computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

SIMPLE EFFECTIVE FAST INVERSE SQUARE ROOT ALGORITHM WITH TWO MAGIC CONSTANTS

Oleh Horyachyy, Leonid Moroz, Viktor Otenko

Lviv Polytechnic National University, 12 Bandera str., 79013 Lviv, Ukraine,
oleh.y.horiachyi@lpnu.ua, moroz_lv@lp.edu.ua, viotenko@gmail.com, http://lpnu.ua

Paper history:
Received 09 January 2019
Received in revised form 04 June 2019
Accepted 25 October 2019
Available online 31 December 2019

Keywords:
inverse square root;
FISR algorithm;
initial approximation;
magic constant;
IEEE 754 standard;
floating-point arithmetic;
FMA function;
maximum relative error;
Newton-Raphson;
Householder.

Abstract: The purpose of this paper is to introduce a modification of Fast Inverse Square Root (FISR) approximation algorithm with reduced relative errors. The original algorithm uses a magic constant trick with input floating-point number to obtain a clever initial approximation and then utilizes the classical iterative Newton-Raphson formula. It was first used in the computer game Quake III Arena, causing widespread discussion among scientists and programmers, and now it can be frequently found in many scientific applications, although it has some drawbacks. The proposed algorithm has such parameters of the modified inverse square root algorithm that minimize the relative error and includes two magic constants in order to avoid one floating-point multiplication. In addition, we use the fused multiply-add function and iterative methods of higher order in the second iteration to improve the accuracy. Such algorithms do not require storage of large tables for initial approximation and can be effectively used on field-programmable gate arrays (FPGAs) and other platforms without hardware support for this function.

1. INTRODUCTION

When solving many important problems in the field of digital signal processing, computer 3D graphics, scientific computing, etc., it is necessary to use floating-point arithmetic [1-7] as the most accurate and common way of representing real numbers in computers. In particular, a common practical task that arises when working with floating-point numbers is the calculation of elementary functions, including the inverse square root:

$$y = \frac{1}{\sqrt{x}}. \qquad (1)$$

The algorithms implementing the inverse square root calculation are widely described in scientific literature [1, 6, 8-14]. Most of them are iterative and require the formation of an initial approximation. Moreover, the more precise the initial approximation, the fewer iterations are needed to calculate a function with the required accuracy. As a rule, the initial approximation is formed using a lookup table (LUT – lookup table) [1-2]. However, a group of iterative algorithms is also known that do not use LUT [6, 8-15]. Here, the initial approximation is formed using integer arithmetic in the form of so-called magic constant, and upon transition from integer arithmetic to floating-point arithmetic, the resulting approximation is fed into an iterative process using corresponding Newton-Raphson formulae.

In this article, we consider algorithms for calculating the inverse square root using magic constants, which offers reduced relative errors and applies to floating-point numbers represented in the IEEE 754 standard format. In addition, an important advantage of the proposed algorithms is a reduction (by one) in the number of multiplication operations.

The best-known version of this algorithm called Fast Inverse Square Root (FISR) [8, 15-18], which was used in the computer game Quake III Arena [7], is given below:

```
1. float InvSqrt1 (float x){
```

```
2.  float half = 0.5f * x;
3.  int i = *(int*)&x;
4.  i = 0x5F3759DF - (i>>1);
5.  x = *(float*)&i;
6.  x = x*(1.5f - half*x*x);
7.  x = x*(1.5f - half*x*x);
8.  return x;
9. }
```

This InvSqrt1 code, written in the C/C++ language, implements a fast algorithm for inverse square root calculation. In line 3, we represent the bits of the input variable x (*float*) as the variable i (*int*). Lines 4 and 5 determine the initial approximation y_0 of the inverse square root. Note that the InvSqrt1 algorithm reuses the variable x for this purpose. It then becomes a part of the iterative process. Here in line 4, R = 0x5f3759df is a "magic constant". In line 5, we represent the bits of the variable i (*int*) back into the variable x (*float*). Note that henceforth in the code, the variable x acts as a new approximation to y. Lines 6 and 7 contain two classic successive Newton-Raphson iterations to refine the results. If the maximum relative error of calculations after the second iteration is denoted by $\delta_{2\max}$, then the accuracy of this algorithm is only

$$\left|\delta_{2\max}\right| = 4.73\cdot 10^{-6}, \text{ or } -\log_2\left(\left|\delta_{2\max}\right|\right) = 17.69 \qquad (2)$$

correct bits. These results are obtained by practical calculations on the Intel Core i7-7700HQ platform using the method that will be discussed in more detail in Section 3.

