Т. 1, № 1, 2019

УДК 519.7

Л. В. Мороз, А. Гринчишин, О. Я. Горячий
Національний університет "Львівська політехніка",
кафедра безпеки інформаційних технологій

ПРОСТА МОДИФІКАЦІЯ АЛГОРИТМУ ШВИДКОГО ОБЧИСЛЕННЯ ЗВОРОТНОГО КВАДРАТНОГО КОРЕНЯ ДЛЯ ЧИСЕЛ З РУХОМОЮ КОМОЮ ОДИНАРНОЇ ТОЧНОСТІ

Опис прості алгоритма швидкого добування зворотного квадратного кореня з використанням магічної константи зі зменшеними відносними похибками обчислень для чисел типу float.

Ключові слова: магічна константа, числа типу float, стандарт IEEE-754, відносна похибка обчислень, зворотний квадратний корінь з рухомою комою.

Simple algorithms of the fast inverse square root with the use of magic constant with reduced relative errors for numbers of type float are described in the paper.

Keywords: magic constant, floating-point numbers, IEEE-754 standard, relative error, fast inverse square root.

Вступ

Операції добування квадратного кореня та зворотного квадратного кореня є базовими для багатьох застосувань [1–16]. Значна кількість обчислювальних засобів та бібліотек підтримують операцію квадратного кореня відповідно до стандарту арифметики з рухомою комою IEEE-754. У більшості сучасних процесорів (CPU) добування квадратного кореня виконується за допомогою вбудованої апаратної чи програмної функції SQRT (24 точних біти мантиси для чисел одинарної точності). Однак вона має суттєвий недолік – її виконання потребує великої кількості тактів процесора [17]. Для прискорення обчислень використовують апаратну наближену інструкцію зворотного квадратного кореня RSQRT (reciprocal square root), за якою легко отримати квадратний корінь. Ця інструкція, побудована на основі переглядових таблиць (lookup tables), є менш точною – приблизно 11.5 бітів мантиси для процесорів Intel. З іншого боку, RSQRT набагато швидша, ніж SQRT. Крім того, інструкцію RSQRT можна використовувати у векторному вигляді (SIMD – single instruction, multiple data) для конвеєризації обчислень. Варто зауважити, що в більшості CPU інструкція RSQRT існує лише в одному варіанті виконання – лише для чисел одинарної точності (чисел типу float). Однак в сучасних CPU від Intel, що реалізують розширення набору команд AVX-

512, існують ще 14- та 28-бітні версії апаратних інструкцій RSQRT вже у двох варіантах: для чисел одинарної та подвійної точності (float та double) [18]. Подібна ситуація спостерігається і в сучасних мікроконтролерах на базі процесорів ARM Cortex, що підтримують розширення набору команд NEON – там теж є свої повільніші функції SQRT, що дають повну точність, і швидкі наближені інструкції RSQRT для чисел одинарної та подвійної точності (8.25 точних бітів мантиси в обох випадках) [12]. Для підвищення точності обчислень із використанням цих наближених інструкцій розробникам пропонується доповнювати їх ітераціями Ньютона–Рафсона в класичній формі. Набір команд NEON навіть має спеціальну апаратну інструкцію для цього. Проте, недоліком таких підходів є прив'язаність до конкретних апаратних платформ, що реалізують відповідні інструкції.

Незважаючи на це, більшість мікроконтролерів, що підтримують обчислення з рухомою комою, можуть використовувати лише повільні (але точні) бібліотечні функції, наприклад SQRT із стандартної математичної бібліотеки C (math.h). При цьому, для обчислення зворотного квадратного кореня доводиться додатково застосовувати операцію ділення 1.0f/SQRT. Для таких пристроїв альтернативою можуть слугувати алгоритми, що базуються на використанні методу швидкого зворотного квадратного кореня (FISR – fast inverse square root) [4, 5, 7–10, 13–16, 19–25], які є достатньо точними і відносно швидкими. У статті описано такі алгоритми та їхні прості модифікації. Наша основна мета полягає у підвищенні точності алгоритму для чисел типу float за незначного зменшення швидкодії.

