Rapport de stage – un algorithme de racine carrée en précision augmentée

Olivier Marty
Encadrant : Mioara Joldes, MAC/LAAS-CNRS

2 juin – 11 juillet 2014

Synthèse

Le contexte général

Mon stage porte sur une bibliothèque de calcul flottant en précision augmentée, écrite pour les processeurs classiques (CPU) ainsi que les unités graphiques (GPU, i.e. graphics processing units) compatibles avec le langage de programmation CUDA, GPU qui sont de plus en plus utilisés dans le calcul scientifique haute performance. Cette bibliothèque basée sur les sommes non évaluées de nombres à virgule flottante a pour but de fournir des outils de calcul rapides et prouvés, en précision arbitraire fixée à la compilation, et avec comme cibles principales les GPU.

Si l'addition et la multiplication sur de telles données ont déjà été étudiées, l'inverse et la racine carrée manquaient à la littérature et à la bibliothèque.

Enfin, s'il existe déjà des bibliothèques de précision arbitraire, telles que GNU MPFR, elles ne sont pas dédiées au GPU et leur plus grande souplesse entraîne un fonctionnement plus lent.

Le problème étudié

La somme, la multiplication, la division et la racine carrée formant une base minimale de l'arithmétique, le but de mon stage était de concevoir, de prouver et d'implémenter un algorithme pour calculer la racine carrée de nombres à précision augmentée manipulés par la bibliothèque. Pour cela je pouvais m'inspirer d'un travail de ma maître de stage sur un algorithme de division rapide et dont le résultat est prouvé.

La contribution proposée

La méthode utilisée pour la division, utilisant les suites de Newton-Raphson, devait convenir pour le calcul de la racine carrée. Parmi les deux suites candidates pour ce calcul, l'une nécessite une division, coûteuse. Je me suis donc intéressé d'abord à l'autre : j'ai écrit la preuve de convergence et après avoir lu – et récrit pour qu'il soit plus proche de la preuve – le code de la division j'ai implémenté ma solution dans la bibliothèque.

La preuve et l'implémentation s'adaptaient facilement à l'autre suite candidate, avec laquelle le calcul s'est effectivement avéré plus lent.

Cette façon de calculer n'est pas nouvelle, mais le plus important était de donner une preuve formelle de l'algorithme sur la borne d'erreur, c'est-à-dire le nombre de bits de confiance dans le résultat.

Les arguments en faveur de sa validité

L'itération de Newton-Raphson est une solution très efficace pour ce problème car sa convergence est quadratique, c'est-à-dire que la précision de l'approximation double à chaque étape. De plus une bonne approximation pour le premier terme de la suite est facilement obtenue en utilisant les opérations arithmétiques natives (disponibles en machine) sur les nombres à virgule flottante.

Ensuite nous avons effectué des tests pour comparer les bibliothèques existantes (sur CPU) qui ont montré que l'algorithme est rapide et correct.

La plupart des hypothèses de travail sont vérifiées pour les nombres manipulés par la bibliothèque, excepté celle de ne travailler qu'avec des nombres à virgule flottante dits *normaux* (voir plus loin), mais les nombres restants qui nécessiteraient une étude à part sont des cas très spéciaux.

Le bilan et les perspectives

Mon algorithme fonctionne et est rapide. Il est d'ores et déjà intégré à la bibliothèque qui peut être téléchargée librement ici (section Campary).

Malheureusement, cette méthode ne peut fonctionner pour beaucoup de fonctions mathématiques : il faut que l'itération de Newton qui en découle soit facilement calculable. Ce n'est pas le cas par exemple pour les fonctions trigonométriques, dont les dérivées sont elles-même des fonctions trigonométriques.

Afin de fournir encore la bibliothèque, on pourrait s'attaquer au calcul de fonctions trigonométriques sur des sommes non évaluées de nombres à virgule flottante avec, par exemple, la méthode CORDIC. Enfin il reste à prouver certains algorithmes qui ne le sont pas encore afin de pouvoir offrir une bibliothèque robuste.

Le laboratoire LAAS-CNRS – laboratoire d'analyse et d'architecture des systèmes – de Toulouse m'a d'abord attiré par le nombre de ses recherches qui gravitent autour du monde de la robotique, cependant la proposition de stage portant sur une bibliothèque de calcul en précision augmentée m'a tout de suite séduit par sa dimension utile : écrire une bibliothèque et la distribuer pourrait servir directement à d'autres. De plus ma mission comporterait plusieurs axes : algorithmique, preuve formelle et implémentation dans la bibliothèque.

1 Motivations

Le calcul avec un ordinateur se fait souvent avec des nombres à virgule flottante, que nous allons présenter, mais ceux-ci ont leurs limites. L'une de ces limites, la précision, a conduit à la création de notre bibliothèque de calcul.