Compared to this algorithm, the following algorithm with a magic constant from [12] has better accuracy, namely 20.37 correct bits:

```
1. float InvSqrt2 (float x){
2.  float half = 0.5f * x;
3.  int i = *(int*)&x;
4.  i = 0x5F376908 - (i>>1);
5.  x = *(float*)&i;
6.  x = x*(1.5008789f - half*x*x);
7.  x = x*(1.5000006f - half*x*x);
8.  return x;
9. }
```

It has different magic constant and two modified Newton-Raphson iterations. The maximum relative error of the algorithm is approximately

$$\left|\delta_{2\max}\right| = 7.37\cdot 10^{-7}, \qquad (3)$$

which is slightly worse than declared by the authors, even when using the fused multiply-add (FMA) function to increase the accuracy of calculations.

Lemaitre et al. [6] suggest using for type *float* Householder's method of order 4 with fused operations. For initial approximation they consider magic constant R = 0x5f375a86 [8]. Thus, their proposed algorithm has the form:

```
1. float InvSqrt3 (float x){
2.  int i = *(int*)&x;
3.  i = 0x5F375A86 - (i>>1);
4.  float y = *(float*)&i;
5.  float a = x*y*y;
6.  float t = fmaf(0.2734375f, a, -1.40625f);
7.  t = fmaf(a, t, 2.953125f);
8.  t = fmaf(a, t, -3.28125f);
9.  y = y*fmaf(a, t, 2.4609375f);
10. return y;
11. }
```

It has maximum relative error

$$\left|\delta_{1\max}\right| = 6.58\cdot 10^{-7}, \qquad (4)$$

or 20.54 (out of 24 possible) bits of accuracy.

Consequently, there is still a need to develop algorithms of higher accuracy, also for IEEE 754 numbers of higher precision, such as double and quadruple, without forgetting about the speed of computations.

The remainder of this paper is organized as follows. Section 2 further gives a theoretical description of the aforementioned algorithms with magic constant. In Sections 3 and 4, we elucidate the proposed algorithms and give their implementations in C++. Section 5 contains simulation results. Conclusions are given in Section 6.

2. ANALYTICAL DESCRIPTION OF KNOWN ALGORITHMS

In this section, we briefly present the main results of [11-12] to explain how InvSqrt1, InvSqrt2, and InvSqrt3 work. Assume that we have a positive floating-point number

$$x = (1 + m_x)\cdot 2^{E_x}, \qquad (5)$$

where

$$m_x \in [0,1) \qquad (6)$$

$$E_x = \lfloor \log_2(x) \rfloor = floor(\log_2(x)). \qquad (7)$$

In the IEEE 754 standard, the floating-point number x is encoded by 32, 64, or 128 bits for three basic binary floating-point formats. In this paper, we will mostly consider single precision (*float* type) numbers (as in the above algorithms), which have 32 bits, but all considerations can also be generalized for other data types, as it will be shown later in Section 4. The first bit corresponds to a sign (sign field). In our case, this bit is equal to zero. The next eight bits correspond to a biased exponent E_x (exponent field), and the last 23 bits encode a fractional part of the mantissa m_x (mantissa field). The integer encoded by these 32 bits, I_x, is given by

$$I_x = (bias + E_x + m_x)N_m, \tag{8}$$

where $N_m = 2^{23}$ and $bias = 127$ for *float*. Line 4 of the code InvSqrt1 or InvSqrt2 and line 3 of InvSqrt3 can be written as:

$$I_{y_0} = R - \lfloor I_x / 2 \rfloor. \tag{9}$$

The result of subtracting the integer $\lfloor I_x/2 \rfloor$ from the magic constant R is the integer number I_{y_0}, which is being represented as a *float* (lines 5 and 4 respectively) and gives the initial (zeroth) piecewise linear approximation y_0 of the function $1/\sqrt{x}$. Further on, this approximation is refined (lines 6-7 and 5-9). The InvSqrt1 and InvSqrt2 algorithms use two classical (in the case of the InvSqrt1 algorithm)

$$y_1 = 1/2 * y_0(3 - x * y_0 * y_0) \tag{10}$$
$$y_2 = 1/2 * y_1(3 - x * y_1 * y_1) \tag{11}$$

or modified (for the InvSqrt2 algorithm)

$$y_1 = 1/2 * y_0(k_2 - x * y_0 * y_0), \tag{12}$$
$$y_2 = 1/2 * y_1(k_4 - x * y_1 * y_1). \tag{13}$$

Newton-Raphson iterations, which provide quadratic convergence of the iterative process. In the last formulas, k_2 and k_4 are constants (see InvSqrt2). Householder's formula in the InvSqrt3 algorithm

$$a = x * y_0 * y_0 \tag{14}$$
$$y_1 = y_0(a(a(a(a * t_1 + t_2) + t_3) + t_4) + t_5), \tag{15}$$

has a rate of convergence equal to 5. Here in (15), the values of the constants are as follows:

$$t_1 = \frac{35}{128}, t_2 = -\frac{45}{32}, t_3 = \frac{189}{64}, \\ t_4 = -\frac{105}{32}, t_5 = \frac{315}{128}. \tag{16}$$

