PIERRE_24

Geste de savoir

0x5f3759df

5 septembre 2021

Table des matières

Introduction

L'histoire de la constante `0x5f3759df` (et de la fonction `Q_rsqrt`) est relativement célèbre, mais pour l'intérêt pédagogique qu'elle représente, je me permets de vous la raconter à nouveau. Pour la faire courte, il s'agit d'une fonction qu'on retrouve entre autres dans le code source de Quake III (qui en est probablement la plus célèbre itération), même si sont histoire est probablement plus ancienne que ça.

Si vous ne la connaissez pas encore, restez par ici, c'est sympa : même si ce n'est plus rentable aujourd'hui, on va un peu explorer ce qui a derrière, et jouer avec le compilateur (et des GPUs !) pour bien comprendre tout ça! 🍹

> *i* Ce billet requiert, pour être compris, d'avoir vu la notion de logarithme, et d'avoir une vague idée de comment fonctionne l'assembleur. Sur cette deuxième partie, je ne garanti pas l'exactitude de toutes les informations, donc n'hésitez pas à me corriger en commentaire 🍊

1. Comment ?

Le code de cette célèbre fonction, telle qu'on la retrouve dans le code source de Quake III , est le suivant (avec les commentaires originaux, mais sans le `isnan`, qui est la version qu'on voit généralement sur internet) :

```
1  float Q_rsqrt( float number )
2  {
3          long i; /* NOTE: il vaut mieux utiliser des `int32_t`
                      aujourd'hui (voir ci-dessous) */
4          float x2, y
5          const float threehalfs = 1.5F;
6
7          x2 = number * 0.5F;
8          y  = number;
9          i  = * ( long * ) &y;                       // evil
                      floating point bit level hacking
10         i  = 0x5f3759df - ( i >> 1 );               // what the
                      fuck?
11         y  = * ( float * ) &i;
12         y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st
                      iteration
13 //        y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd
       iteration, this can be removed
14
15         return y;
16 }
```

Bien que ça ne soit pas évident pour le moment, ce code permet de calculer y comme une bonne approximation de la fonction inverse d'une racine carrée, $y \approx \frac{1}{\sqrt{x}}$, ou $y \approx x^{-\frac{1}{2}}$ (même chose écrit autrement).

Il s'agit d'une opération qu'on retrouve assez couramment dans un moteur 3D. En effet, celui-ci utilise des vecteurs normés, c'est à dire qu'on divise ceux-ci par leur "taille". Autrement dit, à partir d'un vecteur $\vec{v}$, le vecteur $\vec{v}'$ est obtenu via

$$\vec{v}' = \frac{\vec{v}}{\sqrt{v_x^2 + v_y^2 + v_z^2}} = \vec{v} \times \frac{1}{\sqrt{|\vec{v}|^2}},$$

où on voit très clairement apparaître l'inverse de la racine carrée d'un nombre, en l'occurrence le carré de la norme du vecteur, $|\vec{v}|^2$.

L'astuce qui est utilisée ici tient en deux étapes. La première concerne les lignes 8 à 11, et consiste à exploiter la représentation IEEE 754. La seconde étape consiste en la ligne 12 (et 13, mais elle a été commentée), et est une amélioration de la valeur obtenue dans la première étape.

À l'époque ou ce code était utilisé (milieux des années 90), les processeurs étaient évidemment plus lents qu'aujourd'hui, et en plus de ça, le matériel dédié à la 3D était encore balbutiant (la première carte graphique célèbre intégrant de telles fonctions est la Voodoo de 3dfx sort fin 96). Pour ajouter à l'intérêt de cette fonction, travailler avec des nombres flottant était à l'époque beaucoup plus lent que de travailler sur des entiers, et la division était **particulièrement** inefficace (notez que ce code n'en contient pas). Tout ceci fait qu'à l'époque, une telle astuce permettait d'accélérer nettement le calcul pour une perte de précision mineure ($<1\%$).

1.1. Première étape : *evil floating point bit level hacking*, puis *what the fuck ?*

1.1.1. "I triple E sept-cent-cinquante-quatre" !

Comme expliqué par @Aabu dans son tutoriel , la représentation des nombres réel en binaire suis (généralement) le standard IEEE 754 . [1]footnote:1 Celui-ci est stocké dans une unique série de bits en 3 parties distinctes (pour un nombre normalisé) :

— $\mathcal{S}$, le bit de signe, qui vaut zéro (positif) ou 1 (négatif). Par la suite, je vais me permettre de noter $|\mathcal{S}| = 1$, où $|\mathcal{S}|$ est la taille de $\mathcal{S}$, qui vaut ici un bit.
— $\mathcal{E}$, l'exposant. En fait, $\mathcal{E}$ est stocké sous la forme d'un nombre entre 0 et $2^{|\mathcal{E}|} - 1$ (valeur maximale qu'on peut stocker dans un nombre contenant $|\mathcal{E}|$ bits) et l'exposant utilisé en pratique dans le réel est calculé comme $\mathcal{E} - B$, où $B = 2^{|\mathcal{E}|-1}$ est le biais, pour obtenir un exposant compris entre $-2^{|\mathcal{E}|-1}$ et $2^{|\mathcal{E}|-1}$.
— $\mathcal{M}$, la mantisse, avec $|\mathcal{M}| > |\mathcal{E}|$. Encore une fois, en pratique, $\mathcal{M}$ est stocké sous la forme d'un nombre compris entre 0 et $2^{|\mathcal{M}|} - 1$, tandis que dans la représentation IEEE 754, il est divisé par $L = 2^{|\mathcal{M}|}$ pour donner un nombre entre 0 et 1.

On peut écrire le réel x selon le standard IEEE 754 comme $x = \boxed{\mathcal{S}}\boxed{\mathcal{E}}\boxed{\mathcal{M}}$, où $\mathcal{S}$, $\mathcal{E}$ et $\mathcal{M}$ sont les valeurs numériques des 3 parties du nombre flottant. On en connaît la valeur, vu ce qu'on a dit plus haut, grâce à la formule suivante :

$$x = (-1)^{\mathcal{S}} \times 2^{(\mathcal{E}-B)} \times \left(1 + \frac{\mathcal{M}}{L}\right).$$

On peut également (ça nous servira pour la suite) calculer la valeur qu'aurait ce nombre si ces bits étaient interprétés comme un entier avec

$$x_I = \mathcal{M} + L \times (\mathcal{E} + 2^{|\mathcal{E}|} \times \mathcal{S})$$

1.1.2. `float` ?