1.1 Les nombres à virgule flottante

Les nombres à virgule flottante sont une manière de représenter un ensemble fini de nombres réels. Si dans les débuts de l'informatique chaque constructeur les a implémentés différemment, ce qui rendait l'écriture de programme les utilisant très difficile, la norme IEEE 754 [4] (introduite en 1985 et révisée en 2008) a ensuite défini, parmi d'autres, plusieurs formats comme les simple et double précisions en base 2 ou 10.

Cependant tous les formats se ressemblent : étant donnés $\beta \in \mathbb{N} \setminus \{0, 1\}$ la base, $p \in \mathbb{N} \setminus \{0\}$ la précision, $e_{min}, e_{max} \in \mathbb{Z}$ les bornes de l'exposant, les nombres à virgule flottante de forme normale représentables s'écrivent

$$(-1)^s \cdot m \cdot \beta^e$$

avec

— $s \in \{0, 1\}$ le signe ;
— m le significande tel que $1 \leq m < \beta$ et m a un chiffre avant la virgule et au plus $p - 1$ après, en base β ;
— e est l'exposant tel que $e_{min} \leq e \leq e_{max}$.

Dans la norme IEEE 754, d'autres nombres sont représentables, en jouant avec des exposants spéciaux : 0, $+\infty$, $-\infty$, des NaN (not a number) qui signifient une erreur, et des nombres dénormalisés, c'est-à-dire plus petits en valeur absolue que le plus petit nombre normal : le significande s'écrit avec un 0 avant la virgule.

Les deux standards largement utilisés de la norme IEEE 754 utilisent les valeurs :

	β	p	e_{min}	e_{max}	taille totale
simple précision	2	24	-126	127	32 bits
double précision	2	53	-1022	1023	64 bits

Cette norme définit également des opérations arithmétiques ($+$, $-$, $\times$, $/$, $\sqrt{}$), ainsi que plusieurs modes d'arrondis. Lorsqu'une opération arithmétique est effectuée, le résultat doit être celui qui serait obtenu en effectuant l'opération avec une précision infinie puis en appliquant l'arrondi. Une telle opération est dite *correctement arrondie*. La norme donne aussi des détails d'implémentation comme le bit caché ou le biais sur les exposants.

1.2 Limites

La donnée de $(\beta, e_{min}, e_{max}, p)$ fixe l'ensemble des nombres représentables dans ce système, les autres étant alors approchés selon un mode d'arrondi prédéfini (au plus proche noté RN, vers 0, vers $+\infty$, ou vers $-\infty$).

En ne prêtant pas attention à la représentation des nombres, on peut avoir des surprises, comme dans cet exemple [9, page 8] : la suite

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}} \end{cases}$$

tend vers 6, mais quelle que soit la précision utilisée le calcul numérique converge vers 100 à cause d'un arrondi inévitable. Effectivement, les premiers termes ont été choisis afin de correspondre à un cas particulier de convergence de la suite, mais la plus petite erreur d'arrondi l'en écarte et la suite converge finalement vers 100.

Autrement, de nombreux problèmes numériques dans les systèmes dynamiques ou la dynamique des orbites planétaires, comme la stabilité à long terme du système solaire [8], trouver des orbites périodiques stables pour le système dynamique de Hénon [6], ou itérer l'attracteur de Lorenz [1] nécessitent de plus grandes précisions que la double précision standard, maintenant appelée *binary64* [4].

Nous allons voir les différentes solutions envisagées pour disposer de nombres plus précis, notamment à l'aide de sommes non évaluées de nombres à virgule flottante.

1.3 Solutions

Il est possible de faire une addition ou une multiplication de deux nombres à virgule flottante sans erreurs, c'est-à-dire que le résultat de l'opération est composé de deux nombres : l'arrondi et l'erreur. Pour l'addition il y a les algorithmes [7] 2SUM (qui nécessite 6 opérations), et Fast2SUM (qui n'en nécessite que 3 mais il faut alors connaître l'ordre des termes). Pour la multiplication il y a le produit de Dekker et, sur certains processeurs, l'instruction FMA (Fused Multiply-Add).

Ces algorithmes forment des briques de base pour faire des calculs avec une précision accrue : la bibliothèque QD les utilise sur des double, triple et quad-words : jusqu'à quatre nombres à virgule flottante représentent une seule donnée qui est la somme, jamais évaluée, des quatre nombres. Un exemple d'algorithme sur l'addition de deux double-words est donné ci-dessous.

La bibliothèque GNU MPFR [2] fonctionne différemment mais peut utiliser n'importe quelle précision. Elle effectue ses calculs avec plusieurs entiers qui représentent les chiffres ainsi qu'un exposant.