As proven in [11-12], in order to know the behaviour of the relative error for y_0 in the whole range of normalized floating-point numbers, it suffices to consider the range $x \in [1,4)$. In this range, a piecewise linear analytical approximation y_0 of the function consists of three separate parts:

$$y_{01} = -\frac{1}{4}x + \frac{3}{4} + \frac{1}{8}t, \quad x \in [1,2) \tag{17}$$
$$y_{02} = -\frac{1}{8}x + \frac{1}{2} + \frac{1}{8}t, \quad x \in [2,t) \tag{18}$$
$$y_{03} = -\frac{1}{16}x + \frac{1}{2} + \frac{1}{16}t, \quad x \in [t,4), \tag{19}$$

where

$$t = 2 + 4m_R + 2/N_m \tag{20}$$

and the fractional part of the mantissa of the magic constant R is determined by the formula

$$m_R = R/N_m - \lfloor R/N_m \rfloor. \tag{21}$$

In this case, the maximum relative error of such piecewise linear approximations does not exceed the value $1/(2N_m)$. Moreover, if we denote the magic constant R by

$$R = Q \cdot N_m + m_R \cdot N_m, \tag{22}$$

where

$$Q = \lfloor R/N_m \rfloor, \tag{23}$$

then under condition $m_R < 1/2$, the equality $Q = 190$ will be satisfied for Eq. (17)-(19). This condition is true for the magic constants of all considered algorithms. However, in general, m_R can take any values from the range $m_R \in [0,1)$. We will investigate this case in the next section.

3. DESCRIPTION OF THE PROPOSED ALGORITHM

Consider the case when $Q = 190$ and $1/2 \le m_R < 1$. Then, according to the theory, the Eq. (17)-(19) will have the form (24)-(26).

$$y_{01} = -\frac{1}{2}x + 1 + \frac{1}{2}t, \quad x \in [1, t) \tag{24}$$

$$y_{02} = -\frac{1}{4}x + 1 + \frac{1}{4}t, \quad x \in [t, 2) \tag{25}$$

$$y_{03} = -\frac{1}{8}x + \frac{3}{4} + \frac{1}{4}t, \quad x \in [2, 4). \tag{26}$$

Here

$$t = 2m_R + 1/N_m. \tag{27}$$

In order to minimize the relative error, we must set out the requirement to align all the maxima of the relative error for the first iteration in the y_{01} and y_{03} segments of the initial approximation. For this, the first iteration will be performed according to the formula

$$y_1 = k_1 y_0 (k_2 - x y_0 y_0). \tag{28}$$

Note that this iteration requires four multiplication operations. Then, taking into account formulas (24)-(26), we write (28) in the form of three corresponding iterative equations

$$y_{1i} = k_1 y_{0i} (k_2 - x y_{0i} y_{0i}), \quad i = \overline{1,3}. \tag{29}$$

Let us try to reduce the number of multiplications in the first iteration to three. Such algorithms with a reduced number of floating-point multiplications will be especially useful when implementing on FPGA [13-14]. To do this, we set the condition $k_1 = 1/4$. Then the multiplication by this value in software or hardware implementation can be replaced by the integer subtraction. The basic idea is to subtract two from the exponent of a number represented in the IEEE 754 format.

Next, we need to find the best values of the coefficient k_2 and parameter t, which will determine the magic constant R. To do this, we write the relative errors of the first iteration based on the initial approximations y_{01}, y_{02}, and y_{03}:

$$\delta_{1i} = 0.25 y_{0i} (k_2 - x y_{0i} y_{0i}) \cdot sqrt(x) - 1, \quad i = \overline{1,3}. \tag{30}$$

Here we use the formula

$$\delta_1 = y_1 \cdot sqrt(x) - 1 \tag{31}$$

to determine the relative error of the approximation y_1 for the inverse square root function. Note also that to calculate the maximum relative errors after the first iteration, we use the formula

$$|\delta_{1\max}| = \max_{x \in [1,4)} |y_1 \cdot sqrt(x) - 1|. \tag{32}$$

Denote the corresponding upper and lower bounds of the relative errors by $\delta^+_{1\max}$ and $\delta^-_{1\max}$.

