如，在相关图像匹配中计算结果的归一化[9] 时就需要对整个矩阵的每个元素求平方根倒数.

2　FISqrt 算法的理论基础

2.1　浮点数与整数的表示

FISqrt 算法中的浮点数采用 IEEE754 标准[1] 格式表示，且浮点数与整数必须具有相同的比特位数，本为以 32 位数据表示为例进行介绍.

图 1 所示为 IEEE754 标准的单精度浮点数表示格式，其中最高位为符号位 s，当 s 为 0 时表示正数，当 s 为 1 时表示负数. 接下来的 8 位表示带偏移量的指数 E；32 为单精度浮点数中偏移量为固定值 $B=127$，实际求得的指数值为 $e=E-B$. 最后 23 位代表尾数 m，这里 $0\leqslant m<1$，且尾数部分包含一个未表示出来的隐藏位 1. 因此浮点数可以表示为 $x=(-1)^s\times(1+m)\times 2^{E-B}$.

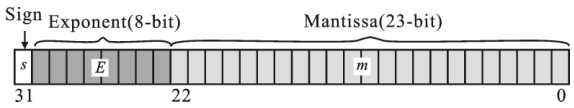


图 1　IEEE754 单精度浮点数格式

整数的表示相对简单，有符号数的最高位为符号位，剩下的 31 位全是数值位；无符号数的所有位数均为数值位. 如果将图 2 的浮点数看作一个整数，那么它由 8 bit 的整数 E 左移 23 位后加上 23 bit 的 m 的整数形式得到. 如果将 m 的整数形式用 M 表示，则所得的整数 $I=E\times 2^{23}+M$，这里的符号位 s 默认为 0，因为平方根倒数的求解只针对正数进行.

2.2　FISqrt 算法原理

FISqrt 算法只针对正浮点数进行，符号位为 0，这样图 2 所示的浮点数的实际数值为 $x=(1+m)\times 2^e$，其中 $e=E-B$；若将其看作一个整数，则实际数值为 $I=EL+M$，其中 $L=2^{n-b-1}$，$M=mL$，当采用 32 位的数据表示时 $n=32$，$b=8$，且 $B=2^{b-1}-1$.

FISqrt 算法需要计算 $y=1/\sqrt{x}$ 的值，其中 x 为正浮点数. 对该式子两边同时取 2 为底的对数得到：

$$\log_2(y)=-\frac{1}{2}\log_2(x) \tag{1}$$

x 和 y 均为浮点数，可以分别表示为 $x=(1+m_x)\times 2^{e_x}$ 和 $y=(1+m_y)\times 2^{e_y}$，将 x，y 的指数形式带入(1) 得到

$$\begin{aligned}&e_y+\log_2(1+m_y)\\&=-\frac{1}{2}e_x-\frac{1}{2}\log_2(1+m_x)\\&0\leqslant m_x<1,0\leqslant m_y<1.\end{aligned} \tag{2}$$

设常数 k 满足 $0\leqslant k<1$，则用 k 可以粗略的表示 $\log_2(1+k)$ 的值，即 $\log_2(1+k)\approx k$. 为了让这个粗略的表示更加合理，可以采用一个常数 δ 来调和 $\log_2(1+k)$ 与 k 之间的误差，即用直线 $y=k+\delta$ 来逼近曲线 $y=\log_2(1+k)$. 如图 3 所示，从图中可以看出直线对曲线的逼近程度随着 δ 的变化而变化. 给定一个合理的 δ 值，当 k 在 $[0,1)$ 中变化时直线与曲线实际上非常接近，即 $\log_2(1+k)\approx k+\delta$，其中 δ 为一个可选的常数.