Voici un algorithme utilisant 2SUM et Fast2SUM qui effectue la somme de deux double-words (x_h, x_l) et (y_h, y_l) [9, page 497] :

$(s_h, s_l) \leftarrow$ 2Sum (x_h, y_h)
$(t_h, t_l) \leftarrow$ 2Sum (x_l, y_l)
$c \leftarrow$ RN $(s_l + t_h)$
$(v_h, v_l) \leftarrow$ Fast2Sum (s_h, c)
$w \leftarrow$ RN $(t_l + v_l)$
$(z_h, z_l) \leftarrow$ Fast2Sum (v_h, w)
return (z_h, z_l)

2 Expansions de nombre à virgule flottante

C'est une extension de la manière de représenter les données dans QD : on utilise des sommes non évaluées de nombres à virgule flottante de taille arbitraire $a = a_0 + \cdots + a_{k-1}$. Si la précision de chaque terme est p, la précision finale peut être au moins kp.

Afin de diminuer la redondance, et également de pouvoir faire des algorithmes plus efficaces, on demande que ces nombres soient deux à deux sans chevauchement et triés dans l'ordre décroissant. Il existe plusieurs définitions de non chevauchement :

Définition 1. *Non chevauchement selon Priest*

Si $x = (-1)^{s_x} \cdot m_x \cdot \beta^{e_x}$ et $y = (-1)^{s_y} \cdot m_y \cdot \beta^{e_y}$ sont deux nombres à virgule flottante sous forme normale, x et y sont sans $\mathcal{P}$-chevauchement si $|e_x - e_y| \geq p$.

C'est-à-dire qu'il n'y a pas de bit de même poids dans les significandes de x et de y.

Définition 2. *Non chevauchement selon Bailey*

Si $x < y$ sont deux nombres à virgule flottante sous forme normale, et si ulp(y) est la distance entre y et son plus proche voisin dans l'ensemble des nombres représentables, x et y sont sans $\mathcal{B}$-chevauchement si $|x| \leq \frac{1}{2}$ulp(y).

C'est-à-dire que y est l'un des nombres à virgule flottante les plus proche de $x + y$ (il y en a au plus deux). Cette dernière définition est plus forte.

Par exemple, en binaire et avec $p = 4$, les nombres $1.000 \cdot 2^4$ et $1.000 \cdot 2^0$ sont sans $\mathcal{P}$-chevauchement mais avec $\mathcal{B}$-chevauchement. La représentation sans $\mathcal{B}$-chevauchement de la somme serait $1.001 \cdot 2^4 - 1.000 \cdot 2^0$.

2.1 Calculs avec des expansions de nombre à virgule flottante

L'addition et la multiplication ont été largement étudiées [9, 3, Thm. 44, Chap. 14], elles utilisent les briques comme 2SUM en effectuant les opérations termes à termes et en propageant les termes d'erreurs.

Ces opérations peuvent ensuite être utilisées comme boîtes noires pour écrire des calculs plus complexes comme la division ou la racine carrée, qui ne sont pas linéaires et ne peuvent pas être calculées de la même manière. Ma maître de stage a utilisé l'itération de Newton-Raphson pour calculer l'inverse [5], et j'ai adapté la méthode au calcul de racines carrées.

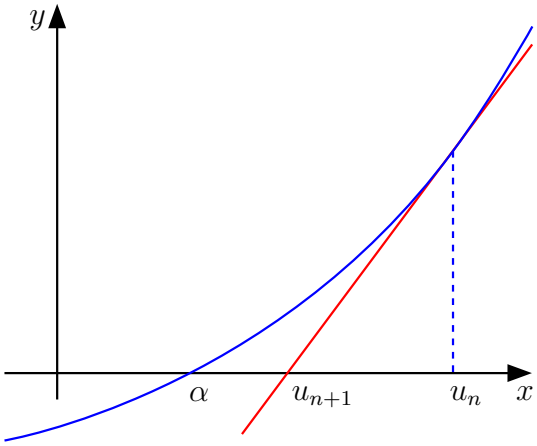
2.2 L'itération de Newton-Raphson

C'est une suite récurrente qui converge vers une racine d'une fonction f donnée de façon quadratique lorsque f'' est continue, pour peu que le premier terme soit suffisamment proche de cette racine.

Si la racine que l'on cherche est isolée, la dérivée f' est non nulle dans son voisinage. À partir de la tangente en $(u_n, f(u_n))$, on trouve u_{n+1} l'abscisse de l'intersection de cette tangente avec l'axe des abscisses, ce qui donne $u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$ (voir figure 1)

Pour calculer l'inverse de a, on cherche la racine de la fonction $f : x \mapsto \frac{1}{x} - a$, ce qui donne la suite récurrente $u_{n+1} = u_n(2 - au_n)$.