Errors (30) have local maxima, in particular, at points

$$x_{11\max} = (2 + t)/3, \tag{33}$$

$$x_{13\max} = (6 + 2t)/3. \tag{34}$$

For the second component y_{02}, one should also take into account the point t, where the relative error δ_{12} (as well as δ_{11}) has the negative maximum δ_{12t} (see Fig. 1). Now substituting (33), (34), and (27) in (30), we obtain expressions for $\delta_{11\max}$, $\delta_{13\max}$, and δ_{12t}. Based on them, we construct a system of two equations:

$$\begin{cases} \delta_{11\max} + \delta_{12t} = 0 \\ \delta_{13\max} + \delta_{11\max} = 0 \end{cases}, \tag{35}$$

which will ensure the fulfillment of the above condition and the requirement to minimize the relative error. The solution of this system will be the following expressions for t and k_2:

$$t = 1.4955357391311740246219261 22, \tag{36}$$

$$k_2 = 4.7642670118683660973601945 44. \tag{37}$$

Given (27), we can determine that for numbers of type *float*

$$m_R = 0.7477678099609422369203380 61. \tag{38}$$

Experimental results have shown that in practice for single-precision floating-point numbers (*float*) the best values of these parameters are k_2= 4.764266968f and the corresponding magic constant R = 0x5F5FB6D3. Similarly, for double-precision numbers: k_2= 4.7642670066528519 and R = 0x5FEBF6DB526DE7D9.

In the next section, we will look at the implementation of the proposed algorithms in C++

for type *float* and consider the peculiarities for numbers of higher precision.

4. IMPLEMENTATION OF THE PROPOSED ALGORITHMS

4.1 SINGLE PRECISION

The final C++ code of the proposed InvSqrt41 algorithm with a single modified Newton-Raphson iteration for type *float* has the form:

```
1. float InvSqrt41 (float x){
2.     int i = *(int*)&x;
3.     i = i>>1;
4.     int ii = 0x5E5FB6D3 - i;
5.     i = 0x5F5FB6D3 - i;
6.     float y = *(float*)&i;
7.     float yy = *(float*)&ii;
8.     y = yy*(4.764266968f - x*y*y);
9.     return y;
10. }
```

The maximum values of the relative error in this case are:

$$\begin{aligned} \delta^{+}_{1\max} &= 6.502572\cdot 10^{-4}, \\ \delta^{-}_{1\max} &= -6.502245\cdot 10^{-4}. \end{aligned} \tag{39}$$

These error values correspond to 10.59 exact bits of the result. Fig. 1 below shows a graph of the relative errors of the InvSqrt41 algorithm.

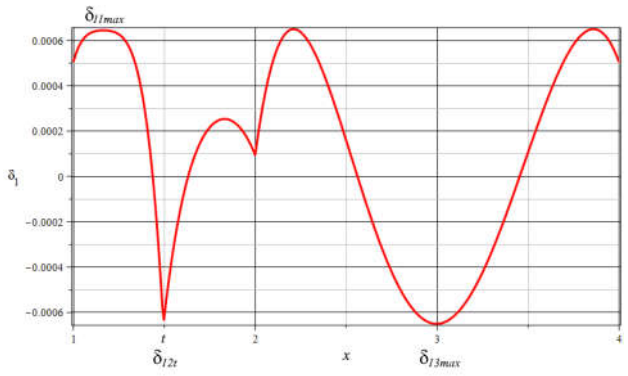

Figure 1 – The graph of the relative errors of the InvSqrt41 algorithm on the interval $x \in [1,4)$

Thus, the above algorithm contains only three floating-point multiplication operations, and the maximum error is reduced compared to the InvSqrt2 algorithm (after the first iteration) by 26%.

If we add the second Newton-Raphson iteration using the FMA function, then we can achieve much smaller maximum errors. The accuracy of the algorithm can be also improved using the method of practical optimization of the algorithm constants, as we did for InvSqrt41. See the InvSqrt42 algorithm. The FMA function is used here to reduce the rounding error of the corresponding floating-point operations, which become noticeable at the last iteration of the algorithm (lines 9-11).

```
1. float InvSqrt42 (float x){
2.     int i = *(int*)&x;
3.     i = i>>1;
4.     int ii = 0x5E5FB432 - i;
5.     i = 0x5F5FB432 - i;
6.     float y = *(float*)&i;
7.     float yy = *(float*)&ii;
8.     y = yy*(4.76405191f - x*y*y);
9.     float c = x*y;
10.    c = fmaf(y, c, -1.0000006f);
11.    y = fmaf(-0.500097573f*y, c, y);
12.    return y;
13. }
```

The maximum relative errors of this algorithm are:

$$\begin{aligned} \delta^{+}_{2\max} &= 3.756709\cdot 10^{-7}, \\ \delta^{-}_{2\max} &= -3.973408\cdot 10^{-7}, \end{aligned} \tag{40}$$

which correspond to 21.26 correct bits of the result. This algorithm has seven floating-point multiplication operations, including FMAs. Compared to the InvSqrt2 and InvSqrt3 algorithms, which also have seven multiplications, the errors of the proposed algorithm are almost 46% and 40% less, respectively.