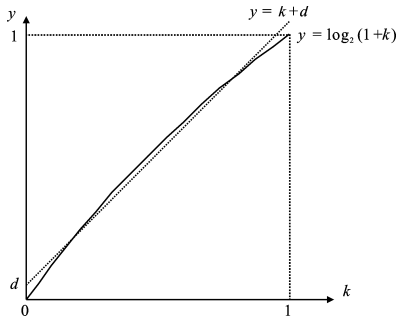


图 3　$y=\log_2(1+k)$ 与 $y=k+\delta$ 的函数图象

采用上述的近似表示之后，式(2) 可以写成：

$$e_y+m_y+\delta\approx-\frac{1}{2}e_x-\frac{1}{2}(m_x+\delta) \tag{3}$$

将 $e_y=E_y-B$，$m_y=M_y/L$，$e_x=E_x-B$，$m_x=M_x/L$ 代入式(3)，并整理得：

$$E_yL+M_y\approx\frac{3}{2}(B-\delta)L-\frac{1}{2}(E_xL+M_x) \tag{4}$$

式(4) 中的左端为将浮点数 y 看成整数 I_y 的一种表示，而右端的 E_xL+M_x 为给定的浮点数 x 的一种整数表示，且给定数据表示的格式时 B 和 L 都为已知. 因此，式(4) 表明如果给定式子 $y=1/\sqrt{x}$，那

么可以根据 x 的整数表示形式 $I_x = E_x L + M_x$ 导出 y 的整数表示形式 I_y 的近似值. 如果将这个导出的 I_y 的近似值定义成一个浮点数，那么它自然也能成为浮点数 y 的近似值，即 x 的平方根倒数的近似值. 应用中如果数值精度不够理想，可以用该近似值为基础进行牛顿迭代，提高结果的精度，这就是 FISqrt 算法的原理.

设 $R = \frac{3}{2}(B-\delta)L$，那么式(4)可以写成：

$$I_y \approx R - \frac{1}{2} I_x \tag{5}$$

式(5)与图1中的第5行代码“$i = 0x5f3759df - (i >> 1)$”相对应，该行代码实现的是平方根倒数的初值估计. 从中可以看出原始代码中的常数“$0x5f3759df$”为 R 的值，即 $R = \frac{3}{2}(B-\delta)L = 0x5f3759df$. 由此可以计算出

$$\begin{aligned}\delta &= B - \frac{2 \times 0x5f3759df}{3L} \\ &= 127 - \frac{2 \times 0x5f3759df}{3 \times 2^{23}} \\ &= 0.045\ 046\ 567\ 916\ 870\ 117\ 187\ 5\end{aligned}$$

在目前的应用中，R 取“$0x5f3759df$”的原因尚未真正明确，对其取值的合理性更没有相应的理论证明，这也是很多研究人员对 FISqrt 算法的关键兴趣所在.

表1根据上述的 FISqrt 算法原理列举了若干个数据的平方根倒数的初始估计值与标准值的对比，从这组数据中可以看出平方根倒数的预估初始值与标准值的误差均未超过 3.5%.

表1 平方根倒数的初始估计值与标准值的对比

x	I_x（浮点格式的整数）	I_y 的预估值 I_y' ($I_y' = R-(I_x >> 1)$)	I_y' 代表的浮点值 $y' = \text{flaot}(I_y')$	平方根倒数标准值 $y = 1.0/\text{sqrt}(x)$	误差(%)：$\eta = 100\% \times \lvert y - y' \rvert / y$
1.0	$0x3f800000$	$0x3f7759df$	0.966 215	1.000 000	3.378 5
16.0	$0x41800000$	$0x3e7759df$	0.241 554	0.250 000	3.378 5
0.075 83	$0x3d9b4cc2$	$0x4069b37e$	3.651 580	3.631 445	0.554 5
67.33 3	$0x4286aa7f$	$0x3df404a0$	0.119 149	0.121 867	2.229 9
481.478	$0x43f0bd2f$	$0x3d3efb48$	0.046 626	0.045 573	2.310 3
702 395.239	$0x492b7bb4$	$0x3aa19c05$	0.001 233	0.001 193	3.335 0

3 R 常数的改进方案

根据当前算法中广泛采用的 R 值，求出了误差调和常数 δ 的值，但实际上 δ 是一个可选的常数. 因此，存在一个问题是如何选择 δ 的值，使得当 $0 \leqslant k < 1$ 时，用直线 $y = k + \delta$ 来近似表示曲线 $y = \log_2(1+k)$ 会更加合理，从而进一步减小平方根倒数初始估计的误差. 通过上一节的分析可以发现，FISqrt 算法中影响初始值估计精度的关键在于由 δ 值确定的直线是否能很好的逼近曲线. 可以设想如果直线和曲线完全重合(没有误差)，那么计算得到的初始值也就精确的代表平方根倒数值. 从图3可见 δ 代表直线在 y 轴上的截距，改变 δ 值可以改变曲线的上下位置. 如果采用一种方法选择一个更加优化的 δ 值，使得直线更加精确的代表曲线，并通过 δ 计算出一个更加优化的 R 值，便能达到提高初始估计精度的目的.

