Fast Algorithms for Computing Floating-Point Reciprocal Cube Root Functions

Leonid Moroz[1†], Volodymyr Samotyy[1†] and Cezary Walczyk[2††],
leonid.moroz@pk.edu.pl vsamotyy@pk.edu.pl c.walczyk@uwb.edu.pl
[1] Cracow University of Technology, Poland, [2] University of Bialystok, Poland

Abstract

In this article the problem of computing floating-point reciprocal cube root functions is considered. Our new algorithms for this task decrease the number of arithmetic operations used for computing $1/\sqrt[3]{x}$. A new approach for selection of magic constants is presented in order to minimize the computation time for reciprocal cube roots of arguments with movable decimal point. The underlying theory enables partitioning of the base argument range $x\in[1,8)$ into 3 segments, what in turn increases accuracy of initial function approximation and decreases the number of iterations to one. Three best algorithms were implemented and carefully tested on 32-bit microcontroller with ARM core. Their custom C implementations were favourable compared with the algorithm based on *cbrtf(x)* function taken from C <math.h> library on three different hardware platforms. As a result, the new fast approximation algorithm for the function $1/\sqrt[3]{x}$ was determined that outperforms all other algorithms in terms of computation time and cycle count.

Keywords:
floating-point, cube root, inverse cube root, Newton-Raphson, Householder.

1. Introduction

The demand for fast numerical computing of mathematical functions such as logarithm, square and cube roots, reciprocal and trigonometric functions is fast growing in such application areas as scientific computations, digital signal processing, multimedia, geometry and 3D graphics, thermodynamics, mobile robot navigation, system security, machine learning etc. Both software and hardware algorithms and their implementations are of interest. Many Floating-Point Units have implemented arithmetic operations in hardware making them extremely fast and efficient. Unfortunately, there are only a few implementations of cube root function in hardware, and they are mainly deployed in Field Programmable Gate Arrays (FPGAs). Because of its computation complexity, a cube root is difficult to implement even in FPGAs.

Calculating the cube root was of interest to mathematicians since ancient times. Babylonians, Greeks, Chinese, and Hindus were looking for an efficient method for calculating the cube root as well as the reciprocal cube root [1]. However, the methods they proposed difficult to implement conveniently in a computer architecture, so computer algorithms do it in a completely different way than we do by manually calculating the cube root.

Long Division Method (LDM) is a technique we typically use to calculate manually the cube root of a number [2]. Although the LDM method gives the best possible result for any computed digit when compared to modern numerical methods, so far no one successfully implemented this method numerically. Usually for numerical computation of the cube root, we use approximation or estimation methods based on the Newton-Raphson algorithm, Halley algorithm or their variants including the magic number approach. Approximation algorithms, in each iteration, bring the root closer to the specified precision and require dividing the number by the newly approximated root. However, each iteration performs division operations that take longer than adding or subtracting. Additionally, approximation algorithms have problems in finding the cube root of a non-perfect cube number with preferred precision and barely operate with large numbers [3]. Alternatively, cube roots can be also calculated using digit-by-digit fashion type of algorithms, such as non-restoring algorithm [4], [5], which are more suitable for ASIC and FPGA implementations [6],[7].

Considering all these factors, it is clear that designing an efficient algorithm to perform a cube root and reciprocal cube root calculation is a difficult task, but in many scientific applications it is an essential factor for ensuring high performance.

The quick calculation of cube root and reciprocal cube root using the magic constant was recently discussed in [8]. The proposed approach was based on the method of the magic constant in the single precision format – IEEE Std 754, and approximations of the cube root for an integer through operations that use the optimized iteration method based on Newton-Raphson or Householder algorithms. Even though the presented methods were efficient and fast, we would like to present in this paper new algorithms for reciprocal cube root computation which extend and improve the preceding approach.