The following modification of the algorithm is a compromise between accuracy and speed.

```
1. float InvSqrt43 (float x){
2.     int i = *(int*)&x;
3.     int ix = i - 0x80800000;
4.     i = i>>1;
5.     int ii = 0x5E5FB3E2 - i;
6.     i = 0x5F5FB3E2 - i;
7.     float y = *(float*)&i;
8.     float yy = *(float*)&ii;
9.     y = yy*(4.76424932f - x*y*y);
10.    float mhalf = *(float*)&ix;
11.    float t = fmaf(mhalf, y*y, 0.500000298f);
12.    y = fmaf(y, t, y);
13.    return y;
14. }
```

This algorithm has six multiplication operations, performing one multiplication of the number x by $-1/2$ in integer representation (line 3). The accuracy of the InvSqrt43 algorithm is 21.21 correct bits.

The accuracy could be further improved by using the Householder's formula of order 2 in the second iteration [6]:

$$a = x * y_1 * y_1 \tag{41}$$

$$y_2 = y_1(a(a * t_1 + t_2) + t_3), \tag{42}$$

where

$$t_1 = \frac{3}{8}, t_2 = -\frac{5}{4}, t_3 = \frac{15}{8}. \tag{43}$$

As a result, we get the following modified algorithm with optimized constants:

```
1. float InvSqrt44 (float x){
2.     int i = *(int*)&x;
3.     i = i>>1;
4.     int ii = 0x5E5FB414 - i;
5.     i = 0x5F5FB414 - i;
6.     float y = *(float*)&i;
7.     float yy = *(float*)&ii;
8.     y = yy*(4.76410007f - x*y*y);
9.     float c = x*y;
10.    float r = fmaf(y, c, -1.0f);
11.    c = fmaf(0.374000013f, r, -0.5f);
12.    y = fmaf(r*y, c, y);
13.    return y;
14. }
```

This algorithm has eight floating-point multiplications. The maximum values of the relative errors of the algorithm are:

$$\begin{aligned} \delta^{+}_{2\max} &= 8.604127 \cdot 10^{-8}, \\ \delta^{-}_{2\max} &= -8.176169 \cdot 10^{-8}, \end{aligned} \tag{44}$$

or 23.47 correct bits of the result out of 24 possible. Compared to the Householder's method of order 5 [6], which has the same number of multiplications, the error of the InvSqrt44 algorithm is reduced by 6 times.

4.2 DOUBLE PRECISION

The same algorithms can be also implemented in *double*. In this case, in (8) $N_m = 2^{52}$ and $bias = 1023$ for 64-bit IEEE 754 numbers. Also, $Q = 1534$ in the magic constant (22).

For example, an analogue of the InvSqrt41 algorithm for *double* will be:

```
1. double InvSqrt45_d (double x){
2.     long long i = *(long long*) &x;
3.     i = i>>1;
4.     long long ii = 0x5FCBF6DB526DE7D9 - i;
5.     i = 0x5FEBF6DB526DE7D9 - i;
6.     double y = *(double*)&i;
7.     double yy = *(double*)&ii;
8.     y = yy*(4.7642670066528519 - x*y*y);
9.     return y;
10. }
```

This algorithm has almost the same accuracy, namely

$$\begin{aligned} \delta^{+}_{1\max} &= 6.501427 \cdot 10^{-4}, \\ \delta^{-}_{1\max} &= -6.501427 \cdot 10^{-4}, \end{aligned} \tag{45}$$

but the use of a double-precision data type will give a tangible improvement after two or three iterations. To illustrate this, the double version of InvSqrt43 algorithm will have an accuracy of 21.59 bits, and it will be 30.44 bits for the InvSqrt44 algorithm.

A modification of the InvSqrt43 algorithm with three Newton-Raphson iterations, optimized for speed, will look like:

```
1. double InvSqrt46_d (double x){
2.     long long i = *(long long*) &x;
3.     long long ix = i - 0x8010000000000000;
4.     i = i>>1;
5.     long long ii = 0x5FCBF6D99EF4C0F4 - i;
6.     i = 0x5FEBF6D99EF4C0F4 - i;
7.     double y = *(double*)&i;
8.     double yy = *(double*)&ii;
9.     y = yy*(4.7642669737958503 - x*y*y);
10.    double mhalf = *(double*)&ix;
11.    double t = fma(mhalf, y*y,
                  0.50000031699508796);
12.    y = fma(y, t, y);
13.    t = fma(mhalf, y*y, 0.50000000000007538);
14.    y = fma(y, t, y);
15.    return y;
16. }
```

This algorithm has nine multiplications and provides 43.59 correct bits. For comparison, the corresponding FISR [6, 8] and Walczyk [12] algorithms have 10 multiplications and an accuracy of 34.88 and 41.85 bits.

