Square Roots and Ways to Calculate Them Faster

Ammar Rasyad Chaeroel - 13521136
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): ammarasyad@gmail.com

***Abstract*—Square roots have always been considered a relatively computationally intensive task for the CPU compared to other commonly used mathematical operations. The most widely used method of calculating square roots is by using Newton's method. This paper is written to discuss the already existing approaches in calculating square roots and determine which is best, starting from varying algorithm techniques to utilizing already available resources/instructions of the CPU.**

Keywords—mathematics; algorithm; complexity; brute-force

I. Introduction

A square root is a mathematical function or operation, which defines a number y such that $y^2 = x$, y is the square root of x. It is one of the most fundamental and commonly used mathematical functions that is in use in computer algorithms to this day, including but not limited to distance calculation, vectors, etc. Without a fast way to compute square roots, real-time computation in programs such as games would not be possible. The question is how do CPUs calculate square roots fast enough to necessitate real-time compute?

Modern CPUs have a component called FPU (Floating-Point Unit) that assists the CPU in floating-point compute. This unit is extremely important, as without the FPU, the CPU would have a hard time calculating floating-point compute, thus making it slow or even impossible for real-time uses. At low-level, the CPU uses instructions dedicated for square root operations, such as FSQRT, SQRTSS, SQRTSD, and so on.

There are algorithms with various techniques/approaches to calculate the square root of a number. Those are: Newton's Method/Babylonian Method/Heron's Method, Bisection Search, etc.

II. Theoretical Basis

A. Brute Force

Brute-forcing, known as the naïve algorithm, operates exactly like its name. This type of approach brute-forces (keeps calculating) until the algorithm is satisfied with their defined parameters. Typically, the algorithm stops when it reaches the maximum number of iterations done by the algorithm, which can either be user-defined or a constant variable defined by the developer. At its core, brute-forcing does not inherently care whether the result is the most optimal result, it just stops depending on the parameters defined.

This results in varying accuracy depending on the parameters. In its nature it does not know and cannot know anything beforehand, therefore it must find the appropriate approach to the result on its own. This makes brute-forcing one of, if not, the slowest algorithm of the bunch, but with the proper approach (and enough time), it can be the most accurate of them all.

B. Babylonian Method

YBC (Yale Babylonian Collection) 7289 is an Old Babylonian mathematical clay tablet. The identity of the maker, the provenance, and the exact date is unknown. They correctly calculated the square root of 2 to three sexadecimal digits after 1.

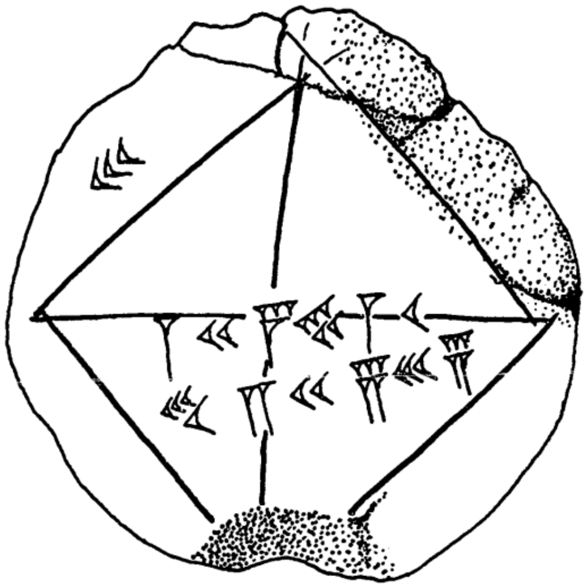

Figure 1 YBC 7289 from [2]

A square with its two diagonals, and on the side is the number 30. The diagonals have the numbers 1, 24, 51, 10, 42, 25, 35. At first glance, it is unclear what the numbers mean. It has been deciphered that the numerical base used in this tablet is sexagesimal (base 60). The number 30, which represents the side length of the square, on the sexagesimal system is the reciprocal of 2 (60/2 = 30) (the *square* root of *2*). The numbers 1;24;51;10 are in sexagesimal figures, and represents the decimal number 1.41421296… ($1 + 24/60 + 51/60^2 + 10/60^3$)

that is only off by less than one part in two million. The numbers 42;25;35 represent the length of the diagonal with the side length of 30, which is 42.4263889… (42 + 25/60 + $35/60^2$).