Огляд відомих алгоритмів

Існує теорія функціонування алгоритмів за методом швидкого зворотного квадратного кореня [6, 19, 20, 23–25]. Швидке початкове наближення алгоритму формується завдяки "трюку" із двійковим представленням чисел з рухомою комою в пам'яті та використовує "магічну константу" для зменшення похибки такого наближення. На цьому етапі алгоритму застосовується арифметика над цілими числами. Після цього застосовується ітераційний метод Ньютона–Рафсона обчислення зворотного квадратного кореня для покращення отриманих результатів. Також в літературі цей метод було модифіковано для обчислення й інших степеневих функцій, таких як оберненого числа та зворотного кубічного кореня.

Наведемо декілька прикладів найвідоміших модифікацій алгоритму FISR для зворотного квадратного кореня. Перший з них – класичний [19]:

```
float InvSqrt1(float x){
float xh = 0.5f * x;
int i = *(int*)&x;
i = 0x5f375a86 - (i>>1);
x = *(float*)&i;
x = x*(1.5f - xh * x * x);
x = x*(1.5f - xh * x * x);
return x;
}
```

Якщо максимальну відносну похибку після другої ітерації позначити як $\delta_{2\max}$, то точність цього алгоритму становитиме лише $|\delta_{2\max}| = 4.86 \cdot 10^{-6}$, або $-\log_2(|\delta_{2\max}|) = 17.65$ коректних бітів.

Покращений алгоритм (із модифікованими ітераціями Ньютона–Рафсона) наведено в [25]:

```
float InvSqrt2(float x){
float halfnumber = 0.5f*x;
int i = *(int*) &x;
i = 0x5F376908 - (i>>1);
x = *(float*) &i;
x = x*(1.50087896f - halfnumber*x*x);
x = x*(1.50000057f - halfnumber*x*x);
return x;
}
```

Тут $|\delta_{2\max}| = 7.37 \cdot 10^{-7}$, що відповідає $-\log_2(|\delta_{2\max}|) = 20.37$ коректним бітам результату.

Найкращий щодо точності алгоритм описано у [9]:

```
float InvSqrt3(float x){
int i = *(int*)&x;
i = 0x5f5ffff8 - (i >> 1);
float y = *(float*)&i;
y = 0.248884737f*y*(4.778488636f - x*y*y);
float c = x*y;
c = fmaf(y, -c, 1.00000065f);
y = fmaf(y, 0.5f*c, y);
return y;
}
```

Тут використовується функція поєднаного множення-додавання fmaf, що виконується із одноразовим заокругленням результату, а отже, підвищує точність обчислень порівняно з послідовним виконанням цих операцій. У більшості CPU ця функція має швидкий апаратний відповідник. Похибки цього алгоритму становлять:

$|\delta_{1\max}| = 6.5025 \cdot 10^{-4}$, чи $-\log_2(|\delta_{1\max}|) = 10.59$ коректних бітів,

$|\delta_{2\max}| = 4.0870 \cdot 10^{-7}$, чи $-\log_2(|\delta_{2\max}|) = 21.22$ коректних бітів.

Шістнадцяткові цілі числа 0x5f375a86, 0x5f376908 та 0x5f5ffff8, що використовуються в цих алгоритмах, і є згаданими вище магічними константами (magic numbers). У всіх описаних алгоритмах застосовано дві класичні ітерації Нь

ютона–Рафсона чи їхні модифікації, що мають другий порядок збіжності.

Пропоновані алгоритми

Запропоновано новий підхід до побудови алгоритму – розбиття нормалізованого діапазону вхідного аргументу $x \in [1,4)$ на дві окремі ділянки $x \in [1,2)$ та $x \in [2,4)$. Перший варіант алгоритму – застосування до обох ділянок однієї модифікованої формули Ньютона–Рафсона на першій ітерації. Залежно від піддіапазону (на ділянці $x \in [1,2)$) ми модифікуємо вхідний аргумент, а після першої ітерації коригуємо результат (див. алгоритм InvSqrt41). Це дає змогу суттєво підвищити точність обчислень. Зауважимо, що друга ітерація, що починається з оголошення та обчислення змінної *c*, містить класичні коефіцієнти та є спільною для двох ділянок.

```
float InvSqrt41(float x){
float xx = x;
int i = *(int*)&x;
int k = i & 0x00800000;
if (k == 0x00800000){
    i  = i & 0xff7fffff;
    x = *(float*)&i;
}
i = 0x5f99e8b6 - (i >> 1);
float y = *(float*)&i;
y = 0.103027083f*y*(8.5998040f - x*y*y);
if (k == 0x00800000)
    y = y*0.707106781186f;
float c = xx*y;
c = fmaf(y, -c, 1.0f);
y = fmaf(y, 0.5f*c, y);
return y;
}
```