To further improve the accuracy of the algorithm in type *double*, you need to perform an additional iteration of the quadratic convergence in the algorithm based on InvSqrt44 or replace the last iteration in the previous algorithm (lines 13-14) with the Householder's formula of order 2 (lines 13-16 of the InvSqrt47_d algorithm). Both algorithms will have similar accuracy for type *double*, but the second option should be faster:

```
1. double InvSqrt47_d (double x){
2.    long long i = *(long long*) &x;
3.    long long ix = i - 0x8010000000000000;
4.    i = i>>1;
5.    long long ii = 0x5FCBF6D9DB9A45CD - i;
6.    i = 0x5FEBF6D9DB9A45CD - i;
7.    double y = *(double*)&i;
8.    double yy = *(double*)&ii;
9.    y = yy*(4.7642670025852993 - x*y*y);
10.   double mhalf = *(double*)&ix;
11.   double t = fma(mhalf, y*y,
                  0.50000031697852854);
12.   y = fma(y, t, y);
13.   double c = x*y;
14.   double r = fma(y, c, -1.0);
15.   c = fma(0.375, r, -0.5);
16.   y = fma(r*y, c, y);
17.   return y;
18. }
```

The maximum relative errors of this algorithm are:

$$\delta^{+}_{3\max} = 1.387779 \cdot 10^{-16}, \quad \delta^{-}_{3\max} = -1.387779 \cdot 10^{-16}. \tag{46}$$

As you can see, the algorithm provides an accuracy of 52.68 bits out of 53 possible for *double* and it has eleven multiplications. Comparing with the algorithm suggested by Lemaitre et al. [6] for double precision numbers, which includes two iterations of the order 3 Householder's method using FMA and contains 12 multiplication operations, the errors of our algorithm are reduced by 3.5 times.

4.3 HIGHER PRECISION

In general, for numbers of any precision, the value of Q in the magic constant R is obtained by the formula

$$Q = bias + \lfloor bias / 2 \rfloor. \tag{47}$$

Then the theoretical value of the basic magic constant R for the proposed method is determined by (22), where the value of the mantissa m_R is equal to (38) and the parameters $bias$ and N_m are determined by the data type that is applied. The second magic constant $R2$ is obtained from R using the following expression:

$$R2 = R - 2 \cdot N_m. \tag{48}$$

The value of the constant k_2 in the first iteration is (37).

Table 1 summarizes the process of determining the theoretical values of the magic constants R and $R2$ for the three basic data types of the IEEE 754 standard (*float*, *double*, and *__float128* respectively).

Table 1. Theoretical values of the magic constants

Constant	IEEE 754 binary floating-point format		
	single (32 bits)	double (64 bits)	quadruple (128 bits)
bias	127	1023	16383
N_m	2^{23}	2^{52}	2^{112}
Q	190	1534	24574
R	5F5FB6DB	5FEBF6DB 610C8A67	5FFEBF6D B610C8A6 75345B2E4 EFA8BA0
$R2$	5E5FB6DB	5FCBF6DB 610C8A67	5FFCBF6D B610C8A6 75345B2E4 EFA8BA0

Please note that in practice, the theoretical values of the magic constants indicated in the table may not be optimal due to the rounding of arithmetic operations.

5. EXPERIMENTAL RESULTS AND DISCUSSION

We have also tested the proposed algorithms on the ESP-WROOM-32 microcontroller [19]. In Table 2 you can find the results of measuring the latency and accuracy of the algorithms for single and double precision numbers.

5.1 SINGLE PRECISION

After analyzing the results for single precision numbers, we can say that the InvSqrt1, InvSqrt2, and InvSqrt43 algorithms have the highest performance, but our algorithm (InvSqrt43) provides the best accuracy among them. InvSqrt44 has the highest accuracy among all the considered algorithms, namely 23.47 bits, and its speed is the same as in the Householder's algorithm of order 5. As you can see, all FISR-based algorithms on this platform are superior in speed to using the *sqrtf* function from the *cmath* library. However, only this mathematical function supports subnormal numbers.

Please note that relative errors may differ slightly on different platforms depending on the compiler and the implementation of arithmetic operations, in particular, the InvSqrt2 algorithm has higher accuracy on ESP-32 than on Intel. The reason is that this microcontroller has fast hardware FMA instructions for type *float* [20], which are

automatically used by the compiler to perform all arithmetic operations. Here we use the following compiler options (for GCC 5.2.0): *-std=gnu++11 -Os -ffp-contract=fast*.