假设实际应用中需要计算平方根倒数的数据元素个数为 n，其中每个元素的浮点格式中的尾数 m_i 确定了 $y(m_i) = m_i + \delta$ 对 $f(m_i) = \log_2(1+m_i)$ 的一个近似表示，其中 $0 \leqslant m_i < 1$，$0 \leqslant i \leqslant n-1$.

为了提高整批数据的求解质量需要让 $y(m_i) = m_i + \delta$ 近似表示 $f(m_i) = \log_2(1+m_i)$ 的平均误差 $\frac{1}{n}\sum_{i=0}^{n-1} \lvert y(m_i) - f(m_i) \rvert$ 尽可能的小. 为了避免绝对值的运算，将平均误差改为均方误差进行计算，如果用 $\bar{\varepsilon}$ 表示均方误差，则需要让

$$\begin{aligned}\bar{\varepsilon} &= \frac{\sum_{i=0}^{n-1} [y(m_i) - f(m_i)]^2}{n} \\ &= \frac{\sum_{i=0}^{n-1} [m_i + \delta - \log_2(1+m_i)]^2}{n}\end{aligned} \tag{6}$$

取到最小值. 针对具体应用，式(6)中的 m_i 和 n 均为已知，变化的只有 δ，因此需要取一个能使 $\bar{\varepsilon}$ 最小的 δ，并将由此计算出的 R 用到算法编程中去. 但是在讨论算法的一般情况的时候，并不知道 m_i 和 n 的具体取值，为了获得一个普遍适用的 δ 常数，可以

将浮点数的尾数 m_i 视为在区间[0，1）上随机均匀分布，并且在求平均值$\bar{\varepsilon}$的过程中对区间[0，1）上的所有可能的m_i值进行遍历求和，这样$n=+\infty$，对无穷多项的求和操作相应的变成积分操作，并对整个区间长度求均值，即

$$\begin{aligned}\bar{\varepsilon}&=\frac{\sum_{i=0}^{+\infty}[y(m_i)-f(m_i)]^2}{\infty}\\&=\frac{1}{1-0}\int_0^1[y(m)-f(m)]^2dm\\&=\int_0^1[m+\delta-\log_2(1+m)]^2dm\end{aligned}\tag{7}$$

为了求得$\bar{\varepsilon}$取最小值时的δ值，由式(7)得

$$\begin{aligned}\bar{\varepsilon}'&=\frac{d\bar{\varepsilon}}{d\delta}=\frac{d[\int_0^1[m+\delta-\log_2(1+m)]^2dm]}{d\delta}\\&=\frac{d[\int_0^1[\delta^2+2\delta(m-\log_2(1+m))]dm]}{d\delta}\\&=2\delta+2\left[\frac{1}{2}m^2-\frac{1+m}{\ln 2}(\ln(1+m)-1)\right]_0^1\\&=2\delta+\frac{2}{\ln 2}-3\end{aligned}\tag{8}$$

令$\bar{\varepsilon}'=0$，有$2\delta+\frac{2}{\ln 2}-3=0$，解得$\delta=\frac{3}{2}-\frac{1}{\ln 2}\approx 0.057\ 304\ 959\ 111\ 036\ 592\ 64$. 将$\delta$的值代入式(7)求得$\bar{\varepsilon}_{\min}=\frac{1}{(\ln 2)^2}-\frac{3}{2\ln 2}+\frac{1}{12}\approx 0.000\ 659\ 753\ 005\ 496$，即在区间[0，1）上利用本文提供的$\delta$值得到的直线$y=k+\delta$来近似表示曲线$y=\log_2(1+k)$，能够使均方误差最小，且最小均方误差值约为 0.000 659 753 005 496.

根据上述δ的值可得$R=\frac{3}{2}(B-\delta)L\approx 159\ 730\ 876\ 1=0x5f34ff59$. 在 FISqrt 算法中采用这里所给出的$R$值，从理论上可以减小批量数据的平方根倒数初始估计值与精确值之间的均方误差，从而减小平均误差. 由于初始估计更加优化，因此用该初始估计值进行牛顿迭代后的值也应该表现出更好的精度性能. 本文提出的方法在单个数据求平方根倒数上的意义在于能够从更大的概率上获得更加精确的初始估计.

