Why is float division slow?

Asked 15 years, 10 months ago Modified 6 years, 5 months ago

Viewed 12k times

23

What are the steps in the algorithm to do floating point division?

Why is the result slower than say, multiplication?

Is it done the same way we do division by hand? By repeatedly dividing by the divisor, subtracting the result to obtain a remainder, aligning the number again and continuing till the remainder is less than a particular value?

Also, why do we gain on performance if instead of doing

```
a = b / c
```

we do

```
d = 1 / c
a = b * d
```

?

Edit: Basically I was asking because someone asked me to distribute a value among contenders based on the assignment of weights. I did all this in integers and was

later asked to convert to float, which caused a slowdown in performance. I was just interested in knowing how would C or C++ do these operations that would cause the slowness.

performance algorithm hardware

Share

Improve this question

Follow

edited Oct 17, 2009 at 13:27

Peter Mortensen

31.6k • 22 • 109 • 133

asked Feb 3, 2009 at 7:16

umar

4,389 • 9 • 35 • 48

related: Floating point division vs floating point multiplication for some more detailed numbers for modern x86 – Peter Cordes Jul 29, 2020 at 16:00

6 Answers

Sorted by: Highest score (default)

25

FPU division often basically uses Newton-Raphson (or some other algorithm) to get a reciprocal then multiplies by that reciprocal. That's why the reciprocal operation is slightly faster than the general division operation.

This HP paper (which is actually more understandable than most papers I come across talking about Newton-Raphson) has this to say about floating point division:

> Floating point division and square root take considerably longer to compute than addition and multiplication. The latter two are computed directly while the former are usually computed with an iterative algorithm. The most common approach is to use a division-free Newton-Raphson iteration to get an approximation to the reciprocal of the denominator (division) or the reciprocal square root, and then multiply by the numerator (division) or input argument (square root).

Share Improve this answer Follow

edited Feb 3, 2009 at 8:37

answered Feb 3, 2009 at 8:26

Michael Burr

340k ● 52 ● 548 ● 768

I'm not saying the HP paper is wrong, I don't know enough to even consider that. However, the paper was last revised in 1994, so I'd guess process design has changed at least *some* in 14 years :-) So take the paper w/a grain of salt. – Walden Leverich Mar 17, 2009 at 14:45

1 @WaldenL Process design might change (miniaturization) however arithmetics hardly change. – Dimitar Slavchev Aug 16, 2012 at 14:01

19

From a hardware point of view division is a iterative algorithm, and the time it takes is proportional to the number of bits. The fastest division that is currently around uses the radix4 algorithm which generates 4 bit of result per iteration. For a 32 bit divide you need 8 steps at least.

Multiplication can be done in parallel to a certain degree. Without going into detail you can break up a large multiplication into several smaller, independent ones. These multiplications can again be broken down until you're at a bit-level, or you stop earlier and use a small lookup-table in hardware. This makes the multiplication hardware heavy from a silicon real estate point of view but very fast as well. It's the classic size/speed tradeoff.

You need log2 steps to combine the parallel computed results, so a 32 bit multiply need 5 logical steps (if you go down to the minimum). Fortunately these 5 steps are a good deal simpler than the division steps (it's just additions). That means in practice multiplies are even faster.

Share Improve this answer

Follow

answered Feb 3, 2009 at 11:07

Nils Pipenbrinck

86.2k ● 33 ● 155 ● 223

As described in the Wikipedia article *Division algorithm*, there are two main aproaches to division in computers:

6

Slow Division

Uses the following recurrence and finds one digit per iteration:

```
partialRemainder[j+1] = radix * partialRemainder[j] - quotientDigit[n-(j+1)]*denominator
```

Fast Division

Starts with an estimation and converges on the quotient. How accurate you are depends on the number of iterations.

Newton-Raphson division (very briefly):

1. Calculate estimate of the reciprocal.
2. Compute more accurate estimates of the reciprocal.
3. Compute quotient by multiplying the dividend by the reciprocal.

Share Improve this answer Follow

edited Jul 6, 2018 at 18:35

answered Feb 3, 2009 at 7:35

John Mulder

10.1k ● 7 ● 34 ● 37

1

Think of the hardware involved, and you'll understand a lot better why it takes so much longer to divide than multiply. Both operations are done down at the Floating Point Unit (FPU) level, and even in the world of integral ALUs, the division circuit is a far busier place than a multiplication circuit. I would suspect this is only more painful in the world of floating point, as now the data isn't just least to most significant digit ordered, but is instead ordered by the IEEE 754 standard.

As for the round off, it's really about wherever the signals traveling between the gates get soldered to ground; where that happens, you lose digits. Not rounding, so much as truncation.

Or were you asking about simulating floating point arithmetic using just integers?

Share Improve this answer

Follow

answered Feb 3, 2009 at 7:33

Ed Carrel

4,244 ● 1 ● 26 ● 17

You won't gain performance by doing

```
d = 1 / c
a = b * d
```

You probably mean:

```
d = 1 / c
a1 = b1 * d
a2 = b2 * d
```

This way the division is done only once.

Division is per se slower than multiplication, however, I don't know the details. The basic reason is that, similar to functions such as sin or sqrt, it's just mathematically more complex. IIRC, a multiplication takes about 10 cycles on an average CPU, while a division takes about 50 or more.

How it is actually done was nicely explained by John Mulder.

edited Jul 12, 2014 at 13:44

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Feb 3, 2009 at 7:19

mafu

32.6k ● 43 ● 162 ● 251

I've always wondered why, though; one's adding in a loop and the other's subtracting. Maybe it's the extra test for

while(x>=y) in the loop, instead of a straight iteration of n... maybe it's figuring out the decimal representation of the final remainder as a fraction??? – L. Cornelius Dol Feb 3, 2009 at 7:33

3 In the case where "c" is a constant, you can indeed save time by multiplying by the reciprocal. – Nosredna Feb 3, 2009 at 8:14

@Software Monkey, multiplication in the ALU is done using shift-add, rather than adding n copies of the number being multiplied, so it only takes k iterations, where k is the number of bits in your number: en.wikipedia.org/wiki/Multiplication_ALU – Kothar Feb 3, 2009 at 11:40

0

Float division is not much slower than integer division, but the compiler may be unable to do the same optimizations.

For example the compiler can replace integer division between 3 with a multiplication and a binary shift. Also it can replace float division between 2.0 with a multipliation by 0.5 but it cannot replace division between 3.0 with a multiplication by 1/3.0 as 1/3.0 cannot be represented exactly using binary numbers, therefore rounding errors may change the result of the division.
As the compiler doesn't know how sensitive is your application to rounding errors (say you were doing a weather simulation, see Butterfly effect)it cannot do the optimization.

Share Improve this answer Follow

answered Feb 3, 2009 at 12:38

ggf31416
3,647 1 27 26