FIGURE 1 – L'iteration de Newton



Pour calculer la racine carrée de a, on peut utiliser la fonction $f : x \mapsto x^2 - a$, ce qui conduit à la suite de Babylone, soit $u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$, qui a le défaut de comporter une division. Pour l'éviter il est possible de calculer $\frac{1}{\sqrt{a}}$ en cherchant le zéro positif de la fonction $f : x \mapsto \frac{1}{x^2} - a$, ce qui nous conduit à la formule de récurrence $u_{n+1} = \frac{1}{2}u_n\left(3 - au_n^2\right)$. Le résultat peut être multiplié par a pour retrouver la racine carrée.

Le point délicat avec cette méthode est de trouver une approximation du résultat suffisamment bonne pour que la suite converge, et que le résultat soit précis dès les premières itérations. Il est très vite résolu ici car on dispose des opérations usuelles sur les nombres à virgule flottante, et donc si on cherche $f(a)$ avec $a = a_0 + \cdots + a_k$ on peut partir de $u_0 = \mathrm{RN}(f(a_0))$ qui peut constituer une bonne approximation si f est suffisamment régulière.

2.3 Algorithme

Afin de diminuer le nombre d'opérations, on essaye de demander la précision minimale à chaque étape de calcul, sans perdre la convergence : à la i-ième itération on va utiliser le résultat de l'itération précédente de 2^{i-1} termes mais les autres données avec 2^i termes afin d'affiner le résultat.

La preuve de cet algorithme, donnée en annexe A, est très importante car il y a deux types d'erreurs de calcul : des erreurs de méthode, puisque l'itération de Newton-Raphson en introduit une à chaque étape, qui décroit quadratiquement, et des erreurs de calcul car les additions et multiplications intermédiaires avec des expansions sont tronquées. En effet une multiplication de i termes avec j termes donne au plus $2ij$ termes, et l'addition donne au plus $i + j$ termes qu'il faut tronquer pour garder un nombre raisonnable d'opérations.

On notera $x^{(i)}$ l'expansion de nombres à virgule flottante x tronquée à ses i premiers termes.

Entrée de l'algorithme : une expansion de nombres à virgule flottante sans $\mathcal{B}$-chevauchement $a = a_0 + \cdots + a_{2^k-1}$ et la précision cible : 2^q termes.

Sortie de l'algorithme : une expansion de nombres à virgule flottante $x = x_0 + \cdots + x_{2^q-1}$ telle que

$$\boxed{\left|x - \frac{1}{\sqrt{a}}\right| \leq \frac{2^{-2^q(p-3)-1}}{\sqrt{a}}}$$

Algorithme :

$x_0 \leftarrow 1/\sqrt{a_0}$
for $i \leftarrow 1$ to q do
  $\hat{v}^{(2^i)} \leftarrow x^{(2^{i-1})} \times \hat{a}^{(2^i)}$
  $\hat{w}^{(2^i)} \leftarrow x^{(2^{i-1})} \times \hat{v}^{(2^i)}$
  $\hat{y}^{(2^i)} \leftarrow 3 - \hat{w}^{(2^i)}$
  $\hat{z}^{(2^i)} \leftarrow x^{(2^{i-1})} \times \hat{y}^{(2^i)}$
  $x^{(2^i)} \leftarrow \hat{z}^{(2^i)} / 2$ (opération très facile en binaire)
end for
return $x = x_0 + \cdots + x_{2^q-1}$

Les détails de la preuve de cet algorithme est en annexe A, mais détaillons son principe ici. On montre par récurrence qu'à la fin du i-ième tour de boucle, $\varepsilon_i = \left|x^{(2^i)} - \frac{1}{\sqrt{a}}\right| \leq \frac{2^{-2^i(p-3)-1}}{\sqrt{a}}$. L'initialisation et l'hérédité se montrent en prenant en compte à chaque étape les erreurs commises par les calculs et du fait de la troncature des opérandes. À l'aide d'une suite d'inégalités triangulaires, on fait apparaître l'expression de la suite de Newton-Raphson et un terme d'erreur.

2.4 Comparaison avec MPFR

Nous avons effectué certains calculs aléatoires avec à la fois MPFR et notre bibliothèque. Les résultats concordent toujours et la borne d'erreur annoncée est vérifiée, ce qui valide expérimentalement l'algorithme.

2.5 Cas du $\mathcal{P}$-chevauchement

Même si le non $\mathcal{P}$-chevauchement est plus faible que le non $\mathcal{B}$-chevauchement et donc une preuve dans le premier cas implique le résultat dans le second, la preuve avec la définition de Bailey est un peu plus limpide et j'ai donc présenté celle-ci en annexe, mais avec quelques modifications elle fonctionne pour la définition de Priest.