Максимальні відносні похибки для першої ітерації складають

$\delta^{+}_{1\max} = 7.462460 \cdot 10^{-5}$, $\delta^{-}_{1\max} = 7.465327 \cdot 10^{-5}$, чи 13.70 коректних бітів,

а для другої –

$\delta^{+}_{2\max} = 7.381320 \cdot 10^{-8}$, $\delta^{-}_{2\max} = 8.021126 \cdot 10^{-8}$, чи 23.57 коректних бітів.

Як бачимо, після першої ітерації відбулося суттєве зменшення максимальної відносної похибки порівняно з алгоритмом InvSqrt3 – більш ніж у 8.7 разу. Точність всього алгоритму також суттєво краща (більше ніж на 2 біти).

Описаний вище алгоритм можна прискорити застосуванням власних магічних констант та ітераційних рівнянь окремо на кожній ділянці. Це стосується лише першої модифікованої ітерації. Отже, другий варіант реалізації алгоритму матиме вигляд:

```
float InvSqrt42(float x){
int i = *(int*)&x;
int k = i & 0x00800000;
if (k != 0x00800000){
    i = 0x5f99e8b6 - (i >> 1);
    float y = *(float*)&i;
    y = 0.103027083f*y*(8.599804f - x*y*y);
}
else {
    i = 0x5f59e8b6 - (i >> 1);
    float y = *(float*)&i;
    y = 0.291411832f*y*(4.2998304f - x*y*y);
}
float c = x*y;
c = fmaf(y, -c, 1.0f);
y = fmaf(y, 0.5f*c, y);
return y;
}
```

Тут максимальні відносні похибки обох знаків такі: після першої ітерації –

$\delta^{+}_{1\max} = 7.462300 \cdot 10^{-5}, \delta^{-}_{1\max} = 7.462916 \cdot 10^{-5},$ чи 13.70 коректних бітів;

після другої ітерації –

$\delta^{+}_{2\max} = 7.381320 \cdot 10^{-8}, \ \delta^{-}_{2\max} = 8.021126 \cdot 10^{-8},$ чи 23.57 коректних бітів.

Як бачимо, вони практично не відрізняються (лише незначна відмінність після першої ітерації).

Висновки

У статті запропоновано прості алгоритми швидкого добування зворотного квадратного кореня з використанням магічної константи. Дослідження показують, що максимальні відносні похибки описаних алгоритмів зменшені більш ніж у 8,7 разу після першої ітерації та більш ніж у 5 разів після другої ітерації для чисел типу float.

References

1. Multiplier-free divide, square root, and log algorithms / F. Auger, Z. Lou, B. Feuvrie, F. Li // IEEE Signal Process. Mag. 2011. Vol. 28. No. 4. P. 122–126.

2. Allie M. A. Root of Less Evil / M. Allie, R. Lyons // IEEE Signal Process. Mag.: DSP Tips and Tricks. 2005. Vol. 22. P. 93–96.

3. Parhami B. Computer Arithmetic: Algorithms and Hardware Designs / B. Parhami. 2nd ed. New York : Oxford Univ. Press, 2010.

4. Lemaitre Florian. Cholesky Factorization on SIMD multi-core architectures / Florian Lemaitre, Benjamin Couturier, Lionel Lacassagne // Journal of Systems Architecture. Elsevier, 2017. Vol. 79. P. 1–15.

5. A Fast FPGA Based Architecture for Computation of Square Root and Inverse Square Root / A. Hasnat, T. Bhattacharyya, A. Dey, S. Halder, D. Bhattacharjee // Devices for Integrated Circuit (DevIC): int. conf., 23–24 Mar., 2017. Kalyani, 2017. P. 383–387.