The outline of this paper is as follows. In the next section we describe the background information on the approximation method proposed in [8]. Section 3 brings all the necessary theory and the main results of the article. In

Manuscript received June 5, 2023
Manuscript revised June 20, 2023
https://doi.org/10.22937/IJCSNS.2023.23.6.10

section 4, we report the experimental results related to the performance of investigated algorithms on 32-bit STM32F767ZIT6 microcontroller produced by STMicroelectronics. We also assess our algorithms and their implementation parameters for CPU, GPU and IPU platforms, and expose performance evaluation of speed and accuracy for different types of computer architectures. Finally, section 5 briefly sums up several conclusions.

2. Basic Concept

Let us recall after [8] the basic theory and results related to reciprocal cube root approximation. The initial approximation of the function $y = 1/\sqrt[3]{x}$ is denoted by y_0. It is known that the behaviour of the relative error in calculating y_0 in the whole range of normalized numbers with a floating point can be described by its behaviour for $x \in [1,8)$. In this range there are four piecewise linear approximations of the function $y_0 = \{y_{01}, y_{02}, y_{03}, y_{04}\}$:

$$y_{01} = \frac{5}{6} - \frac{1}{6}x + \frac{1}{24}t, \quad x \in [1,2); \tag{1}$$

$$y_{02} = \frac{2}{3} - \frac{1}{12}x + \frac{1}{24}t, \quad x \in [2,4); \tag{2}$$

$$y_{03} = \frac{1}{2} - \frac{1}{24}x + \frac{1}{24}t, \quad x \in [4,t) \tag{3}$$

$$y_{04} = \frac{1}{2} - \frac{1}{48}x + \frac{1}{48}t, \quad x \in [t,8); \tag{4}$$

where:

$t = 4 + 12m_R + 6N_m$;

$m_R = N_m^{-1}R - \lfloor N_m^{-1}R \rfloor$, $m_R > 0$ – is the fractional part of the mantissa of the magic constant *R*;

N_m – is the floating point precision of computations.

The maximum relative error of such analytical approximations does not exceed $1/(2N_m)$ for single precision $N_m = 2^{23}$.

The next step of the approximation increases the accuracy of calculations of the first iteration without increasing the higher order of the iteration patterns - the first order convergence by Newton–Raphson method is followed by the Householder method of higher orders of convergence.

The method presented above was a basis for development algorithms presented in [8]. The best constructed algorithm for approximation of reciprocal cube root function i.e. Algorithm 5 (InvCbrt21) shall be a reference point for evaluation of two new algorithms in section 4.

3. The theoretical background and new algorithms

On the background given in previous section our new methods for computing reciprocal cube root can be developed. Let's now divide the interval $x \in [1, 8)$ into three segments, where the *i*-th segment equals: [1, 2) when *i*=1, [2, 4) when *i* =2, and [4, 8) when *i* =3. In contrary to [8], in each segment we assume only two initial linear approximations of the reciprocal cube root function. The condition must be met that the relative errors of both adjacent initial linear approximations for each segment have the shape shown in Figure 1.

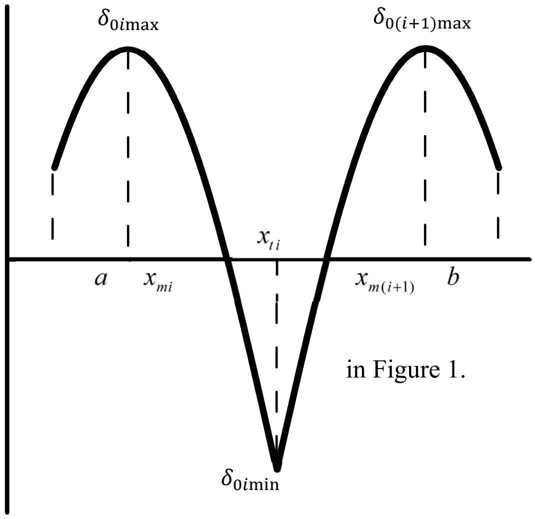

Fig. 1. Expected shape of relative errors for piecewise linear approximation of the function 1/cbrt(x)

Here we have two points of positive maximum values x_{mi} and $x_{m(i+1)}$ and one contact point of diagrams for both errors x_{ti}. This is the point of maximum negative relative error value:

$$\delta_{0i\min} = -\delta_{0i\max} = -\delta_{0(i+1)\max}, \tag{5}$$

which does not depend on the location of the errors on the y axis, where $\delta_{0i} = y_{0i}\sqrt[3]{x} - 1$, *i*=1,2,3.