Ici, on utilise plus particulièrement le `float`, soit, en C, un nombre à virgule défini sur 32 bits. Les 3 parties sont arrangées comme suit :

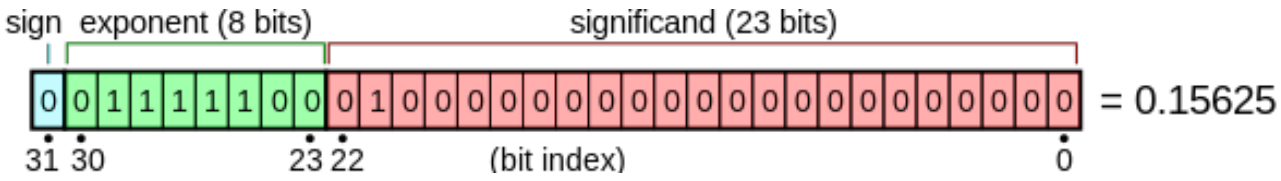

FIGURE 1.1. – Représentation du nombre réel 0,15625.

1. [2]footnote:1 Et à ce sujet, n'oubliez pas de lire *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, http://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf

On a donc $|\mathcal{M}| = 23$, $|\mathcal{E}| = 8$ et dès lors $B = 2^{8-1} = 127$ et $L = 2^{23}$. On peut également vérifier que notre formule fonctionne sur l'exemple donné ci-dessus, puisque $x = \boxed{0_2}\boxed{01111100_2}\boxed{01000000000000000000000_2} = \boxed{0}\boxed{124}\boxed{2097152}$ (notez l'indice 2, qui indique qu'il s'agit de binaire), donc

$$x = (-1)^0 \times 2^{(124-127)} \times \left(1 + \frac{2097152}{2^{23}}\right),$$

ce qui fait bien 0,15625. Par ailleurs, on peut calculer que $01111100010000000000000000000000_2 = 1042284544$, ce qui est bien le résultat du calcul suivant :

$$x_I = 2097152 + 2^{23} \times (124 + 2^8 \times 0).$$

1.1.3. C'est bien gentil, tout ça, mais ou est l'astuce ?

Un peu d'histoire avec des logarithmes dans le dedans

Avant d'expliquer le détail du code, commençons par faire un pas de côté. Il est un temps pas si lointain ou la calculatrice était hors de prix ou n'existait pas. Nos ancêtres (même si certains vivent encore aujourd'hui) utilisaient alors des tables de logarithmes (ou des règles à calcul, selon la précision demandée). Trois propriétés sont alors intéressantes à exploiter :

1. $\log(a \times b) = \log(a) + \log(b)$: autrement dit, pour calculer la multiplication de deux nombres, il suffit d'additionner la valeur de leurs logarithmes, puis d'utiliser une table de logarithme inverse, autrement dit, si on utilise le logarithme en base 10, calculer 10^c, avec $c = \log_{10}(a) + \log_{10}(b)$. Seule limitation, a et b doivent être positif (il n'existe pas de logarithme d'un nombre négatif). Notez également que ça fonctionne pour n'importe quelle base de logarithme, même si les plus courantes sont la base 10 et la base e.
2. $log(a^n) = n \times log(a)$: autrement dit, pour calculer la puissance n (qui peut très bien être réel et/ou négatif) d'un nombre a (lui strictement positif), on multiplie la valeur de son logarithme par n, puis on utilise encore une fois une table de logarithme inverse.
3. Des deux propriétés précédentes, on peut déduire $log\left(\frac{a}{b}\right) = \log(a) - \log(b)$.

Autrement dit, plutôt que de calculer $y = x^{-\frac{1}{2}}$, on pourrait calculer (j'ai utilisé la seconde propriété),

$$\log_2(y) = -\frac{1}{2}\log_2(x).$$

Notez que vu qu'on est dans le contexte de l'informatique, on utilise des logarithmes en base 2, puisqu'on manipule des bits.

Bon, ça parait pas très malin, puisque le logarithme semble être une opération à priori aussi complexe qu'une racine carrée 🤦 Sauf que x et y sont tout deux des nombres réels dont on part du principe qu'il s'agit de deux `float` représentés selon la norme IEEE 754.

Dès lors, vu que $x = \boxed{\mathcal{S}_x}\boxed{\mathcal{E}_x}\boxed{\mathcal{M}_x}$ et $y = \boxed{\mathcal{S}_y}\boxed{\mathcal{E}_y}\boxed{\mathcal{M}_y}$,

$$\log_2(y) = -\frac{1}{2}\log_2(x) \quad \Leftrightarrow \quad \log_2\left[(-1)^{\mathcal{S}_y} \times 2^{(\mathcal{E}_y - B)} \times \left(1 + \frac{\mathcal{M}_y}{L}\right)\right] = -\frac{1}{2}\log_2\left[(-1)^{\mathcal{S}_x} \times 2^{(\mathcal{E}_x - B)} \times \left(1 + \frac{\mathcal{M}}{L}\right.\right.$$

Bien entendu, on ne peut calculer la racine carré que d'un nombre positif, donc on peut partir du principe que $\mathcal{S} = 0$, ce qui simplifie :

$$\log_2\left[2^{(\mathcal{E}_y - B)} \times \left(1 + \frac{\mathcal{M}_y}{L}\right)\right] = -\frac{1}{2}\log_2\left[2^{(\mathcal{E}_x - B)} \times \left(1 + \frac{\mathcal{M}_x}{L}\right)\right].$$

En utilisant la première propriété, on peut séparer le produit en une somme de logarithmes, et

$$(\mathcal{E}_y - B) + \log_2\left(1 + \frac{\mathcal{M}_y}{L}\right) = -\frac{1}{2}\left[(\mathcal{E}_x - B) + \log_2\left(1 + \frac{\mathcal{M}_x}{L}\right)\right].$$