6. Beebe N. H. F. The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library / N. H. F. Beebe. Springer, 2017.

7. Optimizations of Two Compute-bound Scientific Kernels on the SW26010 Many-core Processor / J. Lin, Z. G. Xu, A. Nukada, N. Maruyama, S. Matsuoka // 46th International Conference on Parallel Processing, 14–17 Aug. 2017. Bristol : IEEE, 2017. P. 432–441.

8. Improving Deep Learning By Inverse Square Root Linear Units (ISRLUS) / Brad Carlile, Guy Delamarter, Paul Kinney, Akiko Marti, Brian Whitney. 2018.

9. Andriy Hrynchyshyn. An efficient algorithm for fast inverse square root / Hrynchyshyn Andriy, Horyachyy Oleh, Tymoshenko Oleksandr // Przetwarzanie, transmisja i bezpieczeństwo informacji. Bielsko-Biała : Wydawnictwo Naukowe ATH w Bielsku-Białej, 2018. T. 2. P. 105–113.

10. Hanninen T. Novel detector implementations for 3G LTE downlink and uplink / T. Hanninen, J. Janhunen, M. Juntti // Analog. Integr. Circ. Sig. Process. 2014. Vol. 78. No. 3. P. 645–655.

11. Floating point unit demonstration on STM32 microcontrollers: Application note AN4044. STMicroelectronics N.V., 2016.

12. ARM® NEON™ Intrinsics Reference: IHI 0073B. ARM Limited, 2016.

13. Hsu C. J. An Efficient Hardware Implementation of HON4D Feature Extraction for Real-time Action Recognition / C. J. Hsu, J. L. Chen, L. G. Chen // IEEE International Symposium on Consumer Electronics (ISCE). 2015.

14. A UWB Radar Signal Processing Platform for Real-Time Human Respiratory Feature Extraction Based on Four-Segment Linear Waveform Model / C. H. Hsieh, Y. F. Chiu, Y. H. Shen, T. S. Chu, Y. H. Huang // IEEE Trans. Biomed. Circ. Syst. 2016. Vol. 10. No. 1. P. 219–230.

15. Ziqiang Li. OFDM Synchronization implementation based on Chisel platform for 5G research / Li Ziqiang, Chen Yun, Zeng Xiaoyang // IEEE 11th International Conference on ASIC (ASICON). Chengdu : IEEE, 2015. P. 1–4.

16. Sangeetha D. Efficient Scale Invariant Human Detection using Histogram of Oriented Gradients for IoT Services / D. Sangeetha, P. Deepa // IEEE 30th International Conference on VLSI Design and 16th International Conference on EmbeddedSystems. Hyderabad : IEEE, 2017. P. 61–66.

17. Fog A. Software optimization resources, Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs [Electronic resource] / A. Fog. Regime of access: http://www.agner.org/optimize/.

18. x86 and amd64 instruction reference [Electronic resource]. Regime of access: http://www.felixcloutier.com/x86/index.html.

19. Lomont C. Fast inverse square root [Electronic resource] / C. Lomont // Purdue University : Tech. Rep. – 2003. – Regime of access: http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

20. Blinn J. Floating-point tricks / J. Blinn // IEEE Comput. Graphics Appl. IEEE, 1997. Vol. 17. No. 4. P. 80–84.

21. Zafar S. Hardware architecture design and mapping of "Fast Inverse Square Root's algorithm" / S. Zafar, R. Adapa // International Conference on Advances in Electrical Engineering (ICAEE). 2014. P. 1–4.

22. Martin P. Eight Rooty Pieces / P. Martin // Overload Journal. No. 135. 2016. P. 8–12.

23. Fast calculation of inverse square root with the use of magic constant – analytical approach / L. Moroz, C. J. Walczyk, A. Hrynchyshyn, V. Holimath, J.L. Cieslinski // Appl. Math. Computation. Elsevier, 2018. Vol. 316.P. 245–255.

24. Eberly D. H. GPGPU Programming for Games and Science / D. H. Eberly. Florida : CRC Press, 2015.

25. Walczyk C. J. Improving the accuracy of the fast inverse square root algorithm [Electronic resource] / C. J. Walczyk, L. V. Moroz, J. L. Cieslinski. – arXiv preprint arXiv: 1802.06302. 2018 Regime of access: https://arxiv.org/pdf/1802.06302.pdf.