Andriy HRYNCHYSHYN[1], Oleh HORYACHYY[2],
Oleksandr TYMOSHENKO[3]

Supervisor: Leonid MOROZ[4]

EFEKTYWNY ALGORYTM DLA SZYBKIEGO OBLICZANIA ODWROTNOŚCI PIERWIASTKA KWADRATOWEGO

Streszczenie: Funkcje pierwiastka kwadratowego i odwrotności pierwiastka kwadratowego dla liczb zmiennoprzecinkowych są ważne dla wielu aplikacji. W artykule przedstawiono kilka modyfikacji oryginalnego algorytmu tzw. Fast Inverse Square Root (FISR) w celu poprawy jego dokładności dla liczb zmiennoprzecinkowych pojedynczej precyzji (typ float) standardu IEEE 754. Proponowane algorytmy są przeznaczone dla platform bez wsparcia sprzętowego tychże funkcji, takich jak mikrokontrolery i układy FPGA, ale z szybkimi operacjami zmiennoprzecinkowymi dla dodawania / odejmowania, mnożenia oraz FMA (fused multiply-add).

Słowa kluczowe: odwrotność pierwiastka kwadratowego, arytmetyka zmiennoprzecinkowa, algorytm FISR, stała magiczna, aproksymacja, funkcja FMA

AN EFFICIENT ALGORITHM FOR FAST INVERSE SQUARE ROOT

Summary: The square root and inverse square root functions for floating-point numbers are important for many applications. This paper presents a few modifications of the original Fast Inverse Square Root (FISR) algorithm to improve its accuracy. Normalized single precision floating-point numbers (type *float*) of IEEE 754 standard are considered. The proposed algorithms are designed for platforms without hardware support of these functions, such as microcontrollers and FPGAs, but with fast floating-point operations for addition/subtraction, multiplication, and fused multiply-add (FMA).

Keywords: inverse square root, floating-point arithmetic, FISR algorithm, magic constant, approximation, FMA function

---

[1] Ing., Lviv Polytechnic National University, Department of Information Technology Security, hrynchyshyn.a@gmail.com
[2] Ing., Lviv Polytechnic National University, Department of Information Technology Security, oleh.y.horiachyi@lpnu.ua
[3] Lviv Polytechnic National University, Department of Information Technology Security, oleksandr.tymoshenko.kb.2016@lpnu.ua
[4] Prof., Lviv Polytechnic National University, Department of Information Technology Security, moroz_lv@lp.edu.ua

1. Introduction

A floating-point arithmetic is widely used in many applications, such as 3D graphics, digital signal processing, and scientific computing [1-3]. Common algorithms of such arithmetic are, in particular, algorithms for calculating elementary functions, including inverse square root [3-14]:

$$y = \frac{1}{\sqrt{x}}. \tag{1}$$

All these algorithms are iterative and require the formulation of an initial approximation. The more precise the initial approximation is, the fewer repetitions (iterations) are needed to calculate the function. In most cases, the initial approximation is formed using look-up tables (LUTs), which require memory. However, there is a known group of iterative algorithms without the use of LUT to obtain an initial approximation. They initially obtain an approximation using the so-called magic constant and are then used to implement an iterative process with the Newton-Raphson formulae. The algorithm calculates the initial value based on the magic constant using integer arithmetic and, after switching to a floating-point arithmetic, the resulting approximation is introduced into the iterative process.

The inverse square root algorithms are suitable for software and hardware implementations, for example, in microcontrollers that lack an FPU and in FPGAs.

In this article, we discuss the algorithm for calculating the inverse square root using a magic constant with reduced errors for floating-point numbers represented in the IEEE 754 standard.