À ce niveau, il semblerait qu'on soit coincé. En effet, il n'existe pas de simplification exacte pour le logarithme d'une somme. Par contre, on peut approximer ce logarithme en utilisant une série de Taylor . On va choisir la forme suivante :

$$\log_2\left(1 + \frac{\mathcal{M}}{L}\right) \approx \frac{\mathcal{M}}{L} + \sigma,$$

où σ est une constante, dont la valeur peut être choisie. En effet, puisque $\mathcal{M} < L$, $\frac{\mathcal{M}}{L}$ est compris entre 0 et 1. On va donc choisir cette constante afin de minimiser la valeur du logarithme et notre approximation sur cet intervalle :

$$\min_{\sigma}\left\{\int_0^L d\mathcal{M}\,\left|\log_2\left(1 + \frac{\mathcal{M}}{L}\right) - \left(\frac{\mathcal{M}}{L} + \sigma\right)\right|\right\}$$

On peut montrer que la valeur qui fonctionne bien pour ça est $\sigma = 0.0430357$, mais comme on le verra, il ne s'agit que d'un détail. Graphiquement, on obtient ceci

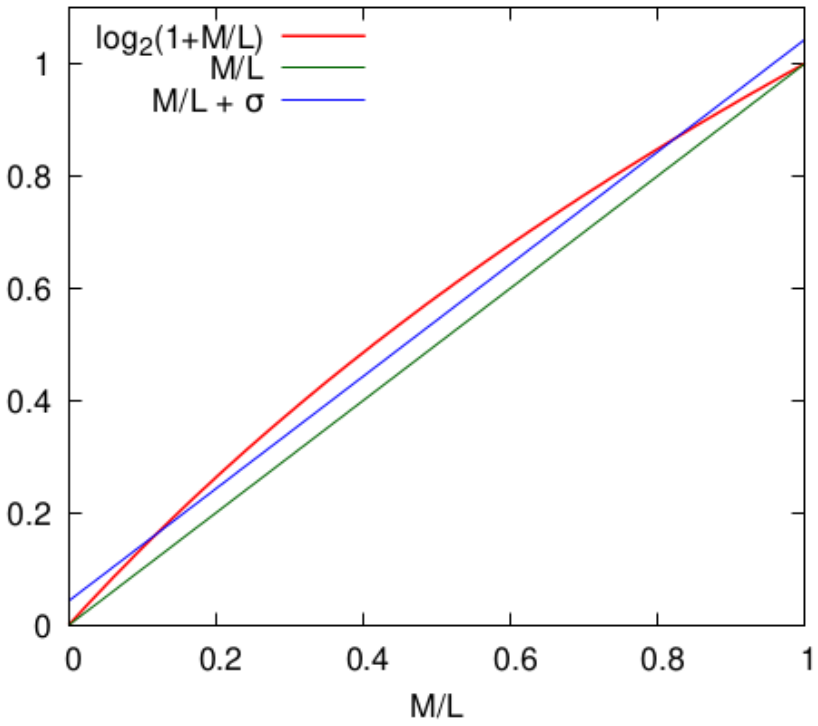

FIGURE 1.2. – Représentation de l'évolution du logarithme (rouge) et de ces approximations ($\sigma = 0$ en vert, σ optimal en bleu).

Autrement dit, le choix de σ permet de passer d'une approximation (courbe verte, $\sigma = 0$) exacte à $\mathcal{M} = 0$ et $\mathcal{M} = L$, mais moins bonne aux alentours de $\mathcal{M} = \frac{L}{2}$ à une approximation (σ idéal) qui donne, globalement, des résultats plus proches de la réalité.

On peut donc maintenant écrire

$$
\begin{aligned}
&(\mathcal{E}_y - B) + \log_2\left(1 + \frac{\mathcal{M}_y}{L}\right) = -\frac{1}{2}\left[(\mathcal{E}_x - B) + \log_2\left(1 + \frac{\mathcal{M}_x}{L}\right)\right] \\
&\quad \Leftrightarrow (\mathcal{E}_y - B) + \frac{\mathcal{M}_y}{L} + \sigma \approx -\frac{1}{2}\left[(\mathcal{E}_x - B) + \frac{\mathcal{M}_x}{L} + \sigma\right] && \text{(par l'approximation ci-dessus)} \\
&\quad \Leftrightarrow \mathcal{E}_y + \frac{\mathcal{M}_y}{L} \approx -\frac{1}{2}\left[(\mathcal{E}_x - B) + \frac{\mathcal{M}_x}{L} + \sigma\right] - \sigma + B && \text{(on fait passer à droite)} \\
&\quad \Leftrightarrow \mathcal{E}_y + \frac{\mathcal{M}_y}{L} \approx -\frac{1}{2}\left[\frac{\mathcal{M}_x}{L} + \mathcal{E}_x\right] - \frac{3}{2}\left[\sigma - B\right] && \text{(on réarange)} \\
&\quad \Leftrightarrow \mathcal{M}_y + L\,\mathcal{E}_y \approx -\frac{1}{2}\left[\mathcal{M}_x + L\,\mathcal{E}_x\right] - \frac{3L}{2}\left[\sigma - B\right] && \text{(on multiplie par L)}
\end{aligned}
$$

Et là, on a réussi à faire apparaître les représentations entières de x et y (avec le bit de signe à 0)! Dès lors, on peut finalement simplifier et obtenir

$$
y_I \approx \underbrace{\frac{3L}{2}\left[B - \sigma\right]}_{\text{constante magique}} - \frac{1}{2}\,x_I
$$

Et ce calcul, c'est littéralement la ligne 10 du code de la fonction `Q_rsqrt` (n'oubliez pas que décaler les bits vers la droite d'un entier le divise par 2, `4 > 1` est égal à `2`). C'est beau, non? Quant au *evil floating point bit level hacking*, il s'agit de la manière d'obtenir la représentation entière d'un nombre flottant. En effet, `i = (long) y` n'aurait pas fonctionné (C aurait uniquement renvoyé la partie entière de `y`, alors qu'on veut lui faire croire qu'il s'agit d'un entier). Il faut donc forcer la main à C, ce qui en développé donne ceci

```
1 /* Ce code est équivalent à la ligne 9 du code ci-dessus: */
2 long* tmp = NULL; // on crée un pointeur vers un `long`
3 tmp = (long*) &y; // on le fait pointer vers `y` (en faisant un
      cast)
4 i = *tmp;         // on donne à `i` la valeur de `tmp`
```

Si les pointeurs sont un peu loin pour vous, je vous invite à relire le chapitre correspondant de l'excellent tutoriel C de ZdS. Notez ceci dit que, bien que ça fonctionne, ce n'est pas une utilisation courante des pointeurs

On fait bien entendu la manipulation inverse à la ligne 11 pour obtenir la représentation flottante de `y`, qui est notre réponse.