Table 2. Evaluation of the algorithms on ESP-32

Algorithm	Accuracy, bits	Latency, ns
float		
1.0f/sqrtf(x)	23.42	801.5
FISR [7] (InvSqrt1)	17.69	281.2
Walczyk [12] (InvSqrt2)	20.40	281.2
Householder (order 4) [6] (InvSqrt3)	20.54	323.1
Householder (order 5) [6]	20.87	348.3
InvSqrt42	21.26	306.4
InvSqrt43	21.21	281.2
InvSqrt44	23.47	348.3
double		
1.0/sqrt(x)	52.42	6893.5
FISR [6, 8] (3 iter.)	34.88	5373.9
FISR [6, 8] (4 iter.)	51.68	6849.9
Walczyk [12] (3 iter.)	41.85	5471.4
Walczyk [12] (4 iter.)	51.69	6982.3
Householder (order 3) [6] (2 iter.)	50.86	8010.5
Householder (order 3) [6] (2 iter.) (*)	50.71	7548.1
Householder (order 3) [6] (2 iter.) (**)	50.62	7455.6
InvSqrt46_d	43.59	6106.0
InvSqrt46_d (**)	43.59	5258.7
InvSqrt47_d	52.68	7286.9
InvSqrt47_d (*)	52.68	6777.1

5.2 DOUBLE PRECISION

Regarding numbers of type *double* on the microcontroller, first of all, it should be said that ESP-32 does not have the hardware to support them [20], so all the corresponding operations are emulated by software, in particular, multiplication, addition, and FMA. That is why, all the considered algorithms are inefficient in terms of speed, and using the FMA function on this platform is especially costly. Therefore, to increase the performance of the approximation algorithms, the sign "*" in Table 2 indicates the implementation of the algorithms using only one FMA function, which is applied at the last iteration. For example, for the InvSqrt47_d (*) algorithm, this is FMA in line 14. Accordingly, "**" indicates the absence of FMA. It should be said that this step may cause a decrease in accuracy.

Based on Table **2** ??? , it can be said that our InvSqrt47_d (*) algorithm provides both the best accuracy and latency when compared to the *sqrt* function and corresponding FISR and Walczyk algorithms. Regarding the InvSqrt46_d (**) algorithm, it is also more accurate than the Walczyk algorithm for three iterations and has better performance on ESP-32.

Thus, based on the obtained results, we can conclude that for the effective implementation of the proposed algorithms with two magic constants, it is important to have fast hardware multiplication and FMA operations of the corresponding floating-point data type.

6. CONCLUSIONS

We have proposed the FISR-based inverse square root calculation algorithms that contain one less floating-point multiplication by introducing an additional magic constant. The advantage of the considered algorithms is that they do not use lookup tables to form an initial approximation and offer increased calculation accuracy by minimizing the maximum relative error and using FMA.

The practical value of the results lies in the fact that the proposed algorithms can easily replace the basic FISR method in many applications, which increases the accuracy of calculations and reduces the latency on some platforms. In particular, the FISR algorithm with the magic constant $R = 0x5f3759df$ is widely used to implement the approximation of the reciprocal square root function in computer vision and object detection tasks [14, 21-24] when it is necessary to improve the performance of the software and/or hardware application, especially in the case of real-time computing.

These algorithms can be effectively used in microcontrollers that support floating-point calculations, but do not have an FPU (floating-point unit) to calculate the inverse square root, such as, for example, ESP-WROOM-32 [19-20]. They can also be effective in hardware implementation on FPGA platforms that have pipelining and cheap integer operations. An example of such a platform is Intel Cyclone, which has fast single-precision floating-point blocks [25].

It should also be noted that only normalized floating-point numbers of single and double precision were discussed in the article. As was shown, these algorithms can be easily generalized to other data formats of the IEEE 754 standard, namely for quadruple and octuple precision.

7. REFERENCES

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, second ed., Oxford Univ. Press, New York, 2010, 641 p.

[2] N.H.F. Beebe, *The Mathematical-Function Computation Handbook: Programming Using*

the MathCW Portable Software Library, Springer, 2017, 1115 p.

[3] M.D. Ercegovac, T. Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations*, Boston: Kluwer Academic Publishers, 1994, 230 p.

[4] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle, S. Torres, *Handbook of Floating-Point Arithmetic*, Springer, 2010, 572 p.

[5] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefevre, G. Melquiond, N. Revol, S. Torres, "Software implementation of floating-point arithmetic," *in: Handbook of Floating-Point Arithmetic*, Birkhäuser, Cham, 2018, pp. 321-374.

[6] F. Lemaitre, B. Couturier, L. Lacassagne, "Cholesky factorization on SIMD multi-core architectures," *Journal of Systems Architecture*, vol. 79, pp. 1-15, 2017.

[7] Id Software, *Quake III Arena*, *quake3-1.32b code*, [Online]. Available: https://github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c

[8] C. Lomont, *Fast Inverse Square Root*, Purdue University, Tech. Rep., 2003, [Online]. Available at: http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf

[9] P. Martin, "Eight rooty pieces," *Overload Journal*, vol. 24, issue 135, pp. 8-12, 2016.

[10] D.H. Eberly, *GPGPU Programming for Games and Science*, CRC Press, 2014, 469 p.