In this case, the lowest range of relative errors will occur. Now we need to find the values of the magic constants for each segment that will provide this minimum range of relative errors. In the interval x$\in [1,8)$ there are three pairs of (adjacent) errors:

$$\delta_{01} \text{ and } \delta_{02}, x \in [1,2),$$

$$\delta_{02} \text{ and } \delta_{03}, x \in [2,4),$$

$$\delta_{03} \text{ and } \delta_{04}, x \in [4,8).$$

On the basis of experimental studies, it was shown that such error combinations are possible for the following magic constants

R_1=0x543bxxxx, R_2=0x5466xxxx, R_3=0x5491xxxx.

For the obtained values of the magic constants, the equations of the corresponding initial approximations of the reciprocal cube root for each segment will be

$$\text{for } i=1,\ y_{01}=\frac{2}{3}+\frac{t_1}{12}-\frac{x}{6},\ t_1=4+6mR_1+\frac{3}{N_m},$$

$$y_{02}=\frac{7}{12}+\frac{t_1}{24}-\frac{x}{12},$$

$$\text{for } i=2,\ y_{02}=\frac{2}{3}+\frac{t_2}{24}-\frac{x}{12},\ t_2=4+12mR_2+\frac{6}{N_m},$$

$$y_{03}=\frac{7}{12}+\frac{t_2}{48}-\frac{x}{24},$$

$$\text{for } i=3,\ y_{03}=\frac{1}{2}+\frac{t_3}{24}-\frac{x}{24},\ t_3=4+12mR_3+\frac{6}{N_m},$$

$$y_{04}=\frac{1}{2}+\frac{t_3}{49}-\frac{x}{49},$$

For the indicated intervals, contact points x coordinates on the diagram were calculated, where the maxima of negative errors appear,

$$x_{t1}=1+0.5t_1,\ t_1=0.800327,$$

$$x_{t2}=2+0.5t_2,\ t_2=1.600645,\quad x_{t3}=t_3,\ t_3=5.600654.$$

Using condition (5), the corrected values of the magic constants for each interval were found:

R_1=0x543bbd84, R_2=0x5466682f, R_3=0x549112da.

The first iteration for each segment is carried out on the basis of a common iterative scheme of Newton-Raphson method of the first degree

$$y_1 = y_0(k_2 - k_1 x y_0 y_0 y_0).$$

For each range, this scheme looks like $y_{1i} = y_{0i}(k_{2i} - k_{1i} x y_{0i} y_{0i} y_{0i})$,
where for

$$i=1, \quad y_{0i}=\{y_{01}, y_{02}\},$$

$$i=2, \quad y_{0i}=\{y_{02}, y_{03}\},$$

$$i=3, \quad y_{0i}=\{y_{03}, y_{04}\}.$$

The points of the positive values of the maximum relative errors x_{mi} and $x_{m(i+1)}$ in individual segments will be:

$$i=1,\ a=1,\ b=2,\ x_{m1}=1+\frac{t_1}{8},\ x_{m2}=\frac{7}{4}+\frac{t_1}{8},$$

$$i=2,\ a=2,\ b=4,\ x_{m2}=2+\frac{t_2}{8},\ x_{m3}=\frac{7}{2}+\frac{t_2}{8},$$

$$i=3,\ a=4,\ b=8,\ x_{m3}=3+\frac{t_3}{4},\ x_{m4}=6+\frac{t_3}{4}.$$

Based on the maximum points, the values of positive relative errors and the contact points (i.e. points connecting sections), the values of the respective coefficients k_{1i} and k_{2i} were calculated using the serial approximation method described in [8], [9]. For the first segment $x \in [1,2)$: k_{11} = 3.3041991,
k_{21} = 2.3659404; for the second segment $x \in [2,4)$: k_{12} = 1.3112723, k_{22} = 1.877848; for the third segment $x \in [4,8)$: k_{13} = 0.52037869, k_{23} = 1.4904488.