2. Known algorithms

The best known version of the algorithm called Fast Inverse Square Root (FISR), which was implemented in the computer game Quake III Arena [7], is given below:

```
1. float InvSqrt(float x){
2.   float halfnumber = 0.5f*x;
3.   int i = *(int*)&x;
4.   i = 0x5f3759df - (i >> 1);
5.   x = *(float*)&i;
6.   x = x*(1.5f - halfnumber*x*x);
7.   x = x*(1.5f - halfnumber*x*x);
8.   return x;
9. }
```

This InvSqrt code written in C/C++ implements a fast algorithm for calculating the inverse square root. In line 3, we convert the bits of the input variable x of type *float* to the variable i of type *int*. And in line 4 we determine an initial approximation y_0 (which subsequently becomes the object of the iterative process) of the inverse square root, where $\mathrm{R} = 0\mathrm{x5f3759df}$ is a "magic constant". Note that from this point onwards, the variable x is regarded as the approximation of the function y. Line 5

converts the bits of the variable i of type *int* into the variable y_0 of type *float*. Lines 6 and 7 contain two classic successive Newton-Raphson iterations.

If the maximum relative error of calculations after the second iteration is designated by δ_{2max}, then the accuracy of this algorithm is only

$$|\delta_{2max}| = 4.86 \cdot 10^{-6}, \quad \text{or} \quad -\log_2(|\delta_{2max}|) = 17.65 \tag{2}$$

correct bits. The higher accuracy of the algorithm with magic constants is achieved in [9], namely 20.37 correct bits. Yet the improved algorithm with modified Newton-Raphson iterations from [9] is more precise:

```
1.  float InvSqrt2(float x){
2.    float halfnumber = 0.5f*x;
3.    int i = *(int*)&x;
4.    i = 0x5f376908 - (i >> 1);
5.    x = *(float*)&i;
6.    x = x*(1.5008789f - halfnumber*x*x);
7.    x = x*(1.5000006f - halfnumber*x*x);
8.    return x;
9.  }
```

The accuracy of the algorithm is approximately

$$|\delta_{2max}| = 7.37 \cdot 10^{-7}, \quad \text{or} \quad -\log_2(|\delta_{2max}|) = 20.37 \tag{3}$$

correct bits.

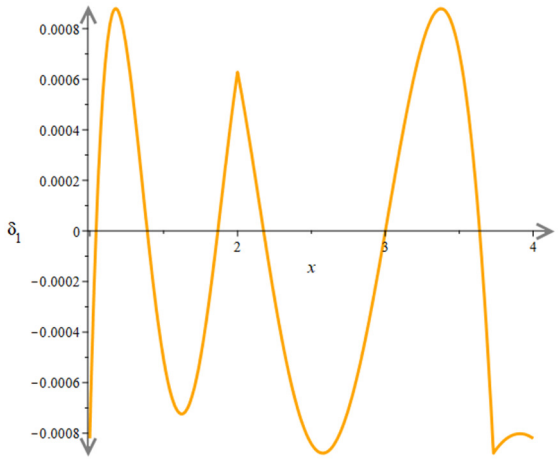

Figure 1. The graph of the relative errors of the InvSqrt2 algorithm after the first iteration for $x \epsilon [1,4)$

3. Analytical description of the algorithm

First of all, in this section, we briefly describe the main results of [8-9]. Positive floating-point numbers that may be represented in form

$$x = (1 + m_x) \cdot 2^{E_x} \tag{4}$$

are considered, where $m_x \epsilon [0,1)$ is the fractional part of the mantissa and E_x is the integer exponent. In other words, x is a binary floating point number written in a normalized exponential form. The exponent is defined by the formula

$$E_x = \lfloor \log_2(x) \rfloor. \tag{5}$$

IEEE 754 standard specifies rules to store real numbers in binary format. For example, the floating-point number x of type *float* is encoded by 32 bits. The first bit corresponds to a sign (sign field). In our case, this bit is equal to zero. The next 8 bits correspond to a biased exponent E_x (exponent field). And the last 23 bits encode a fractional part of mantissa m_x (mantissa field). An integer encoded by these 32 bits, denoted by I_x, is given by