30 secondes de chipotage

Cette astuce fonctionne encore aujourd'hui, comme on le verra plus bas.
En principe, le code était compilé et exécuté dans l'environnement le plus courant à l'époque: les processeurs 32 bits. Dès lors, la taille d'un `long` était de 4 octets (la même que la taille d'un `float`). Aujourd'hui, sur un processeur 64 bits, la taille est de 8 octets, ce qui peut poser des problèmes. On peut donc utiliser un `int32_t` pour régler ça, et c'est ce que je ferais dans la seconde section.
Par ailleurs, comme le fait très justement remarquer Wikipédia , il s'agit normalement d'un *undefined behavior* : le compilateur a le droit d'interpréter la ligne 9 de `Q_rsqrt` (ou la ligne 3 du code ci-dessus) comme bon lui semble. On peut éviter ce problème en utilisant une `union` (en C) ou `std::bit_cast<std::uint32_t>()` (en C++).

Pour en terminer avec cette partie du code, on a donc que $\frac{3L}{2}\ [B - \sigma]$, une fois réinterprété comme un entier, vaut à peu près `0x5f3759df` (1 597 463 007 en base 10), d'où notre constante magique. Je dis "à peu près", car la valeur est normalement 1 597 488 309,5740416 (qu'on peut calculer vu qu'on connaît les valeurs de B et σ pour nos flottants 32 bits), soit , si on en garde la valeur entière, `0x5f37bcb5`. En fait, la constante a très probablement été ajustée pour donner une meilleure précision après la deuxième étape

1.2. Second étape : *1st iteration*

On verra ci-dessous que l'approximation qu'on a obtenue tient la route, mais on peut encore l'améliorer sans faire trop de calculs supplémentaires grâce à la méthode de Newton . Cette méthode est en fait utilisée pour trouver numériquement les zéros d'une fonction [les solutions de l'équation $f(x) = 0$] par itérations successives. Pour ce faire, on approxime que la fonction

est linéaire, et que l'intersection de cette droite avec l'axe des x correspond à une bonne approximation du zéro. Puis on itère.

FIGURE 1.3. – Animation représentant la recherche d'un zéro par la méthode de Newton (source). On voit qu'on se rapproche de plus en plus de la valeur exacte.

Il s'agit donc d'un algorithme itératif, dont le point suivant est obtenu grâce à

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)},$$

où f' est la dérivée de f. Pour plus de détails mathématiques, je vous renvoie à ce tutoriel de @Holosmos.

Du coup, on va choisir une fonction telle que son zéro soit la valeur recherchée. Prenons

$$f(y) = \frac{1}{y^2} - x \Leftrightarrow f'(y) = -\frac{2}{y^3}$$

Grâce à cette formulation, si $f(y)$ est égal à 0, alors cela signifie que y est exactement l'inverse de la racine carrée de x. La formule itérative devient alors

$$y_{k+1} = y_k + \frac{y_k^3}{2}\left(\frac{1}{y_k^2} - x\right) = y_k\left(\frac{3}{2} - \frac{x}{2}y_k^2\right),$$

ce qui correspond bien à la ligne 12 (et 13) du code de `Q_rsqrt`. On voit que les auteurs se sont limités à une itération de la méthode de Newton, probablement pour des raisons de coup/performance. Le tour est joué 🍊

1.3. Généralisation

Évidement, c'est généralisable à tout calcul du type $y = x^{\frac{1}{p}}$ pour tout p différent de zéro. En résumé, on obtient, pour la première étape

$$y_I \approx \underbrace{\left(1 - \frac{1}{p}\right) L\,[B - \sigma]}_{\text{constante magique}} + \frac{1}{p} x_I,$$

et pour la seconde,

$$y_{k+1} = y_k \left(\frac{p-1}{p} + \frac{x}{p} \times \frac{1}{y^p} \right),$$

sauf que vu que le but était d'éviter les divisions, ça n'est intéressant que pour $p < 0$ (qui évite la division par y^p). On remarque également que la constante magique est toujours un multiple de $L\,(B - \sigma)$, c'est à dire pour un `float` environ 1 064 992 206, soit `0x3f7a7dce` (encore une fois, on peut ensuite l'ajuster).

2. Et ça marche ?

TL ;DR : oui, mais ce n'est plus rentable.

2.1. Précision

Premièrement, intéressons-nous à la précision, pour voir si ça vaut (valait ?) le coup :

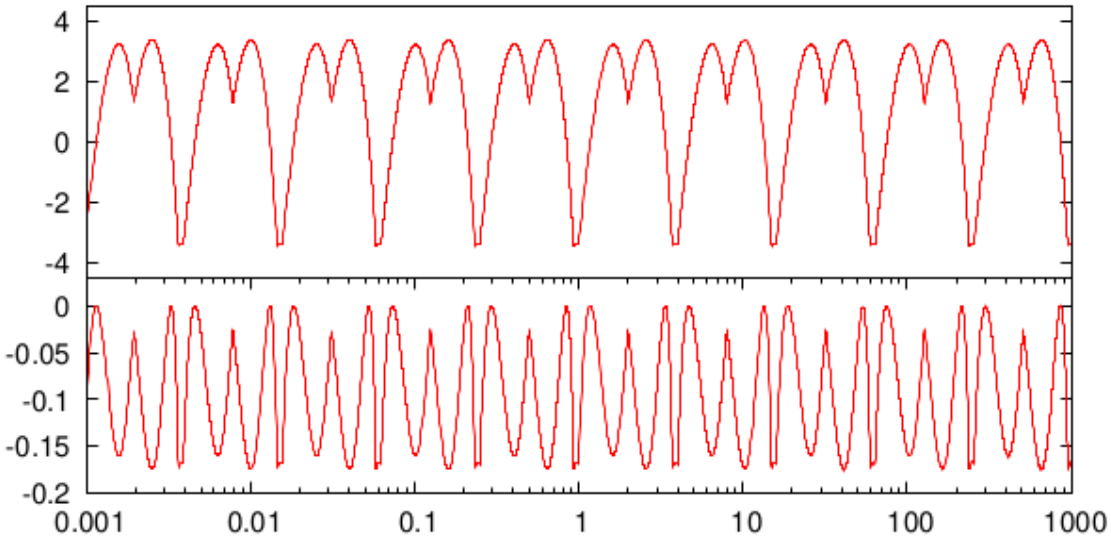

FIGURE 2.4. – Évolution de la précision (pourcentage d'erreur relative, axe des y) en fonction de l'entrée (axe des x), évaluée grâce à ce code , avec (en bas) ou sans (en haut) l'itération de l'algorithme de Newton.