Although there is a missing piece of information from the tablet, and that is the place value of each digit. Under the same interpretation as before (30/60 = 1/2), the number on the diagonals, when done under the same process (0;42;25;35), equals to 0.70711… (0 + 42/60 + $25/60^2$ + $35/60^3$), which is the approximation of $1/\sqrt{2}$, also off by less than one part in two million.

From this information, it's not exactly obvious how the Babylonian method is derived from the clay tablet. When visualized as a square:

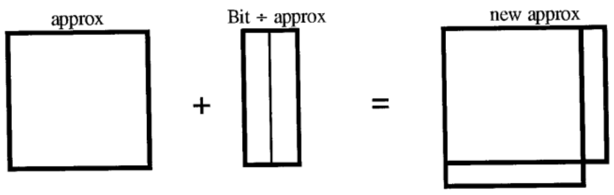

Figure 2 Visualization of the method, taken from [2]

Suppose we want to evaluate the side of the square, which is the square root, it can be approximated as:

$$x^2 = \text{approx}^2 + \text{bit}$$

which can be represented geometrically by the sum of a square with sides *approx* and *bit*. With *bit* as a rectangle with the side *approx*, cut this in two lengthwise. Hence

$$x = \sqrt{(a^2 + B)} \approx a + ½\, B/a$$

That is just for *new approx*, but for the area of the little square missing on the bottom right corner, we would require a new approximation. This is done by subtracting approx^2 with bit, instead of adding. Thus, a new approximation is revealed to be:

$$x = \sqrt{(a^2 - B)} \approx a - ½\, B/a$$

Let's say we are trying to get the square root of 2. It goes as follows:

$$\sqrt{2} = \sqrt{((3/2)^2 - ¼)} \approx 3/2 - 1/2 \times 1/4 \times 2/3 = 17/12$$

17/12 works out to be 1.4167, which is not far off from the square root of 2, but it is close enough. In the sexagesimal system, the number 17/12 is represented as 1;25. If we are to apply the procedure again:

$$\sqrt{((17/12)^2 - 25/60^2)} = \sqrt{((1;25)^2 - 0;00\ 25)} \approx 1;24\ 51\ 10\ 35\ 17\ldots$$

If we truncate to 1;24 51 10, we get our original solution from the clay tablet.

Heron's iterative method, which is also known as the aforementioned Newton's method, is an iterative method of the Babylonian method. The accuracy improves with more iterations.

For $\sqrt{a}$, Heron's method to calculate the square root would be:

$$x_{n+1} = ½\,(x_n + a/x_n)$$

where x_n is the n-th iteration value of the square root approximation, with the initial value x_0. The initial value does not matter, but the further the initial value is from the square root, the more iterations are required to achieve the same level of accuracy as with an initial value closer to the square root.

For example, we would like to calculate the square root of 76. The closest perfect square to 76 is 64, and the square root of 64 is 8. Therefore, the square root of 76 is between 8 and 9.

$$x_0 = 8$$

$$x_1 = ½\,(8 + 76/8) = 8.75$$

$$x_2 = 8.71785714$$

$$x_3 = 8.717797887\ldots$$

and so on.

Heron's iterative method is quadratically convergent, which means with every iteration, the error is approximately the square of the error in the previous step. In other words, the number of correct digits of the approximation roughly doubles with each iteration. With the correct answer being 8.71779789, in x_1, the number of correct digits (including 8) is 2. In x_2, it is 4, and so on and so forth.

C. *Bisection Method*

Bisection method is a root-finding method of a continuous polynomial function. It separates the interval and subdivides the interval in which the square root lies. By narrowing the gap between the positive and negative intervals, it gets closer to the correct answer. It is also known as the binary search method.

Bisection method works iteratively, just like previous methods. It's still relatively slower for real-time use (and compared to the Babylonian method), but it is faster than brute-forcing.

For any continuous function f(x):

- find two points a and b such that $f(a) * f(b) < 0$,
- find the midpoint t, and t is the root if $f(t) = 0$ (or converges to zero). If it does not,
- divide the interval [a, b], if $f(t) * f(a) < 0$, the root is between t and a. Conversely, if $f(t) * f(b) < 0$, the root is between t and b. Repeat until $f(t) = 0$.