$$I_x = (bias + E_x + m_x) N_m, \tag{6}$$

where $N_m = 2^{23}$ and $bias = 127$. As a matter of fact, the algorithm InvSqrt can be rewritten in the following form (lines from 3 to 7):

$$\begin{aligned} &I_x = \text{convertToInt}(x); \\ &I_{y_0} = R - \lfloor I_x/2 \rfloor; \\ &y_0 = \text{convertToFloat}\left(I_{y_0}\right); \\ &y_1 = 1/2 * y_0 * (3 - x * y_0 * y_0); \\ &y_2 = 1/2 * y_1 * (3 - x * y_1 * y_1). \end{aligned} \tag{7}$$

The result of subtracting an integer $\lfloor I_x/2 \rfloor$ from the magic constant R is an integer I_{y_0}, which is translated again into *float* (lines 4 and 5). It gives the initial (zeroth) piecewise linear approximation y_0 of the function $y = 1/\sqrt{x}$.

To find the behavior of the relative error when calculating y_0 in the entire range of normalized floating-point numbers, as it was proven in [8-9], it is sufficient to describe its behavior in the range $x \epsilon [1,4)$. In this range, there are three piecewise linear analytic approximations of the function y_0:

$$\begin{aligned} &y_{01} = -\tfrac{1}{4}x + \tfrac{3}{4} + \tfrac{1}{8}t, \quad x \epsilon [1,2); \\ &y_{02} = -\tfrac{1}{8}x + \tfrac{1}{2} + \tfrac{1}{8}t, \quad x \epsilon [2,t); \\ &y_{03} = -\tfrac{1}{16}x + \tfrac{1}{2} + \tfrac{1}{16}t, \quad x \epsilon [t,4), \end{aligned} \tag{8}$$

where $t = 2 + 4m_R + 2/N_m$. Here m_R is a fractional part of the mantissa of the magic constant R and $m_R = N_m^{-1}R - \lfloor N_m^{-1}R \rfloor$. It should be mentioned that the maximum relative error of such piecewise linear approximations does not exceed $1/(2N_m)$. Taking into account the value of t, these three equations can be written as

$$y_{01} = -\tfrac{1}{4}x + 1 + \tfrac{1}{2}m_R + \tfrac{1}{4N_m}, \quad x \epsilon [1,2);$$

$$y_{02} = -\frac{1}{8}x + \frac{3}{4} + \frac{1}{2}m_R + \frac{1}{4N_m}, \quad x\epsilon[2, t); \tag{9}$$
$$y_{03} = -\frac{1}{16}x + \frac{5}{8} + \frac{1}{4}m_R + \frac{1}{8N_m}, \quad x\epsilon[t, 4).$$

If the magic constant R is denoted by

$$R = (Q+m_R)N_m, \tag{10}$$

where $Q = \lfloor N_m^{-1}R \rfloor$, then under condition $m_R < 1/2$ for equations (8) $Q = 190$ as in previous algorithms. However, in general case, $0 \le m_R < 1$.

Consider an example when $Q = 190$ and $1/2 \le m_R < 1$. In this situation, according to the theory described in [8-9], equations (9) will have the form

$$y_{01} = -\frac{1}{2}x + 1 + m_R + \frac{1}{2N_m}, \quad x\epsilon[1, x_t);$$
$$y_{02} = -\frac{1}{4}x + 1 + \frac{1}{2}m_R + \frac{1}{4N_m}, \quad x\epsilon[x_t, 2); \tag{11}$$
$$y_{03} = -\frac{1}{8}x + \frac{3}{4} + \frac{1}{2}m_R + \frac{1}{4N_m}, \quad x\epsilon[2,4),$$

where

$$x_t = 2m_R + \frac{1}{N_m}. \tag{12}$$

Let us set the requirement to align all the maxima of the relative error (of both signs) for the first iteration on the first and third sections of the initial approximation (i.e. y_{01} and y_{03}). There are only five such points (as you can see later in Fig. 2). Note that in the algorithms InvSqrt1 and InvSqrt2 only some of these separate maxima for the components y_{01}, y_{02}, and y_{03} are aligned.

