Fast Inverse Square Root

Bingyu Wang

September 15, 2017

The goal is calculate

\[ y = \frac{1}{\sqrt{x}} \]  \hspace{1cm} (1)

where the denominator is an Euclidean norm of a vector. Sum of square is fast enough to calculate, but the main problem is to calculate the inverse square root (see equation (1)).

**Single-precision floating-point format**

which contains three part: \( \text{sign} : \text{bit}(31); \text{exponent} : \text{bit}(23 – 30); \text{fraction} : \text{bit}(0 – 22) \). And its value can be written as:

\[ x = (-1)^{b_{31}} \times (1 + 0.b_{22}b_{21} \cdots b_0)_2 \times 2^{(b_{30}b_{29} \cdots b_{23})_2 – 127} \]  \hspace{1cm} (2)

\[ = (-1)^{b_{31}} \times (1 + f) \times 2^e \]  \hspace{1cm} (3)

\[ = (1 + f) \times 2^e \]  \hspace{1cm} (4)

since \( x \) is a norm, always positive and where:

\[ f = (0.b_{22}b_{21} \cdots b_0)_2 \]

\[ = (b_{22}b_{21} \cdots b_0)_2 \]

\[ = \frac{F}{L} \]  \hspace{1cm} (5)

where \( F \) transform the fraction into integer, and \( L \) is a constant(\( 2^{23} \)).

\[ e = (b_{30}b_{29} \cdots b_{23})_2 – 127 \]

\[ = E – B \]  \hspace{1cm} (6)

where \( E \) is the bits format for exponent 8-bits, and \( B \) is a constant(127).

**Floating-point format to Integer-Format** What if we transform the floating-point format (see in the figure) into integer bit using (5) and (6), which can be easily written as:

\[ \text{Integer}(x) = (b_{22}b_{21} \cdot b_0)_2 + (b_{30}b_{29} \cdot b_{23})_2 \times 2^{23} \]

\[ = F + EL \]  \hspace{1cm} (7)

\[ \text{https://en.wikipedia.org/wiki/Single-precision_floating-point_format} \]
**First Step Approximation**

Take a log based on 2 for equation (1):

\[
\log_2(y) = -\frac{1}{2} \log_2(x)
\]  

(9)

⇒ \log_2\left((1 + f_y) \times 2^{e_y}\right) = -\frac{1}{2} \log_2\left((1 + f_x) \times 2^{e_x}\right)

(10)

⇒ \log_2(1 + f_y) + e_y = -\frac{1}{2} \left[\log_2(1 + f_x) + e_x\right]

(11)

There is an approximation for \(\log_2(1 + x)\) if \(x \in [0, 1)\), which is \(x + \sigma\), where \(\sigma\) is pre-defined constant [see the following picture]

Therefore equation (11) can be further inferred:

\[
f_y + \sigma + e_y \approx -\frac{1}{2}(f_x + \sigma + e_x)
\]

(12)

\[
\Rightarrow \frac{F_y}{L} + \sigma + E_y - B \approx -\frac{1}{2}\left(\frac{F_x}{L} + \sigma + E_x - B\right) \text{ using (5), (6)}
\]

(13)

\[
\Rightarrow \frac{3}{2}L(\sigma - B) + F_y + E_yL \approx -\frac{1}{2}(F_x + E_xL)
\]

(14)

\[
\Rightarrow \frac{3}{2}L(\sigma - B) + \text{Integer}(y) \approx -\frac{1}{2}\text{Integer}(x) \text{ using (8)}
\]

(15)

\[
\Rightarrow \text{Integer}(y) \approx -\frac{1}{2}\text{Integer}(x) + \text{magic-number}
\]

(16)

where magic-number is \(-\frac{3}{2}L(\sigma - B)\).

In the algorithm, step

\[i *= \text{(long*)}ky;\]

is trying to transform floating into integer format and then (16) is corresponding to the algorithm step:

\[i = 0x5F3759DF - (i >> 1);\]

(17)

where \(i >> 1\) is divided by 2.

---

Second Step Approximation So far, the first step approximation already did a pretty good job, but there is a way we could improve it even further, which is using Newton method. (1) can be written as function of y:

\[ f(y) = \frac{1}{y^2} - x \]  \hfill (18)

easily to get the first derivation is:

\[ f'(y) = -\frac{2}{y^3} \]  \hfill (19)

According to Newton method

\[ y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} \]  \hfill (20)

\[ = y_n - \frac{1}{2}y_n - \frac{1}{2}xy_n^3 \]  \hfill (21)

\[ = \frac{3}{2}y_n - \frac{1}{2}xy_n^3 \]  \hfill (22)

where \( y_n \) is the first step approximation in equation (16), and \( x \) is the original input, which explains the last step in the algorithm:

\[ y_{\text{new}} = y_{\text{old}} \ast \left( \frac{3}{2} - \frac{x}{2} \ast y_{\text{old}}^2 \right); //\text{Newton iteration} \]