When the algorithm stops is usually determined by a pre-defined accuracy variable by getting the difference between t^2 and x (the value which square root is to be determined). This provides a varying compromise between accuracy and time, and its flexibility allows developers to finely tune the algorithm to fit their use case.

D. Built-in CPU instruction

This is more of using what is already provided by the CPU rather than an algorithm technique, per se. Modern CPUs have what is called an FPU, which is already explained in part I. The FPU is responsible for floating-point arithmetic, and that includes square root calculations. By using the built-in CPU instruction, we can remove all the overhead from previous methods and provide a quick result.

Modern CPUs have a SIMD (Single Instruction Multiple Data) instruction sets, which provide data-level parallelism on a vector of data, executed in parallel with a single instruction as opposed to multiple instructions. There are different SIMD instruction sets available on modern CPUs, such as SSE (Streaming SIMD Extensions), SSE2, SSE3, SSE4, AVX, AVX2, AVX512, and so on. SSE and its subsequent successors have their own floating-point registers XMM0 through XMM7, totaling 8 registers. These registers are 128-bit wide, which means it can fit 4 single-precision floating-point numbers, 2 64-bit double-precision floating-point numbers, 2 64-bit integers, 4 32-bit integers, 8 16-bit short integers, or 16 8-bit characters.

SSE was superseded by AVX, which expanded the registers to 256-bit registers, and named YMM, with XMM being the lower 128-bit register of a single 256-bit YMM register. This allows the CPU to store 8 32-bit single-precision floating-point numbers and 4 64-bit double-precision floating point numbers, instead of 4 and 2 respectively, allowing for even more parallelism. AVX2 is the successor to AVX, which expands most integer instructions to 256-bit and adds several new instructions.

Due to how wide the instructions are, they generate more heat and consume more power than other regular CPU instructions. Using AVX instructions for real-world applications, which tend to use AVX instructions repeatedly, may lead to a higher power consumption, therefore lower power efficiency.

III. Approaches to Calculate Square Roots

With the information in part II, we have multiple ways to calculate square roots, one brute-force, two algorithms with different approaches, and one algorithm using the built-in AVX(2) instruction in the CPU.

All these algorithms were run by an Intel Core i7-10750H with a 5 GHz single-core boost. Results may vary in different processors and compiler optimization flags, but the difference between the algorithms should be the same across different processors. The link to the source code is provided in this paper.

First, the control variable, which is using the built-in *sqrt* method in C++ is determined. Specifically for this single test, the randomly determined double-precision floating-point number 12837.3284290217631 is used, with the control variable (the square root of the number) being 113.301934798227364. All tests will have the decimal precision set to 18, which means 18 digits after the decimal point.

With the control variable set, there are 5 series of tests, each using different algorithms, that run a million (1.000.000) times to put into scale the differences between the algorithms and (at least attempt to) simulate real-world use which does these calculations multiple times. Each of these tests have a 100ms interval between them, to let the CPU “rest” between each test. This lowers the variability between test suites.

First test, the “native” test, which uses the built-in C++ function to compute the square root.

```
Type NATIVE: 5.5322 ms
```

It takes 5.53ms for the C++ function to compute the square root 1 million times. Dividing that by 1 million, we get 5.53 nanoseconds per iteration. This is more than enough for real-time compute. The “native” test being the control variable means we do not need to compare the result of this with the control variable.

Second test, the brute-force method. The first thing the algorithm determines is the closest square to the number. If the number is a perfect square number, and the closest square equals the number, the algorithm stops. If not, the algorithm continues.

Suppose the closest square number is x and the control variable (the square root we’re looking for) is y. If x^2 is less than y, the algorithm adds the step variable 0.1 to x until x^2 is greater than or equal to y. Once the process stops, x is subtracted by 0.1, and the step variable is divided by 10. The process repeats for a set maximum number of iterations, in this case 10 iterations.

How does the algorithm fare?

```
Type BRUTEFORCE: 388.592 ms
```

It certainly does not look good for the brute-force algorithm. With 1 million iterations, it almost reached 500 milliseconds, which translates to 0.38 microseconds per iteration. Considering the time complexity of the algorithm of O(n), the lower the input number, the faster the algorithm goes. This algorithm is certainly not suitable for real-time use.