4　结果验证

为了验证所提出的 R 参数对 FISqrt 算法平均精度的提升，本文的实验中采用五组随机生成的浮点数据进行计算，求解初始估计值以及迭代后的值与标准平方根倒数之间的误差百分比. 每组随机数中包含 20 000 个浮点数，取值区间为(50.0，10 000.0). 精度验证中根据不同的 R 取值，求解这 20 000 个浮点数据的平方根倒数预估值及其一次、两次牛顿迭代后的值，将计算结果与标准平方根倒数值进行比较，并计算出平均误差百分比

$$100\%\times\frac{\sqrt{x_i^{n-1}}}{n}\sum_{i=0}|a_i-1/\sqrt{x_i}|\tag{9}$$

式中，$n=20\ 000$；a_i为实验中计算出的值；y_i为标准平方根倒数值；即$y_i=1/\sqrt{x_i}$；x_i为位于区间(50.0，10 000.0)的原始随机浮点数.

为了实现计算精度比较，每组数据都针对R的四个不同取值进行计算，这四个R值分别为：$0x5f3759df$、$0x5f37642f$、$0x5f375a86$、$0x5f34ff59$，其中$0x5f3759df$为原始算法中的数据，$0x5f34ff59$为本文提出的数据. $0x5f37642f$为 Lomont C. 对算法进行分析得出的数据，但是 Lomont C. 通过实验发现这个数据并不比原始的$0x5f3759df$能获得更好的计算精度，因此他又通过穷举搜索的方式发现与$0x5f37642f$邻近的另外一个数$0x5f375a86$相比于原有的$0x5f3759df$在计算精度上稍微有改进. 关于$0x5f37642f$和$0x5f375a86$这两个R常数的详细论述见文献[3].

图 4 对初始估计值的平均误差百分比进行了比较，每个柱状条代表的数值均按照式(9)进行计算. 从中可以看出本文提出的R参数相比于另外三个R参数将误差百分比减小约三分之一，且五组计算数据显示了这种优势的稳定性. 从实验结果看，本文提供的R参数计算出的平方根倒数初始估计值与标准的平方根倒数值相比，平均误差不到 1.6%，由其它三个R参数得到的预估值误差都约为 2.3%.

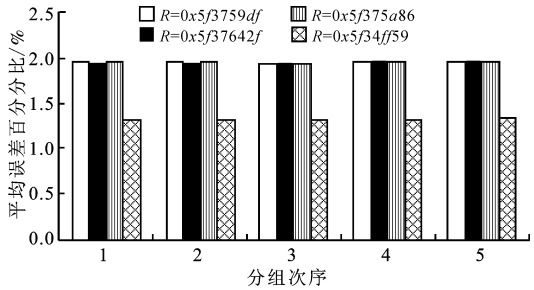


图 4　初始估计与精确值的误差百分比

图 5 为一次牛顿迭代后的精度百分比对比. 本文提供的R参数相比于其它三个R参数将误差百分比降低了超过 40%. 图 6 为两次牛顿迭代后的精度

百分比对照，本文提供的 R 参数同样将误差百分比降低了超过 30%，且两次迭代后的结果与标准的结果相比，误差仅略高于百万分之一.

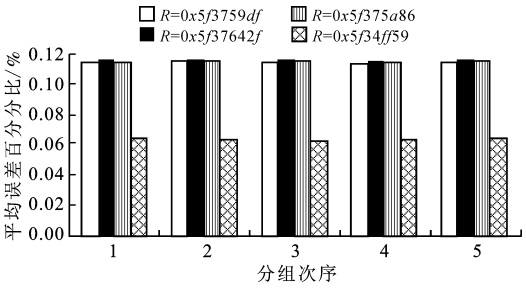


图 5 一次迭代值与精确值的误差百分比

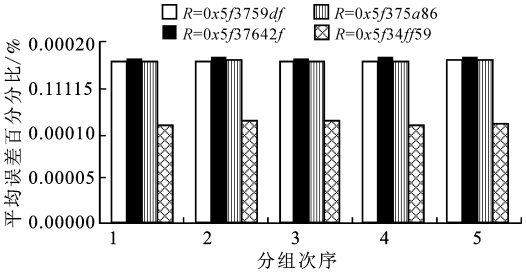
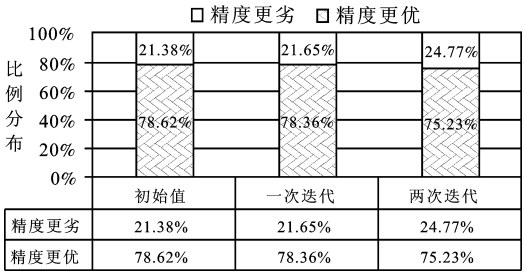