Deux choses sont très clairement visibles :

1. si on se restreint à la première étape, l'erreur relative évolue périodiquement [3]footnote:1 entre environ -3 et 3%, et
2. si on ajoute une itération de l'algorithme de Newton, on améliore l'erreur relative d'un facteur, puisqu'elle est cette fois comprise entre 0 et -0,2%. [5]footnote:2

1. [4]footnote:1 La périodicité s'explique probablement très bien, mais je n'y ai pas réfléchi.
2. [6]footnote:2 Ce qui signifie qu'on sous-estime systématiquement la valeur 🍊

Ça peut paraître encore beaucoup [7]footnote:3, mais c'est amplement suffisant pour l'époque 🍊 En effet, les moteurs de rendus 3D privilégiait (et c'est encore le cas) la quantité à la qualité. [9]footnote:4 Les données y sont d'ailleurs encore représentés sous la forme de `float` au lieu de `double`.

2.2. Performances

2.2.1. Jouons avec le compilateur

Pour tester les performances, j'ai écrit un second code qui mesure les performances respectives de `rsqrt` et `Q_rsqrt` sur un même tableau de 1 000 000 de `float` aléatoires, à l'aide d'une boucle `for`. Le code n'est en soi pas très compliqué si vous connaissez le C, à part peut être le code de la fonction `rsqrt` :

```c
float rsqrt(float number) {
    float res;
#ifdef USE_rsqrtss
    asm ("rsqrtss %1, %0" : "=x" (res) : "xm" (number));
#else
    res = 1.f / sqrtf(number);
#endif
    return res;
}
```

En effet, je vais comparer deux versions : celle, purement en C, où on calcule l'inverse de la racine carrée et celle, en assembleur, où on utilise l'instruction `rsqrtss` , qui est l'instruction processeur (en x86_64) pour calculer l'inverse d'une racine carrée.

> ! Ce test n'est pas uniquement un test de performance. En effet, il y a également une pénalité d'accès la mémoire qui entache la performance des deux approches (mais qui devrait être similaire pour les deux fonctions).

Je vais effectuer ces tests sur mon AMD Ryzen 5 1500x (ordinateur personnel, acheté en 2018 [11]footnote:5). Le programme est lancé 1000 fois et une moyenne est ensuite calculée :

Commande	`Q_rsqrt`	`rsqrt()`	`rsqrt()` avec `-DUSE_rsqrtss`
`gcc -lm -o test perf.c -O0 && ./test`	8.08 ns/float	5.69 ns/float	4.22 ns/float

3. [8]footnote:3 On peut encore réduire l'erreur en ajoutant des itérations de Newton: d'après Wikipédia, 3 suffisent amplement!
4. [10]footnote:4 Qu'on se comprenne bien : c'est "beau", mais on se permet bon nombre d'approximations pour que ça tourne "bien".
5. [12]footnote:5 … Et remplacé quelques mois après à cause d'un horrible problème hardware .

`gcc -lm -o test perf.c -O3 && ./test`	1.40 ns/float	3.28 ns/float	2.42 ns/float
`gcc -lm -o test perf.c -Ofast && ./test`	1.44 ns/float	1.35 ns/float	2.40 ns/float
`gcc -lm -o test perf.c -Ofast -mavx && ./test`	1.38 ns/float	1.38 ns/float	2.41 ns/float

TABLE 2.2. – Comparaison des performances (en nanoseconde par nombre flottant traité) en fonction des différentes options de `gcc`.

On peut constater deux choses : `rsqrt()` est effectivement plus rapide, particulièrement si on empêche `gcc` d'optimiser le code avec `-O0`. Dans ce cas là, il est même plus intéréssant d'utiliser directement la fonction en assembleur. Par contre, `Q_rsqrt` devient étrangement plus intéressant avec `-O3`, puis `rsqrt` (sans assembleur) devient compétitif `-Ofast` (`-O3` et `-ffast-math`). Pour bien comprendre ce qui se passe, on peut s'amuser à regarder le code assembleur du programme avec `objdump`.

Pour `Q_rsqrt` avec `-O0`, `gcc` traduit bêtement le code source en assembleur, puis fait un `call` dans la boucle `for` de main.

> Un peu d'assembleur pour ceux que ça intéresse

Par contre, quand on lui permet d'optimiser (avec `-O3` ou `-Ofast`), `gcc` se permet plusieurs choses :

1. De simplifier un peu le code de la fonction, en retirant les constantes et les variables intermédiaires;
2. D'intégrer directement le code de ladite fonction dans la boucle `for`, afin de supprimer l'appel, et d'avoir à perdre du temps à gérer la *stack* (pile);
3. Mais surtout, d'utiliser des instructions SSE !

> La version optimisée, pour ceux que ça intéresse

C'est la même histoire pour `rsqrt` : avec `-Ofast`, `gcc` se permet de remplacer le code de la fonction par l'instruction `rsqrtps` , qui est la version SSE de `rsqrtss`. Il utilise par contre `sqrtss` (qui calcule la racine carrée, non son inverse) avec `-O3`.

> Le code assembleur, toujours.

Bref, si on laisse `gcc` faire son travail, il peut optimiser un maximum l'exécution, ici pour arriver à environ 1 nanoseconde par nombre flottant avec les instructions SSE. Ça démontre également qu'il est illusoire d'essayer d'être plus malin que le compilateur en écrivant de l'assembleur (qui n'est pas optimisé par `-Ofast`!).

