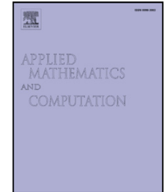
Applied Mathematics and Computation 316 (2018) 245–255

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc



# Fast calculation of inverse square root with the use of magic constant – analytical approach

Leonid V. Moroz [a], Cezary J. Walczyk [b], Andriy Hrynchyshyn [a], Vijay Holimath [c], Jan L. Cieśliński [b,*]

[a] Department of Security Information and Technology, Lviv Polytechnic National University, st. Kn. Romana 1/3, Lviv 79000, Ukraine
[b] Uniwersytet w Białymstoku, Wydział Fizyki, ul. Ciołkowskiego 1L, Białystok 15–245, Poland
[c] VividSparks IT Solutions, Hubli 580031, No. 38, BSK Layout, India

A R T I C L E I N F O

*Keywords:*
Floating-point arithmetic
Inverse square root
Magic constant
Newton–Raphson method

A B S T R A C T

We present a mathematical analysis of transformations used in fast calculation of inverse square root for single-precision floating-point numbers. Optimal values of the so called magic constants are derived in a systematic way, minimizing either relative or absolute errors. We show that the value of the magic constant can depend on the number of Newton–Raphson iterations. We present results for one and two iterations.

© 2017 Elsevier Inc. All rights reserved.

## 1. Introduction

In order to understand the behaviour of equations, functions or models, one has to simulate, compile or execute the functions or equations. They may involve not only the basic operators such as: addition, subtraction and multiplication but also the advanced operators such as division, square root, inverse square root and trigonometric functions. The response time of a processor is a critical factor. The faster is the processing speed of the underlying hardware, the quicker one can simulate or execute these models, equations or functions. In contrast to basic arithmetic operators the advanced operators are relatively complicated to design, slower and take up more hardware [1,2].

Floating-point inverse square root is widely used in several areas such as image and digital signal processing, communications and design of scientific engines [1–4]. Many algorithms can be used to approximate the inverse square root function [1,3,5]. All of these algorithms require an initial seed to approximate function.

If the initial seed is accurate, then iteration required for this function is less time-consuming, i.e., the function requires less cycles. In most of the cases, the initial seed is obtained from Look-Up Table (LUT) and the LUT consume significant silicon area of a chip. In this paper we obtain the initial seed using the so called magic constant [6–10] which does not require LUT. Then we approximate inverse square root function using the Newton–Raphson method once and twice. In both cases we derive analytically the best value of the magic constant solving exactly (up to round-off errors) the non-trivial problem of choosing the best starting values for Newton–Raphson iterations [11].

We present the first mathematically rigorous description of the fast algorithm for computing inverse square root for single-precision IEEE Standard 754 floating-point numbers (type **float**).

* Corresponding author.
*E-mail addresses:* moroz_lv@polynet.lviv.ua (L.V. Moroz), walcez@gmail.com (C.J. Walczyk), hrynchyshyn.a@gmail.com (A. Hrynchyshyn), vijay.holimath@vivid-sparks.com (V. Holimath), j.cieslinski@uwb.edu.pl (J.L. Cieśliński).

http://dx.doi.org/10.1016/j.amc.2017.08.025
0096-3003/© 2017 Elsevier Inc. All rights reserved.

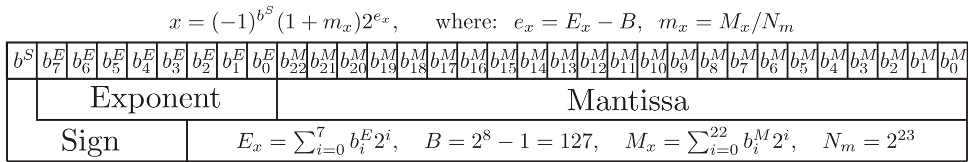



**Fig. 1.** The layout of a 32-bit floating-point number.

```
1. float InvSqrt(float x){
2. float halfnumber = 0.5f * x;
3. int i = *(int*) &x;
4. i = R -(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 code, written in **C**, will be referred to as function *InvSqrt*. It realizes a fast algorithm for calculation of the inverse square root. In line *3* bits of variable x (type **float**) are transferred to variable i (type **int**). In line *4* there is determined an initial value (then subject to the iteration process) of the inverse square root, where *R* is a "magic constant". In line *5* bits of a variable i (type **int**) are transferred to the variable x (type **float**). Lines *6* and *7* contain subsequent iterations of the Newton–Raphson algorithm.

The algorithm *InvSqrt* has numerous applications, in software [8,14,15] and hardware implementation [12,13,16]. The most important among them is 3D computer graphics, where normalization of vectors is ubiquitous. *InvSqrt* is characterized by a high speed, more than 3 times higher than in computing the inverse square root using library functions. This property is discussed in detail in [17]. One has to remember taht the *InvSqrt* works well on Intel processors but on some other architectures it does not perform as well. For example, *InvSqrt* executed on the PowerPC processor has to include loading/storing between floating-point registers and integer registers (load-hit-store penalties) and hardware provided alternatives can be more effective [7].

The errors of the fast inverse square root algorithm depend on the choice of *R*. The original value was $R = $ 0x5F3759DF [10,17]. Its motivation is not quite clear (certainly this is not the best choice), a possible explanation is given in [18]. Lomont found the best values of the magic constant numerically, testing the errors for all the floating-point numbers [11]. In several theoretical papers [11,17–20] (see also the Eberly's monograph [7]) attempts were made to determine analytically the optimal value (i.e. minimizing errors) of the magic constant. Lomont provided a quite detailed analysis showing how the magic number was chosen [11], Eberly showed that in fact there are many magic constants that are quite effective [19]. In our paper we recover analytically the results of Lomont, presenting missing mathematical description of all steps of the fast inverse square root algorithm. In particular, we explain why the optimal value of the magic constant depends on the number of Newton–Raphson iterations.

## 2. Preliminaries

The value of a normal floating-point number can be represented as:

$$x = (-1)^{s_x}(1+m_x)2^{e_x}, \tag{2.1}$$

where $s_x$ is the sign bit ($s_x = 1$ for negative numbers and $s_x = 0$ for positive numbers), $1 + m_x$ is the normalized mantissa (or significand), where $m_x \in [0, 1)$ and, finally, $e_x$ is an integer.

In the case of the *IEEE-754* standard, a floating-point number is encoded by 32 bits (Fig. 1). The first bit corresponds to a sign, the next 8 bits correspond to an exponent $e_x$ and the last 23 bits encodes a mantissa. The fractional part of the mantissa is represented by an integer (without a sign) $M_x$:

$$M_x = N_m m_x, \qquad \text{where:} \quad N_m = 2^{23}, \tag{2.2}$$

and the exponent is represented by a positive value $E_x$ resulting from the shift of $e_x$ by a constant $B$ (biased exponent):

$$E_x = e_x + B, \qquad \text{where:} \quad B = 127 \tag{2.3}$$

and $E_x = 1, \ldots, 254$ (we exclude special cases $E_x = 0$ for 0 and subnormals, and $E_x = 255$ for the infinities and NaN).

In what follows we confine ourselves to positive numbers ($s_x \equiv b_S = 0$). Bits of a floating-point number can be interpreted as an integer given by:

$$I_x = N_m E_x + M_x. \tag{2.4}$$

Download English Version:

https://daneshyari.com/en/article/5775558

Download Persian Version:

https://daneshyari.com/article/5775558

Daneshyari.com