图 6 两次迭代值与精确值的误差百分比

本文提出的 R 常数对单次计算平方根倒数的意义在于能够从更大的概率上获得一个更加优化的初始估计及其迭代值. 图 7 统计了 20 000 次计算中本文给出的 R 值相比于目前使用的 $R = 0x5f3759df$ 的精度优劣占比分布情况，其中 20 000 次计算的源数据仍然是位于区间 $(50.0, 10\,000.0)$ 中的随机浮点数. 从图中可以看出当采用本文的 $R = 0x5f34ff59$ 时，在 20000 次计算中占 78.62% 的初始估计值比 $R = 0x5f3759df$ 拥有更高的精度，而剩余 21.38% 的精度不如 $R = 0x5f3759df$. 经过一次迭代和两次迭代后，$R = 0x5f34ff59$ 时，整批数据中仍然分别有 78.36% 和 75.23% 精度更优.



	初始值	一次迭代	两次迭代
精度更劣	21.38%	21.65%	24.77%
精度更优	78.62%	78.36%	75.23%

图 7 $R = 0x5f34ff59$ 相比于 $R = 0x5f3759df$ 在计算精度上更优 / 劣的占比分布

这三个比例的平均值超过了 77%，通过这样的一个样本统计可以得出采用本文的 R 值能够在约 77% 的概率下获得一个比 $R = 0x5f3759df$ 更加精确的结果. 随着迭代次数的增加，计算的结果会迅速的收敛到平方根倒数的标准值，这一比例优势会逐渐降低.

5 FISqrt 算法的进一步探讨

由 FISqrt 算法的原理，可以进一步考虑计算 $y = x^p$（x 为正浮点数），并根据 FISqrt 算法作类似推导可得 $E_yL + M_y \approx (1-p)(B-\delta)L + p(E_xL + M_x)$，即 $I_y \approx R + pI_x$，这里 $R = (1-p)(B-\delta)L$，同时后续的牛顿迭代也相应的表示为 $i_{n+1} = (1-p)i_n + pxi_n^{1-1/p}$. 将指数用 p 表示之后，当 p 取某些值时初始估计的计算中 pI_x 可能不能用移位运算来实现，同时后续的牛顿迭代，特别是 $i_n^{1-1/p}$ 项的计算也会变得很耗时，从而大大降低了算法的适用性，但是这种方法的理论意义还是值得保留的. FISqrt 算法不仅提供了一种快速计算浮点数平方根倒数的方法，更重要的是还提供了一种将浮点数据位当作整数进行运算，从而充分利用浮点格式的思想方法. 根据以上叙述，计算平方根倒数只是 $p = -1/2$ 的一种特例；如果令 $p = 0$，那么有 $y = x^0 = 1$，另一方面 $I_y \approx R + pI_x = (B-\delta)L$，将本文的 δ 值带入得 $I_y \approx 0x3f78aa3b = (0.971\,3\cdots)_{float} \approx x^0 = 1$，即 I_y 所代表的浮点数与 $x^0 = 1$ 接近，近似值的计算依然成立.

该算法的一项约束条件是计算 $R + pI_x$ 得到预估值时一定不能改变浮点格式表示中的符号位，必须保证符号位为 0，算法才能正确运行. 例如当 $p = -1$ 时（计算 x 的倒数），采用本文给出的 δ 值可以计算得到 $R = 0x7ef15476$，因此在计算预估初值的时候需要计算 $i_0 = 0x7ef15476 - I_x$，如果某个浮点数 x 的整数表示形式 $I_x > 0x7ef15476$，那么将导致 i_0 为负数，而负整数在不同的机器上可能会采用不同的表示形式，一般最高位的符号位都为 1，后续的牛顿迭代中再将 i_0 解释成浮点数已经不能很好的表示 x^{-1} 的初始近似值. 不过值得注意的是，如果 $I_x > 0x7ef15476$，此时 x 指数位的值大于等于 253，这样的浮点数虽然存在，但在一般的应用中也很少见. 如果事先明确知道待处理的数据不会出现那么大的数值，令 $p = -1$，采用这种方法计算一个数的倒数，或者进而实现除法计算（被除数乘以除数的倒数）也是一种很高效的方法，甚至在牛顿迭代初值的计算中根本不用移位，且牛顿迭代的形式也很简单. 当然，计算一个数的倒数也可以将 FISqrt 算法的计算

结果平方后得到，但是采用前一种方法会更加高效.