For this purpose, the first iteration will be carried out according to the formula

$$y_1 = k_1 y_0 (k_2 - x y_0 y_0). \tag{13}$$

Then, taking into account (11), the equations for piecewise linear approximations of the function y_0 (which has three components) are represented by

$$y_{11} = k_1 y_{01}(k_2 - x y_{01} y_{01});$$
$$y_{12} = k_1 y_{02}(k_2 - x y_{02} y_{02}); \tag{14}$$
$$y_{13} = k_1 y_{03}(k_2 - x y_{03} y_{03}).$$

The task is to find values k_1, k_2, and t, which in turn determines the magic constant R. To do that, we should define relative errors of each component y_{01}, y_{02}, and y_{03}. Using the following formula to calculate the relative error of the function $y = 1/\sqrt{x}$ after the first iteration

$$\delta_1 = y_1 \cdot \sqrt{x} - 1, \tag{15}$$

we obtain the next equations:

$$\delta_{11} = k_1 y_{01}(k_2 - x y_{01} y_{01}) \cdot \sqrt{x} - 1; \tag{16}$$

$$\delta_{12} = k_1 y_{02}(k_2 - x y_{02} y_{02}) \cdot \sqrt{x} - 1; \tag{17}$$

$$\delta_{13} = k_1 y_{03}(k_2 - x y_{03} y_{03}) \cdot \sqrt{x} - 1. \tag{18}$$

This errors have local maxima, in particular, at points

$$x_{11max} = \frac{2}{3} + \frac{2}{3} m_R + \frac{1}{3N_m}, \tag{19}$$

$$x_{13max} = 2 + \frac{4}{3} m_R + \frac{2}{3N_m}. \tag{20}$$

For the second component y_{02}, the point x_t from equation (12) should be taken into account. The relative error δ_{12} has a negative maximum δ_{12t} as well as error δ_{11}. Now substituting (19) in (16), (20) in (18), and (12) in (17), we get the expressions δ_{11max}, δ_{13max}, and δ_{12t}. Then we construct a system of two equations:

$$\begin{cases} \delta_{11max} + \delta_{12t} = 0 \\ \delta_{13max} - \delta_{12t} = 0 \end{cases}. \tag{21}$$

The solution of this system will be exactly the values k_1 and k_2:

$$k_1 = 0.24888502640450491415149326898911 60; \tag{22}$$

$$k_2 = 4.77849063743002298547316564913655 16. \tag{23}$$

Moreover, when $m_R = 0.75$, which corresponds to the value of the magic constant

$$R = 0\text{x}5\text{F}600000, \tag{24}$$

then the maximum relative errors are equal

$$\delta^+_{1max} = 6.50209 \cdot 10^{-4}, \quad \delta^-_{1max} = -6.501887 \cdot 10^{-4}. \tag{25}$$

4. Experimental results

Because of numerical discretization and rounding errors, the relative errors (25) for type *float* in practice are not achievable. In this situation, the parameters of the algorithm may be adjusted to minimize the error. For example, modified values

$$k_1 = 0.24888471; \quad k_2 = 4.7784891 \tag{26}$$

with the same constant R give errors $\delta^+_{1max} = 6.502244 \cdot 10^{-4}$, $\delta^-_{1max} = -6.502372 \cdot 10^{-4}$ which correspond to 10.59 correct bits of the result.