2.6 Cas de la suite de Babylone

Le même genre de preuve s'applique pour un algorithme basé sur la suite de Babylone, en prenant les mêmes troncatures que dans le cas étudié ci-dessus. On obtient par contre une borne un peu moins serrée, pour les deux définitions de non chevauchement.

Entrée de l'algorithme : une expansion de nombres à virgule flottante sans $\mathcal{B}$-chevauchement $a = a_0 + \cdots + a_{2^k-1}$ et la précision cible : 2^q termes.

Sortie de l'algorithme : une expansion de nombres à virgule flottante $x = x_0 + \cdots + x_{2^q-1}$ telle que

$$\boxed{\left|x - \sqrt{a}\right| \leq 3\sqrt{a} \cdot 2^{-2^q(p-3)-2}}$$

Algorithme :

$x_0 \leftarrow \sqrt{a_0}$
for $i \leftarrow 1$ to q do
  $\hat{v}^{(2^i)} \leftarrow a^{(2^i)} / x^{(2^{i-1})}$
  $\hat{w}^{(2^i)} \leftarrow x^{(2^{i-1})} + \hat{v}^{(2^i)}$
  $x^{(2^i)} \leftarrow \hat{w}^{(2^i)} / 2$ (opération très facile en binaire)
end for
return $x = x_0 + \cdots + x_{2^q-1}$

2.7 Détails sur l'implémentation

L'implémentation d'algorithmes les plus efficaces possibles, de plus sur un GPU, nécessite quelques précautions. Par exemple, chaque branchement conditionnel vide la chaîne de traitement du processeur (pipeline) ce qui ralentit considérablement l'exécution du programme. L'algorithme 2SUM utilise 6 opérations sur les nombres à virgules flottante contre 3 pour sa variante Fast2SUM qui nécessite par contre d'ordonner les deux termes. Dans le cas où l'ordre est inconnu, il est plus rapide d'utiliser 2SUM que de faire un branchement conditionnel pour appeler Fast2SUM.

Dans la même idée, les boucles for peuvent être déroulées, afin de permettre au processeur d'exécuter le code plus rapidement. Notre bibliothèque est développée en C++ en utilisant les templates pour véhiculer l'information du nombre de termes dans les expansions. Cette façon de faire permet à un compilateur suffisamment optimisant de dérouler les boucles et de pré-calculer certains branchements conditionnels pour peu que tous les termes soit constants. Cela demande parfois de récrire une portion de programme pour permettre au compilateur d'effectuer correctement la propagation des constantes.

Une deuxième raison de l'utilisation des templates est liée à l'architecture GPU : lorsque la taille des tableaux (ici les données des expansions) n'est pas fixée à la compilation, ceux-ci sont placés dans une mémoire beaucoup plus lente que dans le cas contraire.

Le défaut de ces optimisations est la taille du binaire produit qui peut augmenter très vite : pour chaque paramètres template différents, et il peut y en avoir plusieurs (précision des arguments et du résultat), un code est généré, et chaque code est lui-même plus gros à cause du déroulement des boucles et de l'*inlining*.

Enfin le code est compatible avec gcc pour cibler un CPU, ainsi que nvcc pour les GPU, le compilateur de CUDA [10].

Conclusion

La méthode avec l'itération de Newton-Raphson s'adaptait très bien au cas de la racine carrée, avec l'astuce de calculer plutôt son inverse. De même, l'idée de la preuve fonctionnait, même si j'ai fait quelques remaniements et améliorations (surtout sur l'hérédité de la dernière récurrence, qui se conclut directement par décroissance d'une bonne fonction).

L'algorithme et sa preuve contribuent au développement de la bibliothèque, ce qui remplit parfaitement les objectifs principaux fixés pour mon stage.

Pour finir je voudrais remercier mon tuteur pour m'avoir aidé dans mes recherches, ma maître de stage pour m'avoir pris, et mes collègues pour les très bonnes semaines que j'ai pu passer avec eux.

A Preuve de convergence

La preuve suit le même raisonnement que dans [5]. Le but est de montrer que l'erreur relative diminue à chaque boucle de l'algorithme, en prenant en compte les arrondis de chaque calcul. La stratégie est de faire apparaître le terme exact de la suite de Newton, qu'on sait borner, avec un terme d'erreur.

Démonstration. Dans toute la suite on note p la précision des nombres à virgule flottante utilisés, et $a > 0$ une expansion de nombres à virgule flottante sans $\mathcal{B}$-chevauchement avec 2^k termes (on notera $a = a_0 + \cdots + a_{2^k-1}$), et dont on cherche la racine carrée de l'inverse.