影响算法可靠性的情况还会发生在计算 $R+pI_x$ 时由于结果数值过大，产生了向最高位符号位的进位，同样会导致计算的失败. 在平方根倒数的计算中 $p=-1/2$，计算预估初值时需要计算 $i_0=0x5f34ff59-(I_x>>1)$，由于 $0\leqslant(I_x>>1)\leqslant 0x3fffffff$，因此 $0x1f34ff5a\leqslant i_0\leqslant 0x5f34ff59$，对浮点格式中的符号位无影响，且除去符号位之外的比特位能够有效表示 i_0 的数值，从而保证了算法的可靠执行. 再比如当 $p=1/2$ 时，需要计算 $i_0=0x1fbc551e+(I_x>>1)$，从而有 $0x1fbc551e\leqslant i_0\leqslant 0x5fbc551d$，算法同样能正确执行，因此这也为求解平方根提供了一种初值获取方法. 总之，如果要将 FISqrt 算法的思想扩展到计算指幂函数 $y=x^p$ 的值（x 为正浮点数），需要满足 $0\leqslant R+pI_x\leqslant 0x7fffffff$.

本文对 FISqrt 算法的叙述与实验都是以 32 位单精度为例进行说明的，但是很明显 FISqrt 算法不仅限于单精度的浮点数，对 64 位，128 位等浮点数同样适用，只需根据具体的数据表示格式并结合 δ 计算出 R 的值即可.

6　结束语

本文从 FISqrt 算法的原理出发，探索了一个新的算法常数 $0x5f34ff59$，能够从统计角度有效提升 FISqrt 算法的计算精度. 本文方法对单个数据的平方根倒数计算的意义在于能够从更大的概率上获得一个比现有方法更加精确的计算结果. 文中对所提出的方法提供了完整的理论支持和实验验证，最后对 FISqrt 算法的应用作了进一步讨论.

参考文献：

[1] IEEE Computer Society. IEEE Standard 754-2008: IEEE Standard for Floating-Point Arithmetic [S]. New York: IEEE, 2008.

[2] SOMMEFELDT R. Origin of quake3's fast InvSqrt(), Beyond3D [EB/OL]. (2006-11-29). [2018-09-03]. http://www.beyond3d.com/content/articles/8/

[3] LOMONT C. Fast inverse square root [R]. West Lafayette: Purdue University, Department of Mathematics, 2003.

[4] THAKKAR S, HUFF T. Internet streaming SIMD extensions [J]. Computer, 1999, 32(12): 26-34.

[5] RAMAN S K, PENTKOVSKI V, KESHAVA J. Implementing streaming SIMD Extensions on the pentium III processor [J]. IEEE Micro, 2002, 20(4): 47-57.

[6] ERCEGOVAC M D, LANG T, MULLER J M, et al. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers [J]. IEEE Transactions on Computers, 1998, 49(7): 628-637.

[7] TAKAGI N. A hardware algorithm for computing reciprocal square root [C]//Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH-15). Vail, Colorado: IEEE, 2001: 94-100.

[8] ZAFAR S, ADAPA R. Hardware architecture design and mapping of fast inverse square root algorithm [C]//IEEE International Conference on Advances in Electrical Engineering (ICAEE). Vellore, India: IEEE, 2014: 1-4.

[9] ZHOU Q, YANG L, YAN X. Reconfigurable instruction-based multicore parallel convolution and its application in real-time template matching [J]. IEEE Transactions on Computers, doi: 10.1109/TC.2018.2844351.

作者简介：

周　泉　男，(1987-)，博士研究生. 研究方向为计算机系统结构、数字集成电路设计、高性能计算.

E-mail: zhouquan1234.ok@163.com.

杨　靓　男，(1975-)，博士，研究员. 研究方向为计算机系统结构、数字集成电路设计.

何卫强　男，(1987-)，硕士. 研究方向为计算机系统结构、数字集成电路设计.