The C code of the resulting algorithm RcpCbrt_7 is presented below.

```
float RcpCbrt_7 (float x) {
      float y, c,k1,k2;
      int i,k,R;
      i = *(int*)&x;
      i=i/3;
      k=i&0x007fffff;
      if (k> 5592405) {            // second segment
        R= 0x5466682F;
        k1=1.3112723f;
        k2=1.877848f;
      } else {
         if (k>2796203) {          // first segment
           R= 0x543bbd84;
           k1=3.3041991f;
           k2=2.3659404f;
         } else {                  // third segment
             R= 0x549112da;
             k1=0.52037869f;
             k2=1.4904488f;
           }
       }
      i = R - i;
      y = *(float*)&i;             // initial approximation
      y = y*(k2-k1*x*y*y*y);       // 1st iteration
      c = 1.0f - x * y * y * y;
```

```
        y = y * (1.0f + 0.33333333f * c);    //            2nd
iteration
        return y;
        }
```

For increasing accuracy in 2nd iteration the value of c can be computed by C function *fmaf*. Initial tests conducted on 32-bit microcontroller for comparison of the algorithm RcpCbrt_7 versus the algorithm InvCbrt21 [8] showed a moderate progress in decreasing the computation time and the number of iterations of the new method (cf. the corresponding rows of Table 1 in Section 4). In order to improve the execution time in the second iteration even further we slightly modified the algorithm RcpCbrt_7. The new version of the algorithm called RcpCbrt_8 with lower number of arithmetic operations is presented below.

```
  float RcpCbrt_8 (float x) {
       float y, xh;
       int i,k;
       i = *(int*)&x;
       i=i/3;
       k=i&0x007fffff;
      if (k>5592405) {
         i= 0x5466682F-i;                             //
       second segment
         y = *(float*)&i;
         xh=0.99708012f*x;
         y=y*(1.4278993f-xh*y*y*y);
         y=y*(1.753483f-xh*y*y*y);
       } else {
            if (k>2796203) {
                 i= 0x543bbd84-i;                     //
       first segment
                 y = *(float*)&i;
                 xh=2.0884723f*x;
                 y=y*(1.4954307f-xh*y*y*y);
                 y=y*(2.1094839f-xh*y*y*y);
            } else {
                   i= 0x549112DA-i;                   //
         third segment
                   y = *(float*)&i;
                   xh=0.47602674f*x;
                   y=y*(1.3634177f-xh*y*y*y);         // 1st
            iteration
                   y=y*(1.4575615f-xh*y*y*y);         //
          2nd iteration
              }
         }
       return y;
       }
```

4. Experimental performance verification

In our first experiment we checked what were the relative errors and accuracies of the method based on C library functions and custom approximation algorithms. For reference the algorithm RcpCbrt_7 was used. Figure 2 shows a graph of the initial approximations for $y_0(x)$ (solid lines) and the function *1/cbrt(x)* based on C <math.h> library function *cbrt(x)* (dashed line). The maximum difference between the plots is in the first half of the examined section of the argument $x \in [1,4)$. Figure 3 presents relative errors of initial approximations for $\delta_0 = (y_0(x)\cdot \mathrm{cbrt}(x) - 1)\cdot 100$. The maximum error for $x \in [1,8)$ was calculated $\Delta_{0\max} = y_0(x)\cdot \mathrm{cbrt}(x) - 1$ =0.4406, and it corresponds to 1.18 correct bits.

Let us examine the relative errors and accuracy of the algorithm RcpCbrt_7. In Figure 4 we can see the plot for the first iteration $y_1(x)$. Figure 5 shows a diagram of its relative error $\delta_1 = (y_1(x)\cdot \mathrm{cbrt}(x) - 1)\cdot 10^5$. The relative maximum error of the first iteration for $x \in [1,8)$ was calculated $\Delta_{1\max} = y_1(x)\cdot \mathrm{cbrt}(x) - 1$ =$5.5930\cdot 10^{-5}$, and it corresponds to 14.12 correct bits. The relative maximum error of the second iteration for $x \in [1,8)$ does not exceed $\Delta_{2\max} = y_2(x)\cdot \mathrm{cbrt}(x) - 1$ =$1.5204\cdot 10^{-7}$, and it corresponds to 22.64 correct bits.