2.2.2. SSE ?

SSE, pour *Streaming SIMD Extensions* (équivalent de 3DNow! , depuis abandonné, chez AMD), est un *set* d'instructions processeur un peu spéciales, qui permettent de réaliser des

opérations sur plusieurs nombres flottants en même temps. Comme le laisse penser le nom de la version AMD, cette extension était à l'époque pensée pour aider le calcul en virgule flottante, en particulier dans le contexte des applications en 3 dimensions.
En effet, le paradigme SIMD (*single instruction, multiple data*) signifie que la même instruction est appliquée à différentes données. Par exemple, soit les trois tableaux suivants

```
1 float a[N], b[N], c[N];
```

Si on souhaite réaliser l'addition des valeurs de `a` et `b` pour les stocker dans `c`, on écrirait le code suivant :

```
1 for(i=0; i<N; i++)
2   c[i] = a[i]+b[i];
```

ce qui, dans l'absolu, exécute `N` fois les instructions qui composent la boucle. Ici, une instruction SSE permet de réduire ce nombre d'exécution n fois, ou n est la largeur de l'unité vectorielle, c'est à dire le nombre de données que le processeur peut traiter en même temps. Par exemple, les premières instructions SSE permettait de traiter 128 bits à la fois, soit 4 `float` en même temps : le temps d'exécution de la boucle ci-dessus est donc réduit d'à peu près 4 fois avec une telle instruction. C'est d'ailleurs ce que `gcc` a fait dans le test ci-dessus, puisque toutes les instructions font partie du *set* original de SSE (les extensions suivantes ajoutent les `double`, et d'autres opérations qu'on retrouve dans des contextes bien précis, mais sans dévier de 128 bits). Bien entendu, les développeurs ne se sont pas arrêté là, et les extensions AVX2 et AVX512 proposent de traiter, respectivement, 256 et 512 bits à la fois (donc 8 et 16 `float` à la fois).

> *i* D'ailleurs, on peut forcer `gcc` à utiliser des instructions AVX avec `-mavx`, comme fait dans la dernière ligne du tableau ci-dessus. Et pour le laisser optimiser un maximum, on peut carrément utiliser `-march=native`. Ceci dit, ça n'est intéressant que dans le contexte du calcul intensif, puisqu'ici arrive déjà à quelque chose de potable avec `-Ofast`

2.2.3. Est ce qu'on peut faire encore mieux ?

TL ;DR : oui, mais pas spécialement ici.
Mon intérêt pour `Q_rsqrt` n'est pas anodin. En effet, je suis (re)tombé sur cette fonction alors que je navigait sur les internets à la recherche de ressources pour le calcul intensif, que j'ai eu l'occasion de pratiquer durant ma thèse de doctorat . Or, le nouveau truc, en calcul intensif, c'est les GPUs, dont les performances sont relativement intéressantes s'ils sont bien exploités. Preuve en est, le prochain supercalculateur *hexascale* européen, LUMI sera principalement composé de GPU (des AMD Radeon Insctinct).
Dans le cadre du consortium des équipement de calculs intensifs belge (fédération qui rassemble les clusters des 6 universités francophones du pays), j'ai pour le moment (aout 2021) accès à deux types de GPUs de la marque Nvidia [13]footnote:6 : une Tesla V100 , qui comme tout les produits de la gamme Tesla est un GPU expressément orienté pour le calcul intensif, et un

6. [14]footnote:6 Ceci n'est pas une pub : il se trouve qu'on croyait que LUMI allait être équipé en cartes NVidia et qu'on voulait prendre de l'avance. Grossière erreur

GPU GeForce RTX 2080 , qui est normalement prévue pour le grand public [15]footnote:7 mais qui envoi du pâté tout de même (pour le calcul simple précision, c'est le plus puissant des deux).
Le code de test (pour `rsqrt`), écrit en CUDA (une surcouche du C++ pour écrire du code pour les GPU Nvidia) est disponible ici . La partie importante est la suivante :

```
1 cudaMemcpy(d_comm, floats_source, N_FLOAT *sizeof(float),
      cudaMemcpyHostToDevice);
2
3 int blocksize = 512;
4 int nblock = N_FLOAT/blocksize + (N_FLOAT % blocksize > 0 ? 1: 0);
5 rsqrt_vec<<<nblock, blocksize>>>(d_comm, N_FLOAT);
6
7 cudaMemcpy(floats_dest, d_comm, N_FLOAT *sizeof(float),
      cudaMemcpyDeviceToHost);
```

La première et la dernière ligne expliquent à elles seules les futures performances : en effet, pour que le calcul sur GPU aie lieu, il faut que les données soient disponibles dans sa mémoire vive, et la communication est évidemment l'étape limitante (le problème est le même en 3D "classique"). Autrement dit, pour qu'un calcul GPU soit rentable, il faut communiquer très peu et exploiter à fond les données une fois qu'elles sont communiquées (effectuer plusieurs séries de calculs avec).
Pour faire des calculs, on écrit des *(compute) kernels* , c'est à dire des fonctions qui suivent grosso modo le paradigme SIMD en effectuant une même série d'opération sur une série de données. C'est donc le même principe que les instructions SSE/AVX qu'on a croisé plus haut, sauf que les GPUs portent ce principe à l'extrême, en pouvant traiter d'énormes quantité de données "en même temps", grâce à une quantité proprement démentielle de coeurs de calculs (5120 pour le Tesla, 4352 pour le 2080 RTX). Ceci dit, ce n'est pas magique : comme on le voit aux lignes 4 et 5, il faut dire au GPU comment organiser les calculs : ceux-ci sont arrangés en "blocs" indépendants de `blocksize` *thread* qui collaborent étroitement (avec de la mémoire partagée, et une synchronisation entre autres). L'idée, c'est évidemment que chaque bloc traite une partie des données, et que les blocs soient assignés automatiquement par le GPU à des coeurs comme bon lui semble, tant qu'il reste des blocs à exécuter. Cela signifie que pour optimiser le code, on peut jouer sur différents paramètres (ce que je n'ai pas pris le temps de faire ici).
Quant au *kernel*, c'est la fonction `rsqrt_vec()`, qui se borne à utiliser la fonction `rsqrt` que CUDA met à notre disposition.
Bref, voici les résultats. Encore une fois, le code a été exécuté 1000 fois, avec 100 000 000 nombres flottants générés (c'est 100x plus que précédemment, pour tenter de réduire l'effet de la communication), et je donne les moyennes. Notez que le temps reporté inclus la communication vers et depuis le GPU, pour comparer des choses similaires (ce qui nous intéresse, c'est le résultat).