How far off is the result?

```
Control: 113.301934798227364

Brute force: 113.301934798199952

Difference: 2.74127387456246652e-11
```

The algorithm overshoots the control variable by 99.9999921%. This is a more than acceptable margin of error. In just 10 iterations, it can provide such an accurate approximation.

Moving on to the third, the Babylonian method, or (in this case) more accurately, Heron's iterative method. In this implementation, it works by setting the desirable accuracy variable.

Suppose we have the same variables as the previous test. The accuracy variable is then compared to the difference between x^2 and *y*. If the difference is ever so slightly greater than the accuracy variable, then we have acquired our answer.

If that condition has not been reached yet, the algorithm changes *x* to be equal to ½ *(x + y/x)*. The algorithm will repeat until that condition is true.

So how does it fare against other algorithms?

```
Type BABYLONIAN: 117.265 ms
```

This method is 331% faster than the brute-force method, taking 0.117 microseconds per iteration. With a time complexity of O(log(log(n/m)), where *n* is the input and *m* is the permissible margin of error, the algorithm is faster than the brute-force method. How accurate is it, though?

```
Control: 113.301934798227364

Babylonian: 113.301934798227393

Difference: -2.84217094304040074e-14
```

It is much more accurate than the brute-force method, while massively reducing the time spent calculating. Although promising, it is simply not enough to necessitate real-time use.

Fourth, the Bisection method. In this implementation, there are 3 declared variables, *left*, *right*, and *mid*. An additional accuracy variable is also defined like the previous method.

Suppose *x* is the result and *y* is the control variable to approach to by the algorithm. What the algorithm does first is define *left* as zero, *right* as the input number, and *mid* as *(left + right)/2*.

While the difference between x^2 and *y* is less than the accuracy variable, the algorithm does three things:

- If mid^2 is greater than the input number, *right* equals *mid*.
- Else, *left* equals *mid*.
- *mid* is recalculated by the same formula as before (*(left + right)/2*). The algorithm repeats until the defined condition is satisfied.

How is the performance?

```
Type BISECTION: 253.518 ms
```

The algorithm is slower than the Babylonian/Heron's method, while being faster than the brute-force algorithm. Time taken per iteration is around 0.253 microseconds, which puts it nicely in the middle, between the Babylonian method and the brute-force method.

```
Control: 113.301934798227364

Bisection: 113.301934609560462

Difference: 1.88666902545264747e-07
```

It is the worst of the bunch, being much less accurate by orders of magnitude above all methods. This method is much less preferable than previous methods, as it does not provide the right balance between run-time and accuracy. It can still be used when accuracy is of lesser importance, but the time required to calculate is just not preferable.

How about tweaking the accuracy variable?

```
Control: 113.301934798227364

Bisection: 113.301934798226682

Difference: 6.82121026329696178e-13
```

It looks like it's getting there. Just by making the accuracy variable smaller (in this case, 1e-4 to 1e-9), the result is already much more accurate.

But what is the cost?

```
Type BISECTION: 407.298 ms
```

The simple change in the accuracy variable results in a much higher run-time, 60.66% slower than the previous, less accurate run. It's even slower than the brute-force method, while being less accurate.

Lastly, the AVX method. This method uses the AVX(2) instruction *vsqrtpd* in the CPU. Bypassing all the C++ function overhead and several variable checks, this function is written in Assembly to provide a much more accurate result in run-time.

Generally, the lower level the language is, the faster it is, as developers have much better control over the code. It has its own drawbacks though, as Assembly is harder to write compared to C++. This assembly function is written by the author without any references, except for the instruction used, which shows how difficult it is to find the proper resources needed.

How does it compare to others?

```
Type ASM: 3.3549 ms
```

It's much faster than any other method, including C++'s own implementation, being faster by 60.64%. Using this method, developers can process 4 64-bit numbers at a time, with virtually zero slowdown. This is a much preferable method of calculating square roots in a time-sensitive, largescale environment, as it has data-level parallelism while being extremely fast (3.35 nanoseconds per iteration).