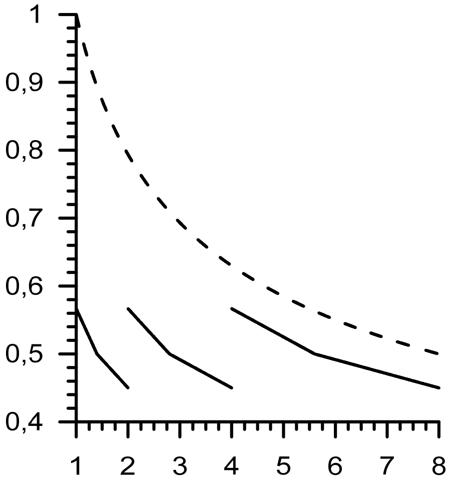

Figure 2. Functions $y_0(x)$ (solid lines) and 1/cbrt(x) (dashed line)

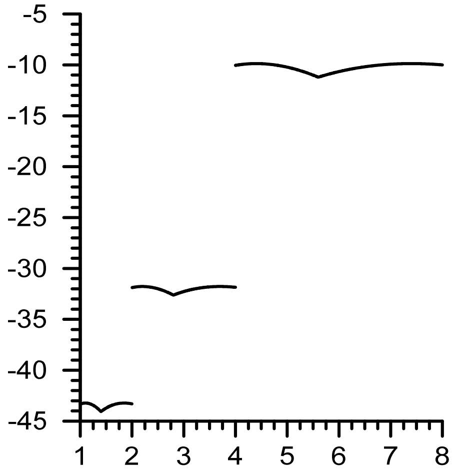

Figure 3. Relative errors $\delta_0 = (y_0(x)cbrt(x) - 1)10^2$

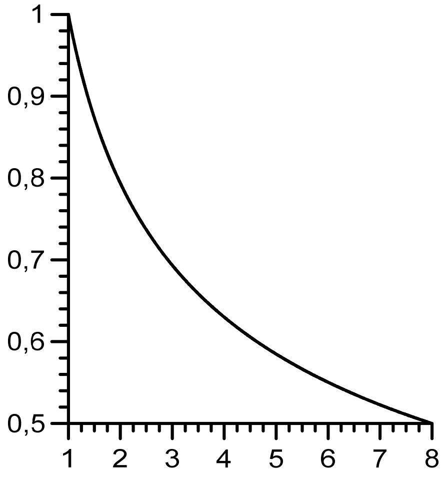

Figure 4. Function $y_1(x)$

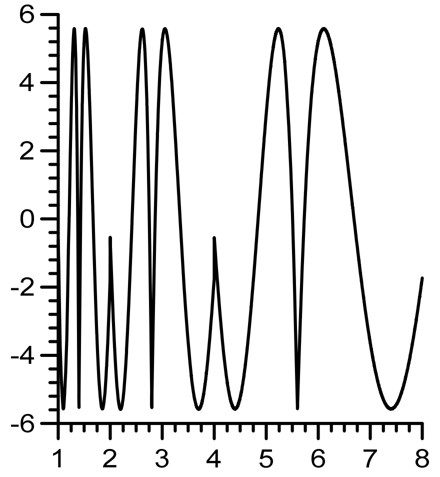

Figure 5. Relative errors $\delta_1 = (y_1(x)cbrt(x) - 1) * 10^5$

comparison we were utilizing 32-bit STM32F767ZIT6 microcontroller. For program compilation the optimization option -O3 was applied. The obtained results are presented in Table 1. We can observe that on the first iteration RcpCbrt_7 has a slightly lower accuracy but is by 5.65% faster than InvCbrt21. In the second iteration, these algorithms have similar accuracy but RcpCbrt_7 is by 2.14% faster than InvCbrt21.