Préliminaires On rappelle un résultat démontré dans ce papier : soit une expansion de nombres à virgule flottante $u = u_0 + \cdots + u_k$ avec $k > 0$ nombres à virgule flottante binaires normaux de précision p. On pose $\boxed{\eta = \sum_{j=0}^{+\infty} 2^{-p(j+1)} = \frac{1}{2^p - 1}}$.
Alors on a $|u_i| \leq 2^{-ip}|u_0|$ et $|u - u^{(i+1)}| \leq 2^{-ip}|u|\frac{\eta}{1-\eta}$ (1) ainsi que $\frac{|u|}{1+\eta} \leq |u_0| \leq \frac{|u|}{1-\eta}$ (2).
Dans la même idée que dans ce papier on montre que $\left|\frac{1}{\sqrt{u}} - \frac{1}{\sqrt{u_0}}\right| \leq \eta\frac{1}{\sqrt{u}}$ (3) :
Remarquons déjà que $\left|\frac{1}{\sqrt{u}} - \frac{1}{\sqrt{u_0}}\right| = \frac{1}{\sqrt{u}}\left|1 - \frac{\sqrt{u}}{\sqrt{u_0}}\right|$ puis par (2), avec u et u_0 qui ont le même signe d'après les inégalités de non-chevauchement (et ils seront positifs dans la suite), et par croissance de la fonction racine carrée, on a $\left|1 - \sqrt{\frac{u_0}{u}}\right| \leq 1 - \sqrt{\frac{1}{1+\eta}} \leq \eta$ ce qui permet de conclure.

Dans toute la suite, la notation x_i dénotera l'expansion x lors de la i-ième itération, et non plus le i-ième terme de l'expansion, dont on n'a plus besoin directement.

Borne de ε_0 On cherche à borner $\varepsilon_0 = \left|x_0 - \frac{1}{\sqrt{a}}\right|$:
On pose $\alpha = \text{RN}(\sqrt{a_0})$, et alors $x_0 = \text{RN}\,(1/\alpha)$
Par RN, $|\alpha - \sqrt{a_0}| \leq 2^{-p}\sqrt{a_0}$ (4) et $\left|x_0 - \frac{1}{\alpha}\right| \leq 2^{-p}\frac{1}{\alpha}$ (5)
De (4) on obtient $\left|\frac{1}{\alpha} - \frac{1}{\sqrt{a_0}}\right| \leq \left(\frac{1}{1-2^{-p}} - 1\right)\frac{1}{\sqrt{a_0}}$
Et

$$\begin{aligned}
\left|x_0 - \tfrac{1}{\sqrt{a_0}}\right| &\leq \left|x_0 - \tfrac{1}{\alpha}\right| + \left|\tfrac{1}{\alpha} - \tfrac{1}{\sqrt{a_0}}\right| \\
&\leq 2^{-p}\tfrac{1}{\alpha} + \left|\tfrac{1}{\alpha} - \tfrac{1}{\sqrt{a_0}}\right| \text{ par (5)} \\
&\leq 2^{-p}\tfrac{1}{\sqrt{a_0}} + (2^{-p}+1)\left|\tfrac{1}{\alpha} - \tfrac{1}{\sqrt{a_0}}\right| \\
&\leq 2\eta\tfrac{1}{\sqrt{a_0}} \\
\varepsilon_0 &\leq \left|x_0 - \tfrac{1}{\sqrt{a_0}}\right| + \left|\tfrac{1}{\sqrt{a}} - \tfrac{1}{\sqrt{a_0}}\right| \\
&\leq 2\eta\tfrac{1}{\sqrt{a_0}} + \eta\tfrac{1}{\sqrt{a}} \text{ par (3)}
\end{aligned}$$

Or $\frac{\sqrt{a}}{\sqrt{1+\eta}} \leq \sqrt{a_0}$ et donc $\frac{1}{\sqrt{a_0}} \leq \frac{\sqrt{1+\eta}}{\sqrt{a}} \leq \frac{1+\eta/2}{\sqrt{a}}$
On obtient $\varepsilon_0 \leq \left(2\eta(1+\frac{\eta}{2}) + \eta\right)\frac{1}{\sqrt{a}}$ soit $\varepsilon_0 \leq \eta(3+\eta)\frac{1}{\sqrt{a}}$
On pose $\boxed{E_i = \varepsilon_i\sqrt{a}}$ et alors

$$\boxed{E_0 \leq \eta(3+\eta)}$$

Borne de ε_{i+1} À chaque étape de l'algorithme, v_i, w_i, et y_i sans chapeau dénotent les résultats exacts des calculs, et avec un chapeau les résultats tronqués à 2^{i+1} termes. τ_i

désigne le résultat exact des deux dernières opérations : $\frac{1}{2}x_i\hat{y}_i$, dont x_{i+1} est le tronqué à 2^{i+1} termes.