How about the difference?

```
Control: 113.301934798227364

Assembly: 113.301934798227364

Difference: 0
```

There is no difference at all between this method and the control variable. This can indicate that the native method already uses SIMD instructions (not necessarily AVX, as it can use SSE instructions instead), which by itself means it is already optimized and is already fast. It boils down to what it does under the hood—the native C++ method has variable and other necessary checks, while the AVX method shown here does not.

IV. Conclusion

Calculating the square root is a relatively arduous task for the CPU. Everyone takes for granted how fast CPUs are to achieve such performance despite their shortcomings. With that said, to properly utilize the CPU's performance, the proper approach is required. Not only in how to approach the problem, but how to properly write and implement the algorithm to solve the problem in just the right amount of time and accuracy.

Clearly the best method highlighted here is the AVX method and the native C++ method. The rest are insufficient to carry out for real-time use, both in run-time and in accuracy. Despite its drawbacks, it's a great way to learn how algorithms work, how to speed up and optimize the performance of the algorithm, and the benefits and drawbacks of each algorithm.

With that said, each algorithm has its own ups and downs. While each algorithm achieves varying accuracy with varying run-times, each has their own purpose, and can be further optimized for a highly specific task, should they need be. Among the three algorithms (brute-force, Babylonian method, Bisection method), the Babylonian method is the preferable alternative than the rest, as it provides the right compromise/balance between performance and accuracy. With proper parameters and optimizations, it can achieve better accuracy than the rest while performing faster.

It is recommended to use the built-in *sqrt()* function in C++ in all cases, as the function is already optimized enough for quick computing. Using the AVX method is also advisable, but then the question becomes "is it necessary," rather than "which one is faster," considering how fast the built-in C++ function already is.

There is a phenomenon known as premature optimization, where developers will try their best to avoid using already available tools at their disposal, but rather implement things themselves, believing that they know better than the compiler, which already optimizes certain parts of the code mid-compiling, and thus be able to write "better" and "faster" code.

This is outside the scope of this paper, but in conclusion, it's better to stick with the built-in method rather than implementing it with the goal to achieve better performance, as the time spent on writing such algorithm can be used for other important aspects of the program.

With that said, not all premature optimization is bad. Using the AVX approach is not bad by any means, but it just means that it offloads the difficulty of the code to the developer (such as checking whether the input variable is in the domain) instead of using already available code via intrinsics.

It is possible to implement the AVX method without writing it in Assembly by using compiler intrinsics. Compiler intrinsics are functions that the compiler implements directly to the program, rather than linking to a library containing the implementation. Using intrinsics, it is possible to explicitly utilize AVX instructions without the need to write Assembly and linking the function to C++.

The drawback of the AVX approach is compatibility, but that drawback is slowly diminishing as time flies. AVX compatibility was introduced into new CPUs since 2011 in Intel's Sandy Bridge lineup and AMD's Bulldozer lineup of processors. Considering that was 12 years ago, compatibility is no longer an issue.

Source Code Link at GitHub

Source code for this program used for this paper can be found ***here***, which contains all the algorithms used for this paper. The program is written in C++ with Assembly for the AVX approach.

Acknowledgment

This paper would not have been brought to fruition without the resources provided by the IF2211 Algorithm Strategies class with the guidance of Dr. Nur Ulfa Maulidevi, S. T, M.Sc. as the lecturer of the author in class.

The author would also like to thank past researchers and other programmers on the Internet that have provided tutorials and documentation. Without it, the author would not have been able to learn and make a program for this paper.

References

[1] Burden, Richard L.; Faires, J. Douglas (1985), "2.1 The Bisection Algorithm", Numerical Analysis (3rd ed.), PWS Publishers, ISBN 0-87150-857-5

[2] Khalasi, Bhargav (2020), https://www.codesdope.com/blog/article/find-square-root-of-a-number/ [Accessed: May 22, 2023],

[3] Fowler, David; Robson, Eleanor (1998), "Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context". *Historia Mathematica*. 25 (4): 376

[4] Lahanas, Michael, https://www.hellenicaworld.com/Greece/Science/en/HeronsMath.html [Accessed: May 22, 2023].

[5] Verma, Priyank (2014), https://priyankvex.com/2014/04/01/newton-raphson-vs-bisection-search-vs-brute-force-for-finding-square-root-of-number/comment-page-1/ [Accessed: May 22, 2023].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023

[illegible]

Ammar Rasyad Chaeroel 13521136