However, our experimental results in the neighborhood of values (22)-(24) show that the best accuracy can be achieved in practice for the following values of parameters:

$$k_1 = 0.248884737;$$

$$k_2 = 4.778488636; \qquad (27)$$
$$R = 0\text{x}5\text{F}5\text{FFFF}8.$$

Then the algorithm with one iteration for C++ has the form:

```
1.  float InvSqrt31(float x){
2.    int i = *(int*)&x;
3.    i = 0x5f5ffff8 - (i >> 1);
4.    float y = *(float*)&i;
5.    y = 0.248884737f*y*(4.778488636f - x*y*y);
6.    return y;
7.  }
```

The maximum values of the relative errors of the algorithm are

$$\delta_{1max}^{+} = 6.501923 \cdot 10^{-4}, \quad \delta_{1max}^{-} = -6.502141 \cdot 10^{-4}. \qquad (28)$$

The graph of the relative errors of the algorithm InvSqrt31 is given below.

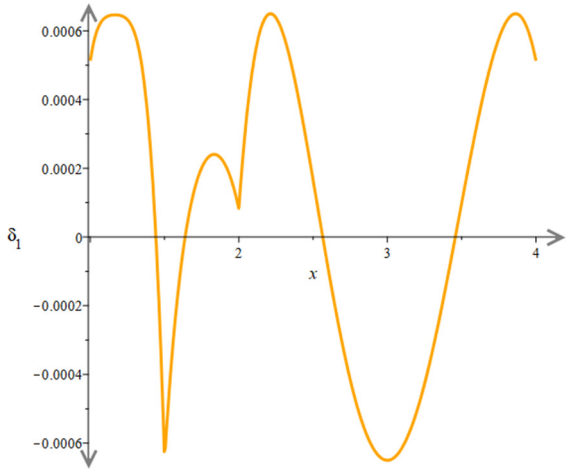

Figure 2. The graph of the relative errors of the InvSqrt31 algorithm for $x\epsilon[1,4)$

The given algorithm has four floating-point multiplications. Compared with the best-known algorithm [9], the maximum error is reduced by 26% (see also Fig. 1) with the same number of multiplications.

By adding a second Newton-Raphson iteration, we obtain the following results:

```
1.  float InvSqrt32(float x){
2.    int i = *(int*)&x;
3.    i = 0x5f5ffff8 - (i >> 1);
4.    float y = *(float*)&i;
5.    y = 0.248884737f*y*(4.778488636f - x*y*y);
6.    float c = x*y;
7.    c = fmaf(y, -c, 1.00000065f);
```

```
8.    y = fmaf(y, 0.5f*c, y);
9.    return y;
10. }
```

The new iteration (lines from 6 to 8) is written with fused multiply-add functions (*fmaf*) to improve the accuracy of the algorithm. InvSqrt32 function also contains one additional parameter.

The maximum values of the relative errors of this algorithm are

$$\delta_{2max}^{+} = 3.687961 \cdot 10^{-7}, \quad \delta_{2max}^{-} = -4.086946 \cdot 10^{-7}. \tag{29}$$

Here, in comparison with the best-known algorithm [9], the maximum errors are reduced by 44.5%. The accuracy of the algorithm is 21.22 correct bits.

It is still possible to improve the accuracy if you use the Householder formula [3] in the second iteration. Then we get the following algorithm:

```
1.  float InvSqrt33(float x){
2.    int i = *(int*)&x;
3.    i = 0x5f5ffff8 - (i >> 1);
4.    float y = *(float*)&i;
5.    y = 0.248884737f*y*(4.778488636f - x*y*y);
6.    float c = x*y;
7.    float r = fmaf(y, -c, 1.0f);
8.    c = fmaf(0.375f, r, 0.5f);
9.    r = r*c;
10.   y = fmaf(y, r, y);
11.   return y;
12. }
```

This algorithm has nine floating-point multiplication operations. The maximum values of the relative errors of the algorithm are

$$\delta_{2max}^{+} = 8.958924 \cdot 10^{-8}, \quad \delta_{2max}^{-} = -8.776532 \cdot 10^{-8} \tag{30}$$

which correspond to 23.41 correct bits. Compared with the algorithm described in [9], the error in InvSqrt33 was reduced by a factor of 8.24.