On pose $\boxed{\gamma_i = 2^{-(2^{i+1}-1)p}\dfrac{\eta}{1-\eta}}$ et on a alors par (1) :

$$\begin{array}{lcll} |x_{i+1}-\tau_i| & \leq & \gamma_i|\tau_i| \leq \gamma_i\left|\frac{1}{2}x_i\hat{y}_i\right| & \text{(6a)}\\ |y_i-\hat{y}_i| & \leq & \gamma_i|y_i| \leq \gamma_i|3-\hat{w}_i| & \text{(6b)}\\ |w_i-\hat{w}_i| & \leq & \gamma_i|w_i| \leq \gamma_i|x_i\hat{v}_i| & \text{(6c)}\\ |v_i-\hat{v}_i| & \leq & \gamma_i|v_i| \leq \gamma_i|x_i a^{(f_i)}| & \text{(6d)}\\ |a-a^{(f_i)}| & \leq & \gamma_i|a| & \text{(6e)} \end{array}$$

Par une suite d'inégalités triangulaires, on a

$$\begin{array}{rcl} \varepsilon_{i+1} & = & \left|x_{i+1}-\frac{1}{\sqrt{a}}\right| \\ & \leq & |x_{i+1}-\tau_i| + \left|\tau_i-\frac{1}{\sqrt{a}}\right| \\ & \leq & \gamma_i\left|\frac{1}{2}x_i\hat{y}_i\right| + |\frac{1}{2}x_i\hat{y}_i-\frac{1}{\sqrt{a}}| \text{ par (6a), or } |\hat{y}_i| \leq (1+\gamma_i)|3-\hat{w}_i| \text{ par (6b)} \\ & \leq & \gamma_i(2+\gamma_i)|\frac{1}{2}x_i(3-\hat{w}_i)| + |\frac{1}{2}x_i(3-\hat{w}_i)-\frac{1}{\sqrt{a}}| \text{ par (6b)} \\ & \leq & \gamma_i(2+\gamma_i)|\frac{1}{2}x_i|(|3-w_i|+\gamma_i|w_i|) + \gamma_i|\frac{1}{2}x_iw_i| + |\frac{1}{2}x_i(3-w_i)-\frac{1}{\sqrt{a}}| \text{ par (6c)} \\ & \leq & \gamma_i|\frac{1}{2}x_i|\left((2+\gamma_i)|3-x_i\hat{v}_i| + (1+\gamma_i(2+\gamma_i))|x_i\hat{v}_i|\right) + |\frac{1}{2}x_i(3-x_i\hat{v}_i)-\frac{1}{\sqrt{a}}| \\ & \leq & \gamma_i|\frac{1}{2}x_i|\left((2+\gamma_i)(|3-x_iv_i|+\gamma_i|v_ix_i|) + (1+\gamma_i(2+\gamma_i))(1+\gamma_i)|x_iv_i|\right) \\ & & +\gamma_i|\frac{1}{2}x_i^2v_i| + |\frac{1}{2}x_i(3-x_iv_i)-\frac{1}{\sqrt{a}}| \text{ par (6d)} \\ & \leq & \gamma_i(2+\gamma_i)|\frac{1}{2}x_i|(|3-x_i^2a^{(f_i)}| + (\gamma_i+1)^2|x_i^2a^{(f_i)}|) + |\frac{1}{2}x_i(3-x_i^2a^{(f_i)})-\frac{1}{\sqrt{a}}| \\ & \leq & \gamma_i(2+\gamma_i)(|\frac{1}{2}x_i(3-x_i^2a)| + \gamma_i|\frac{1}{2}x_i^3a|) + \gamma_i|\frac{1}{2}x_i^3a| + \gamma_i(2+\gamma_i)(\gamma_i+1)^3|\frac{1}{2}x_i^3a| \\ & & +|\frac{1}{2}x_i(3-x_i^2a)-\frac{1}{\sqrt{a}}| \text{ par (6e)} \\ & \leq & \gamma_i(2+\gamma_i)|\frac{1}{2}x_i(3-x_i^2a)-\frac{1}{\sqrt{a}}| + \gamma_i((\gamma_i+1)^2+(\gamma_i+1)^3+(\gamma_i+1)^4)|\frac{1}{2}x_i^3a| \\ & & +|\frac{1}{2}x_i(3-x_i^2a)-\frac{1}{\sqrt{a}}| + \gamma_i(2+\gamma_i)\frac{1}{\sqrt{a}} \text{ en faisant apparaître dans le premier terme l'expression de la suite de Newton-Raphson.} \\ & \leq & (1+\gamma_i)^2|x_{i+1}-\frac{1}{\sqrt{a}}| + \gamma_i((\gamma_i+1)^2+(\gamma_i+1)^3+(\gamma_i+1)^4)|\frac{1}{2}x_i^3a| + \gamma_i(2+\gamma_i)\frac{1}{\sqrt{a}} \end{array}$$