GPU	Performance
Tesla V100 @ 877 Mhz	2.36 ns/float
GTX 2080 RTX @ 1545 Mhz	1.01 ns/float

7. [16]footnote:7 Et achetée avant la pénurie de 2021

TABLE 2.4. – Performance (en nanoseconde par nombre flottant traité) en fonction du GPU.
Comme annoncé, le gain de performance n'est pas fou, et c'est probablement la communication qui domine le tout. Encore une fois, ça commence à devenir sérieusement intéréssant lorsqu'on utilise des *kernels* plus complexes 🍊

CUDA … mais encore?

Sachez que si vous n'avez pas envie d'utiliser CUDA (par exemple parce que vous possédez une carte graphique d'une marque concurrente), il est possible d'utiliser d'autres méthodes plus "agnostiques". On peut citer deux alternatives : OpenCL (globalement équivalent à CUDA en termes de philosophie) et OpenACC, le second étant un équivalent de OpenMP, c'est à dire une méthode de programmation ou on indique (par des `#pragma`) les parties du code qu'on souhaitera que le compilateur parallélise (du mieux qu'il peut) au lieu d'écrire explicitement du code parallèle. À condition que le compilateur fasse correctement son boulot,[17] footnote:8 on peut alors assez facilement paralléliser un code existant.

h. [18] footnote:8 C'est toujours en développement, et GCC est un peu à la traine pour OpenACC, il semblerait.

Conclusion

Au regard des performances respectives de `Q_rsqrt` et de `rsqrt`, le gain de performance du premier est aujourd'hui quasiment inexistant, donc son intérêt devient limité (surtout vu la perte de précision).
Quand à l'utilisation du GPU, ce n'est rentable que si plusieurs opérations sont appliquées de suite une fois les données placées en mémoire … Ce qui est totalement le cas dans les moteurs 3D actuels, ou les *vertex* sont sytématiquement *offloadés* sur le GPU (via, par exemple, la fonction `glGenVertexArrays` et consoeurs) pour être exploité *ad nauseam*. Dès lors, ça rend l'astuce d'autant plus inutile dans le cadre ou la fonction `Q_rsqrt` est apparue 🍊

Sources et autres curiosités
Outre les liens dans le texte,

- Wikipédia, comme d'habitude.
- Pour la dérivation de la première étape, j'ai choisi de suivre cet excellent post de Christian Plesner Hansen.
- Cet article qui détaille un peu l'histoire derrière la recherche de l'auteur de ce code.

Les codes sources et images sont disponibles sur Github, sous licence MIT.

Contenu masqué

Contenu masqué n°1 :

Contenu masqué

Un peu d'assembleur pour ceux que ça intéresse

```
$ gcc -g -lm -o test perf.c -O0 && objdump -dS -M intel ./test

(...)

float Q_rsqrt(float number)
{
  401166:        55                              push   rbp
  401167:        48 89 e5                        mov    rbp,rsp
  40116a:        f3 0f 11 45 ec                  movss  DWORD PTR
      [rbp-0x14],xmm0
        int32_t i;
        float x2, y;
        const float threehalfs = 1.5F;
  40116f:        f3 0f 10 05 f5 0e 00            movss  xmm0,DWORD
      PTR [rip+0xef5]        # 40206c <__dso_handle+0x64>
  401176:        00
  401177:        f3 0f 11 45 fc                  movss  DWORD PTR
      [rbp-0x4],xmm0

        x2 = number * 0.5F;
  40117c:        f3 0f 10 4d ec                  movss  xmm1,DWORD
      PTR [rbp-0x14]
  401181:        f3 0f 10 05 e7 0e 00            movss  xmm0,DWORD
      PTR [rip+0xee7]        # 402070 <__dso_handle+0x68>
  401188:        00
  401189:        f3 0f 59 c1                     mulss  xmm0,xmm1
  40118d:        f3 0f 11 45 f8                  movss  DWORD PTR
      [rbp-0x8],xmm0
        y  = number;
  401192:        f3 0f 10 45 ec                  movss  xmm0,DWORD
      PTR [rbp-0x14]
  401197:        f3 0f 11 45 f0                  movss  DWORD PTR
      [rbp-0x10],xmm0
        i  = * ( int32_t * ) &y;
  40119c:        48 8d 45 f0                     lea    rax,[rbp-0x10]
  4011a0:        8b 00                           mov    eax,DWORD PTR
      [rax]
  4011a2:        89 45 f4                        mov    DWORD PTR
      [rbp-0xc],eax
        i  = 0x5f3759df - ( i >> 1 );
  4011a5:        8b 45 f4                        mov    eax,DWORD PTR
      [rbp-0xc]
  4011a8:        d1 f8                           sar    eax,1
  4011aa:        89 c2                           mov    edx,eax
  4011ac:        b8 df 59 37 5f                  mov    eax,0x5f3759df
  4011b1:        29 d0                           sub    eax,edx
```

Contenu masqué

```
36    4011b3:        89 45 f4                    mov    DWORD PTR
          [rbp-0xc],eax
37          y  = * ( float * ) &i;
38    4011b6:        48 8d 45 f4                 lea    rax,[rbp-0xc]
39    4011ba:        f3 0f 10 00                 movss  xmm0,DWORD
          PTR [rax]
40    4011be:        f3 0f 11 45 f0              movss  DWORD PTR
          [rbp-0x10],xmm0
41          y  = y * ( threehalfs - ( x2 * y * y ) );
42    4011c3:        f3 0f 10 45 f0              movss  xmm0,DWORD
          PTR [rbp-0x10]
43    4011c8:        0f 28 c8                    movaps xmm1,xmm0
44    4011cb:        f3 0f 59 4d f8              mulss  xmm1,DWORD
          PTR [rbp-0x8]
45    4011d0:        f3 0f 10 45 f0              movss  xmm0,DWORD
          PTR [rbp-0x10]
46    4011d5:        0f 28 d1                    movaps xmm2,xmm1
47    4011d8:        f3 0f 59 d0                 mulss  xmm2,xmm0
48    4011dc:        f3 0f 10 45 fc              movss  xmm0,DWORD
          PTR [rbp-0x4]
49    4011e1:        0f 28 c8                    movaps xmm1,xmm0
50    4011e4:        f3 0f 5c ca                 subss  xmm1,xmm2
51    4011e8:        f3 0f 10 45 f0              movss  xmm0,DWORD
          PTR [rbp-0x10]
52    4011ed:        f3 0f 59 c1                 mulss  xmm0,xmm1
53    4011f1:        f3 0f 11 45 f0              movss  DWORD PTR
          [rbp-0x10],xmm0
54
55          return y;
56    4011f6:        f3 0f 10 45 f0              movss  xmm0,DWORD
          PTR [rbp-0x10]
57  }
58
59  (...)
60
61          results_rsqrt[i] = rsqrt(the_floats[i]);
62    401319:        8b 45 f8                    mov    eax,DWORD PTR
          [rbp-0x8]
63    40131c:        48 98                       cdqe
64    40131e:        8b 84 85 60 e5 f9 ff        mov    eax,DWORD PTR
          [rbp+rax*4-0x61aa0]
65    401325:        66 0f 6e c0                 movd   xmm0,eax
66    401329:        e8 df fe ff ff              call   40120d <rsqrt>
67    40132e:        66 0f 7e c0                 movd   eax,xmm0
68    401332:        8b 55 f8                    mov    edx,DWORD PTR
          [rbp-0x8]
69    401335:        48 63 d2                    movsxd rdx,edx
70    401338:        89 84 95 60 b0 ed ff        mov    DWORD PTR
          [rbp+rdx*4-0x124fa0],eax
```