Based on the formulas given in [3], one can construct an algorithm that also has nine multiplications. However, the maximum relative errors of our InvSqrt33 algorithm will be 2.44 times less (twice less if the original algorithm uses functions *fmaf*).

5. Conclusions

To conclude, a few modifications of the FISR algorithm with a magic constant were proposed in this paper. The purpose of these algorithms is to calculate the inverse square root for floating-point numbers in the format of IEEE 754 standard. It should be noted that only normalized numbers of type *float* were considered in the paper. However, other versions of the algorithms for higher precision data types such as *double* can also be easily constructed.

The considered algorithms can be effectively used on platforms that support floating-point arithmetic but without native hardware implementation of inverse square root or square root functions. In many cases, these algorithms will be also faster than calling *sqrtf* function from the *cmath* C++ library. A vivid example of such a device is ESP-WROOM-32 microcontroller [15]. In addition, such algorithms may be implemented on modern FPGAs [16] with single precision floating-point blocks for addition, multiplication, and fused multiply-add operation.

REFERENCES

1. PARHAMI B.: Computer Arithmetic: Algorithms and Hardware Designs, 2nd edition. Oxford Univ. Press, New York 2010.
2. BEEBE N.H.F.: The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library. Springer, 2017.
3. LEMAITRE F., COUTURIER B., LACASSAGNE L.: Cholesky Factorization on SIMD multi-core architectures. Journal of Systems Architecture, **79**(2017), 1-15.
4. LIN J., XU Z.G., NUKADA A., MARUYAMA N., MATSUOKA S.: Optimizations of Two Compute-bound Scientific Kernels on the SW26010 Many-core Processor. 46th Inter. Conf. on Parallel Processing (ICPP), IEEE 2017, 432-441.
5. EBERLY D.H.: GPGPU Programming for Games and Science. CRC Press 2015.
6. CARLILE B., DELAMARTER G., KINNEY P., MARTI A., WHITNEY B.: Improving Deep Learning By Inverse Square Root Linear Units (ISRLUS). arXiv preprint arXiv:1710.09967(2017).
7. id software, quake3-1.32b/code/game/q_math.c, Quake III Arena, 1999.
8. MOROZ L., WALCZYK C.J., HRYNCHYSHYN A., HOLIMATH V., CIESLINSKI J.L.: Fast calculation of inverse square root with the use of magic constant – analytical approach. Applied Mathematics and Computation, **316**(2018), 245-255.
9. WALCZYK C.J., MOROZ L.V., CIESLINSKI J.L.: Improving the accuracy of the fast inverse square root algorithm. arXiv preprint arXiv:1802.06302(2018).
10. HASNAT A., BHATTACHARYYA, DEY A., HALDER S., BHATTACHARYEE D.: A Fast FPGA Based Architecture for Computation of Square Root and Inverse Square Root. Devices for Integrated Circuit (DevIC), Kalyani 2017, 383-387.
11. HSU C.J., CHEN J.L., CHEN L.G.: An Effcient Hardware Implementation of HON4D Feature Extraction for Real-time Action Recognition. Inter. Symp. on Consumer Electronics (ISCE), IEEE 2015, 1-2.
12. LI Z., CHEN Y., ZENG X.: OFDM Synchronization implementation based on Chisel platform for 5G research. 11th Inter. Conf. on ASIC (ASICON), IEEE 2015, 1-4.
13. MARTIN P.: Eight Rooty Pieces. Overload Journal, **135**(2016), 8-12.
14. LOMONT C.: Fast inverse square root. Purdue University, Tech. Rep., 2003: *http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf,* 26.05.2018.

15. Espressif Systems, ESP32-WROOM-32 (ESP-WROOM-32) datasheet. Version 2.4, 2018.
16. Intel Corporation, Intel® Cyclone® 10 GX device overview. C10GX51001, 2018.