Similarly, we tested experimentally the RcpCbrt_8 algorithm against InvCbrt21 and RcpCbrt_7 algorithms implemented on the same STM32F767ZIT6 microcontroller. For program compilation again the option -O3 was applied. The results presented in Table 1 show that the newest algorithm is 9.5% faster than InvCbrt21 and RcpCbrt_7. The efficiency and precision of the second final iteration of RcpCbrt_8 are essential. In fact, the first iteration of RcpCbrt_8 algorithm provides lower accuracy in comparison to the predecessors, but the second iteration compensates for this and brings the precision to the appropriate level.

Table 1 Comparison of three custom approximation algorithms on 32-bit microcontroller

Algorithm	Iteration no. i	Execution time (ns)	No. of cycles	Maximum negative error $\Delta_{i\,min}$	Maximum positive error $\Delta_{i\,max}$	No. of correct bits
InvCbrt21 [8]	1	203.75	43.0	$-2.6860 \cdot 10^{-5}$	$2.6825 \cdot 10^{-5}$	15.18
	2	287.11	62	**$-1.3276 \cdot 10^{-7}$**	**$1.3301 \cdot 10^{-7}$**	22.84
RcpCbrt_7	1	188.45	40.7	$-5.5930 \cdot 10^{-5}$	$5.5918 \cdot 10^{-5}$	14.12
	2	281.07	60.7	$-1.5204 \cdot 10^{-7}$	$1.487 \cdot 10^{-7}$	22.64
RcpCbrt_8	1	188.45	40.7	$-3.6797 \cdot 10^{-1}$	$-8.5179 \cdot 10^{-2}$	1.44
	2	**259.53**	**56.1**	$-1.704 \cdot 10^{-7}$	$1.5719 \cdot 10^{-7}$	**22.48**

We also compared the performance of three custom implementations of our algorithms with the performance of the algorithm based on the *cbrtf (x)* function taken from the C library <math.h>. In our test scenario we used the AMD CPU, Nvidia GPU and Graphcore IPU, verifying the execution time, cycle count and RMSD (root-mean-square deviation). In opposition to CPU, the execution time in GPU and IPU is significantly affected by communication between the host and GPU or IPU accelerators, so it doesn't make much sense to measure the time in the offload accelerator scenario.

The results of computational experiments performed on three different platforms are shown in Table 2. In all cases, we used the latest available version of the compilers and SDK for the given architecture and compilation with the -O3 maximum optimization flag. For x variable range [1, 8) and minimal step determined by the function *nextafter* the number of iterations equals 25165824. The function *1/cbrtf(x)* out of C <math.h> library was used as the reference (predicted) value for calculation of RMSD (Root-Mean-Square_Deviation) of custom implementations of the three investigated algorithms.

Table 2. Experimental comparison of three custom C implementations of algorithms for computing the function $1/\sqrt[3]{x}$ versus the C <math.h> library function *1/cbrtf(x)*.

PLATFORM	AMD EPYC 7742 64-Core Processor/ g++ (Ubuntu 7.5.0-3 ubuntu1~18.04) 7.5.0			Nvidia GPU A100 / Nvidia Cuda 11.5.1_496.13		Graphcore IPU - M2000 system running on single core / Poplar SDK 2.3	
STEPS / ITERATIONS	n/a		25165824	n/a	25165824	n/a	25165824
Algorithm	Execution time (ns)	Cycle count	RMSD	Cycle count	RMSD	Cycle count	RMSD
STANDARD C LIBRARY IMPLEMENTATION							
1/cbrtf(x) (<math.h>)	16	37	reference value	167	reference value	22 507	reference value
CUSTOM C IMPLEMENTATIONS							
InvCbrt21 [8]	11	25	**3,90E-08**	131	**3,91E-08**	321	**3,81E-08**
RcpCbrt_7	11	26	4,00E-08	127	3,92E-08	325	3,92E-08
RcpCbrt_8	**9**	**21**	4,26E-08	**112**	4,18E-08	**319**	4,18E-08

The obtained comparison results clearly show that RcpCbrt_8 outperforms *1/cbrtf(x)* and the two other implementations discussed in this article in terms of computation time. On AMD CPU we obtained the execution time 9 ns for RcpCbrt_8 versus 11 ns for RcpCbrt_7 and InvCbrt21, as well as 16 ns for *1/cbrtf(x)*. Also the number of cycles for RcpCbrt_8 is the lowest. For the Nvidia A100 GPU we have 112 cycles for RcpCbrt_8 and difference against RcpCbrt_7 and InvCbrt21 looks very similar to the scenario we observed with the CPU. On the Graphcore IPU the difference is not such significant but still RcpCbrt_8 performs with the lowest number of cycles. The coincidence of RMSD values for GPU and IPU for the two new algorithms is no surprise.