Retourner au texte.

Contenu masqué n°2 :
La version optimisée, pour ceux que ça intéresse

```
$ gcc -g -lm -o test perf.c -Ofast && objdump -dS -M intel ./test

(...)

    for(i=0; i < N_FLOAT; i++)
        results_qrsqrt[i] = Q_rsqrt(the_floats[i]);
  401100:        0f 28 44 04 10               movaps xmm0,XMMWORD
      PTR [rsp+rax*1+0x10]
        i  = 0x5f3759df - ( i >> 1 );
  401105:        66 0f 6f d5                  movdqa xmm2,xmm5
  401109:        66 0f 6f c8                  movdqa xmm1,xmm0
        y  = y * ( threehalfs - ( x2 * y * y ) );
  40110d:        0f 59 c4                     mulps  xmm0,xmm4
        i  = 0x5f3759df - ( i >> 1 );
  401110:        66 0f 72 e1 01               psrad  xmm1,0x1
  401115:        66 0f fa d1                  psubd  xmm2,xmm1
        y  = y * ( threehalfs - ( x2 * y * y ) );
  401119:        0f 28 ca                     movaps xmm1,xmm2
  40111c:        0f 59 ca                     mulps  xmm1,xmm2
  40111f:        0f 59 c1                     mulps  xmm0,xmm1
  401122:        0f 28 cb                     movaps xmm1,xmm3
  401125:        0f 5c c8                     subps  xmm1,xmm0
  401128:        0f 59 ca                     mulps  xmm1,xmm2
        results_qrsqrt[i] = Q_rsqrt(the_floats[i]);
  40112b:        0f 29 4c 05 00               movaps XMMWORD PTR
      [rbp+rax*1+0x0],xmm1
    for(i=0; i < N_FLOAT; i++)
  401130:        48 83 c0 10                  add    rax,0x10
  401134:        48 3d 80 1a 06 00            cmp    rax,0x61a80
  40113a:        75 c4                        jne    401100
      <main+0x80>
```

Retourner au texte.

Contenu masqué n°3 :
Le code assembleur, toujours.

```
$ gcc -g -lm -o test perf.c -O3 && objdump -dS -M intel ./test

(...)

    for(i=0; i < N_FLOAT; i++)
```

Contenu masqué

```
 6    40116a:        66 0f 1f 44 00 00            nop    WORD PTR
          [rax+rax*1+0x0]
 7          results_rsqrt[i] = rsqrt(the_floats[i]);
 8    401170:        f3 0f 10 44 1c 10            movss  xmm0,DWORD
          PTR [rsp+rbx*1+0x10]
 9      res = 1.f / sqrtf(number);
10    401176:        0f 2e c8                     ucomiss xmm1,xmm0
11    401179:        0f 87 2c 01 00 00            ja     4012ab
          <main+0x21b>
12    40117f:        f3 0f 51 c0                  sqrtss xmm0,xmm0
13    401183:        0f 28 d3                     movaps xmm2,xmm3
14    401186:        f3 0f 5e d0                  divss  xmm2,xmm0
15          results_rsqrt[i] = rsqrt(the_floats[i]);
16    40118a:        f3 0f 11 94 1c 10 35         movss  DWORD PTR
          [rsp+rbx*1+0xc3510],xmm2
17    401191:        0c 00
18      for(i=0; i < N_FLOAT; i++)
19    401193:        48 83 c3 04                  add    rbx,0x4
20    401197:        48 81 fb 80 1a 06 00         cmp    rbx,0x61a80
21    40119e:        75 d0                        jne    401170
          <main+0xe0>
22
23  $ gcc -g -lm -o test perf.c -Ofast && objdump -dS -M intel ./test
24
25  (...)
26
27      res = 1.f / sqrtf(number);
28    401160:        0f 28 74 04 10               movaps xmm6,XMMWORD
          PTR [rsp+rax*1+0x10]
29    401165:        0f 52 ce                     rsqrtps xmm1,xmm6
30    401168:        0f 28 c6                     movaps xmm0,xmm6
31    40116b:        0f 29 34 24                  movaps XMMWORD PTR
          [rsp],xmm6
32    40116f:        0f 59 c1                     mulps  xmm0,xmm1
33    401172:        0f 59 c1                     mulps  xmm0,xmm1
34    401175:        0f 59 ca                     mulps  xmm1,xmm2
35    401178:        0f 58 c3                     addps  xmm0,xmm3
36    40117b:        0f 59 c1                     mulps  xmm0,xmm1
37      for(i=0; i < N_FLOAT; i++)
38          results_rsqrt[i] = rsqrt(the_floats[i]);
39    40117e:        0f 29 84 04 10 35 0c         movaps XMMWORD PTR
          [rsp+rax*1+0xc3510],xmm0
40    401185:        00
41      for(i=0; i < N_FLOAT; i++)
42    401186:        48 83 c0 10                  add    rax,0x10
43    40118a:        48 3d 80 1a 06 00            cmp    rax,0x61a80
44    401190:        75 ce                        jne    401160
          <main+0xe0>
```

Retourner au texte.