Or $|x_{i+1}-\frac{1}{\sqrt{a}}| = \frac{1}{2}\sqrt{a}(x_i\sqrt{a}+2)|x_i-\frac{1}{\sqrt{a}}|^2$ (convergence quadratique de la suite) et d'après $\varepsilon_i = |x_i-\frac{1}{\sqrt{a}}|$, on a $|x_i\sqrt{a}| \leq \varepsilon_i\sqrt{a}+1$ et $|x_i^3a| \leq \frac{(\varepsilon_i\sqrt{a}+1)^3}{\sqrt{a}}$ d'où

$$\begin{array}{rcl} \varepsilon_{i+1} & \leq & \frac{1}{2}(1+\gamma_i)^2\sqrt{a}(\varepsilon_i\sqrt{a}+3)\varepsilon_i^2 \\ & & +\frac{1}{2}\gamma_i((\gamma_i+1)^2+(\gamma_i+1)^3+(\gamma_i+1)^4)\frac{(\varepsilon_i\sqrt{a}+1)^3}{\sqrt{a}} \\ & & +\gamma_i(2+\gamma_i)\frac{1}{\sqrt{a}} \end{array}$$

On obtient avec $E_i = \varepsilon_i\sqrt{a}$ une équation indépendante de a :

$$\boxed{E_{i+1} \leq \frac{1}{2}(1+\gamma_i)^2(E_i+3)E_i^2 + \frac{1}{2}\gamma_i((\gamma_i+1)^2+(\gamma_i+1)^3+(\gamma_i+1)^4)(E_i+1)^3 + \gamma_i(2+\gamma_i)}$$

On note f la fonction telle que l'inégalité précédente s'écrive $\boxed{E_{i+1} \leq f(E_i, i)}$.

Conclusion On pose $\boxed{ind(i) = 2^{-2^i(p-3)-1}}$, et on veut montrer $\forall i \in \mathbb{N}, E_i \leq ind(i)$

Pour $i = 0$ on vérifie que $E_0 \leq ind(0)$ dès que $p \geq 3$.

Pour $i \geq 1$ par récurrence :

Initialisation : pour $i = 1$, on trouve avec un logiciel de calcul formel que l'inégalité est vérifiée dès que $p \geq 4$, ce qui est toujours le cas en pratique.

Hérédité : notons d'abord que la fonction $i \mapsto \frac{f(ind(i),i)}{ind(i+1)}$ est décroissante et sa valeur en 1 est inférieure à 1 dès que $p \geq 3$. Ainsi, sous cette condition, pour tout $i \geq 1$, $\frac{f(ind(i),i)}{ind(i+1)} \leq 1$.

Soit $i \geq 1$, on suppose que $E_i \leq ind(i)$ et on a $E_{i+1} \leq f(E_i, i)$ d'où, par croissance, $E_{i+1} \leq f(ind(i), i)$. On a alors $\frac{E_{i+1}}{ind(i+1)} \leq \frac{f(ind(i),i)}{ind(i+1)} \leq 1$ ce qui conclut l'hérédité et la

récurrence.
Enfin on trouve l'inégalité finale avec $i = q$. □

Références

[1] A. Abad, R. Barrio, and Á. Dena. Computing periodic orbits with arbitrary precision. *Phys. Rev. E*, 84 :016701, Jul 2011.

[2] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR : A Multiple-Precision Binary Floating-Point Library with Correct Rounding. 33(2), 2007.

[3] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 155–162, Vail, CO, June 2001.

[4] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008. available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[5] M. Joldes, J.-M. Muller, and Valentina Popescu. On the computation of the reciprocal of floating point expansions using an adapted Newton-Raphson iteration. In *Proceedings of 25th IEEE International Conference on Application-specific Systems, Architectures and Processors*, page to appear, Zurich, Suisse, 2014.

[6] M. Joldes, V. Popescu, and W. Tucker. Searching for sinks of Hénon map using a multiple-precision GPU arithmetic library. Technical report, LAAS, Nov 2013.

[7] D. Knuth. *The art of computer programming*, volume 2. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[8] J. Laskar and M. Gastineau. Existence of collisional trajectories of Mercury, Mars and Venus with the Earth. *Nature*, 459(7248) :817–819, June 2009.

[9] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[10] NVIDIA. *The CUDA Compiler Driver NVCC*, 2007.