5. Conclusions

The article presents the theoretical background for the selection of optimal magic constants in our new algorithms for computing the reciprocal cube roots of floating-point numbers. We have shown that the quality of the selection of the magic constants plays a fundamental role in minimizing the relative error values for the initial Newton-Raphson approximation.

The idea of the present work was to modify algorithm InvCbrt21 presented in [8] in order to reduce the number of arithmetic operations and finally decreasing the calculation time of $1/\sqrt[3]{x}$ function. The achieved results show that the accuracy of the calculation decreased to <1.0 bits for the first iteration of the RcpCbrt_7 algorithm. In the second iteration the difference in accuracy is minimal and equals 0.2 bits. The final proposed modification implemented in RcpCbrt_8 makes this algorithm by 7,5-9,5% faster than InvCbrt21 – the best algorithm presented in [8]. At the same time, the accuracy of the calculations of RcpCbrt_8 dropped to 0.4 bits only versus InvCbrt21. Ultimately, the RcpCbrt_8 algorithm is the fastest one with approximately the same accuracy. Additionally, this algorithm enables dividing the segment x∈ [1,8) into 3 subsegments, which would improve the accuracy of the initial approximation when necessary and reduce the number of iterations to one.

For the performance verification of our algorithms we used 32-bit STM32F767ZIT6 microcontroller, but we also made the comparison of the algorithms on AMD CPU, Nvidia GPU and Graphcore IPU platforms. The obtained results clearly show that the algorithm RcpCbrt_8 provides the best performance characteristics among the three examined approximation algorithms.

References

[1] Bailey D. and Borwein, J. (2012). Ancient Indian Square Roots: An Exercise in Forensic Paleo-Mathematics, American Mathematical Monthly. 119, doi: 10.4169/amer.math.monthly.119.08.646.

[2] Long Division Method: https://www.wikihow.com/Calculate-Cube-Root-by-Hand

[3] Singh Y.K. Computing cube root of a positive number. ADBU-Journal of Engineering Technology, AJET, Volume 4(1), 2016, pp. 85-89, ISSN: 2348-7305,

[4] Beasley, A.E., Watson, R.J., Clarke, C.T. "Efficient digital implementaton of a multi-precision square-root algorithm", IET Computers & Digital Techniques. 2019, Vol. 13 Issue 2, pp. 110-117, doi: 10.1049/iet-cdt.2018.5051

[5] Peng, H. "Algorithms for extracting square roots and cube roots", 1981 IEEE 5th Symposium on Computer Arithmetic (ARITH), 1981, pp. 121-126, doi: 10.1109/ARITH.1981.6159287.

[6] Guardia C. M. and Boemo E., "FPGA implementation of a binary32 floating point cube root", 2014 IX Southern Conference on Programmable Logic (SPL), 2014, pp. 1-6, doi: 10.1109/SPL.2014.7002202.

[7] Putra R. V. W. and Adiono T., "Optimized hardware algorithm for integer cube root calculation and its efficient architecture", 2015 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), 2015, pp. 263-267, doi: 10.1109/ISPACS.2015.7432777.

[8] Moroz, L., Samotyy, V.; Walczyk, C.J.; Cieślinski, J.L. "Fast Calculation of Cube and Inverse Cube Roots Using a Magic Constant and Its Implementation on Microcontrollers", Energies, 2021, 14, 1058, doi: 10.3390/en14041058

[9] Moroz, L.; Samotyy, V.; Horyachyy, O.; Dzelendzyak, U. "Algorithms for calculating the square root and inverse square root based on the second-order Householder's method", In Proceedings of the 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 18–21 September 2019; pp. 436–442. doi: 10.1109/IDAACS.2019.8924302