[11] L. Moroz, C.J. Walczyk, A. Hrynchyshyn, V. Holimath, J.L. Cieśliński, "Fast calculation of inverse square root with the use of magic constant – analytical approach," *Applied Mathematics and Computation*, vol. 316, issue C, pp. 245-255, 2018.

[12] C.J. Walczyk, L.V. Moroz, J.L. Cieśliński, "Improving the accuracy of the fast inverse square root algorithm," *ArXiv Preprint*, arXiv:1802.06302, 2018.

[13] A. Hasnat, T. Bhattacharyya, A. Dey, S. Halder and D. Bhattacharjee, "A fast FPGA based architecture for computation of square root and inverse square root," *Proceedings of the International Conference on Devices for Integrated Circuit (DevIC)*, Kalyani, India, 2017, pp. 383-387.

[14] C.J. Hsu, J.L. Chen and L.G. Chen, "An efficient hardware implementation of HON4D feature extraction for real-time action recognition," *Proceedings of the 2015 IEEE International Symposium on Consumer Electronics (ISCE)*, 2015, pp. 1-2.

[15] Z. Li, Y. Chen and X. Zeng, "OFDM synchronization implementation based on Chisel platform for 5G research," *Proceedings of the 2015 IEEE 11th International Conference on ASIC (ASICON)*, 2015, pp. 1-4.

[16] S. Zafar and R. Adapa, "Hardware architecture design and mapping of fast inverse square root's algorithm," *Proceedings of the International Conference on Advances in Electrical Engineering (ICAEE)*, 2014, pp. 1-4.

[17] C.A. Navarro, M. Vernier, N. Hitschfeld, B. Bustos, "Competitiveness of a non-linear block-space GPU thread map for simplex domains," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, issue 12, pp. 2728-2741, 2018.

[18] A. Gazneli et al. *Adaptive Nonlinear System Control using Robust and Low-complexity Coefficient Estimation*, U.S. Patent No 15/648, 205, 2018.

[19] Espressif Systems, *ESP32-WROOM-32 (ESP-WROOM-32) datasheet. Version 2.4*, 2018.

[20] Tensilica Inc., *Xtensa Instruction Set Architecture (ISA)*, Reference Manual, 2018.

[21] S. Xiao, T. Isshiki, D. Li, H. Kunieda, "HOG-based object detection processor design using ASIP methodology," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 100, issue 12, pp. 2972-2984, 2017.

[22] M. Ramirez-Martinez, F. Sanchez-Fernandez, P. Brunet, S. M. Senouci and El-Bay Bourennane, "Dynamic management of a partial reconfigurable hardware architecture for pedestrian detection in regions of interest," *Proceedings of the 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1-7.

[23] J. Lv, F. Wang and Z. Ma, "Peach fruit recognition method under natural environment," *Proceedings of the Eighth International Conference on Digital Image Processing (ICDIP'2016)*, Chengu, China, August 29, 2016, vol. 10033, paper 1003317.

[24] B.K. Anirudh, V. Venkatraman, A. R. Kumar and S. D. Sumam, "Accelerating real-time computer vision applications using HW/SW co-design," *Proceedings of the 2017 International Conference on Computer, Communications and Electronics (Comptelix)*, Jaipur, India, July 1-2, 2017, pp. 458-463.

[25] Intel Corporation, *Intel® Cyclone® 10 GX device overview. C10GX51001*, 2018.

Oleh Horyachyy *received his MSc degrees in Applied Computer Science from Ivan Franko National University of Lviv in 2013 and in Information and Communication Systems Security from Lviv Polytechnic National University in 2017. Currently, he is a PhD student at the Institute of Computer Technologies, Automation, and Metrology at Lviv Polytechnic National University, Ukraine and an engineer at the Department of Information Technologies Security. His research interests include iterative methods, secure multiparty computations, biometric identification, security of communication technologies, cryptography, and public key infrastructure.*

Leonid Moroz *received his MSc and PhD degrees from Lviv Polytechnic National University, Ukraine in 1978 and 1985, respectively. He defen-ded his DSc degree in Computer Systems and Components in 2013. Currently, he works as a Professor at the Department of Information Technologies Security of the Institute of Computer Technologies, Automation, and Metrology at Lviv Polytechnic National University. His research interests include computer arithmetic, numerical methods, and digital signal processing.*

Viktor Otenko *graduated from Lviv Polytechnic National University, Ukraine and received his MSc degree in Automation and Telemecha-nics in 1979. In 1986, he was awarded the title "Best young inventor of Ukraine". He received a PhD degree in Elements and Appliances of Computer Technology and Control Systems in 1998 from the same university. He has been working at the Department of Information Security of the Institute of Computer Technologies, Automation, and Metrology at Lviv Polytechnic National University as an Assoc. Professor since 2006. His research interests are software engineering, program security, and pulse-numerical functional transformation.*