HAL
open science

Matériel et logiciel pour l'évaluation de fonctions numériques :précision, performance et validation

Florent de Dinechin

To cite this version:

Florent de Dinechin. Matériel et logiciel pour l'évaluation de fonctions numériques :précision, performance et validation. Informatique [cs]. Université Claude Bernard - Lyon I, 2007. tel-00270151

HAL Id: tel-00270151

https://theses.hal.science/tel-00270151v1

Submitted on 3 Apr 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Claude Bernard Lyon 1

Matériel et logiciel pour l'évaluation de fonctions numériques

Précision, performance et validation

Florent de Dinechin

Maître de conférences à l'École Normale Supérieure de Lyon

Mémoire d'habilitation à diriger des recherches

présenté le **28 juin 2007**, après avis des rapporteurs

Jean-Claude BAJARD, Professeur, Université de Montpellier 2
Miloš ERCEGOVAC, Professor, University of California, Los Angeles
Paul ZIMMERMANN, Directeur de Recherche, INRIA

devant le jury composé de

Jean-Claude BAJARD, Professeur, Université de Montpellier 2
Bernard GOOSSENS, Professeur, Université de Perpignan
Peter MARKSTEIN, Senior Engineer, Hewlett Packard
Jean-Michel MOREAU, Professeur, Université Claude Bernard Lyon 1
Jean-Michel MULLER, Directeur de Recherche, CNRS
Paul ZIMMERMANN, Directeur de Recherche, INRIA

Numéro d'ordre 22-2007

À mon Papa, qui l'aurait lu.

Remerciements

À tout seigneur tout honneur, je voudrais avant tout remercier Jean-Michel Muller de m'avoir accueilli dans le projet Arénaire, d'avoir guidé mes premiers pas dans l'arithmétique des ordinateurs, d'être à l'origine de la plupart des sujets de recherche rapportés dans ce mémoire, de m'avoir appris avec autant d'efficacité que de désintéressement à encadrer des étudiants, et de l'indispensable plaisanterie douteuse quotidienne.

Je voudrais aussi remercier Jean-Claude Bajard, Miloš Ercegovac et Paul Zimmermann d'avoir accepté de rapporter sur ces travaux. Leurs suggestions ont notablement contribué à améliorer ce document. Je remercie également les membres du jury, Bernard Goossens, Peter Markstein et Jean-Michel Moreau, de leurs remarques, questions et encouragements.

Ce mémoire serait moins épais s'il ne résumait que mon travail : on y trouve aussi celui de mes trois thésards, David Defour, Jérémie Detrey et Christoph Quirin Lauter. Il paraît qu'on doit parfois passer du temps à s'occuper de ses étudiants ; ces trois-là, j'ai surtout passé du temps à essayer de les suivre. Je les remercie de tout ce qu'ils ont apporté à mes recherches, et du reste.

Le projet Arénaire a toujours été une vraie équipe, où chacun m'a, à un moment ou à l'autre, donné un coup de main. Je ne résiste donc pas à la facilité de remercier tous ses membres présents ou passés. Je remercie en particulier, dans l'ordre alphabétique : Jean-Luc Beuchat, cryptologue et défenseur du gruyère sans trou ; Sylvie Boyer, sans laquelle on n'irait pas très loin ; Nicolas Brisebarre, de mettre des mathématiques là où on ne lui a rien demandé ; Sylvain Chevillard, qui cuisine les meilleurs polynômes de tout l'univers ; Catherine Daramy, d'avoir géré mon CVS comme son sac à main ; Marc Daumas, de m'avoir prêté si longtemps un de ses thésards ; Claude-Pierre Jeannerod, de m'avoir prêté si longtemps son stagiaire de master ; Philippe Langlois, de son approche méridionale de tout ; Vincent Lefèvre, de m'avoir aidé dans au moins 17 occasions ; Guillaume Melquiond, de Gappa et du service après-vente associé ; Nathalie Revol, de sa disponibilité permanente qui ne l'aide pas à rédiger son habilitation à elle ; Arnaud Tisserand, de m'avoir supporté dans le bureau, d'avoir toujours pris le temps de répondre à mes questions, et aussi d'avoir assuré la maintenance d'outils utilisés surtout par mes étudiants à moi ; Serge Torres, de sa perfection modeste, tant au plan professionnel qu'au plan humain ; Gilles Villard, de son soutien sans faille, et aussi de son irremplaçable capacité à jouer les Candide.

Je remercie encore Andrey Naraikin et Sergey Maidanov, de l'*Intel Nizhny-Novgorod Laboratory*, de m'avoir donné la chance de découvrir en même temps la vie dans un laboratoire Intel et l'hiver russe.

Merci aussi à mon frère Christophe qui a relu et commenté ce manuscript.

Enfin et surtout, je remercie mon épouse Marina et mes petites filles Daria et Aliona d'avoir juste été là.

Table des matières

1 Généralités **3**
1.1 Les données du problème . . . 3
1.2 Une affaire de compromis . . . 4
1.3 Les techniques d'évaluation d'une fonction . . . 4
1.3.1 Approximation polynomiale ou rationnelle . . . 4
1.3.2 Réduction d'argument . . . 5
1.3.3 Schémas d'évaluation polynomiale . . . 6
1.4 La maîtrise des erreurs . . . 7
1.4.1 Les différentes sources d'erreur . . . 7
1.4.2 Pour des bornes d'erreur fines . . . 8
1.4.3 Le vocabulaire des calculs d'erreurs . . . 9

2 Méthodes d'approximation adaptées au matériel **11**
2.1 Introduction . . . 11
2.2 Quelques applications . . . 12
2.2.1 Traitement du signal . . . 12
2.2.2 Initialisation des itérations quadratiques pour la division et la racine carrée . . . 13
2.2.3 Opérateurs logarithmiques . . . 14
2.3 Les cibles technologiques . . . 15
2.3.1 ASIC ou FPGA . . . 15
2.3.2 Les métriques du matériel et les compromis architecturaux . . . 16
2.3.3 Les briques de base et leur coût . . . 18
2.4 Méthodes d'ordre 1 . . . 20
2.4.1 La méthode bipartite . . . 20
2.4.2 Une optimisation : l'exploitation de la symétrie . . . 22
2.4.3 Les méthodes multipartites . . . 22
2.5 Méthodes d'ordre supérieur . . . 27
2.5.1 Travaux antérieurs . . . 27
2.5.2 La méthode HOTBM . . . 29
2.5.3 Applications aux opérateurs logarithmiques . . . 29
2.5.4 Perspectives sur l'évaluation de fonctions en virgule fixe . . . 30
2.6 Fonctions élémentaires en virgule flottante pour FPGAs . . . 31
2.6.1 Les degrés de liberté du matériel . . . 31
2.6.2 L'exemple du logarithme . . . 32

2.6.3 Performance : le FPGA surpasse le processeur 35
2.7 Conclusion : le retour des fonctions élémentaires en matériel 36

3 L'arrondi correct dans la bibliothèque mathématique 39
3.1 Enjeux et difficultés 39
3.1.1 La virgule flottante dans les calculateurs modernes 39
3.1.2 Des fonctions élémentaires imparfaitement spécifiées 40
3.1.3 Le dilemme du fabricant de tables 40
3.1.4 L'arrondi correct par raffinement de précision 42
3.1.5 Quelle précision pour l'arrondi correct ? 42
3.2 CRLibm, vers une bibliothèque mathématique parfaite 44
3.2.1 Deux étapes de Ziv 44
3.2.2 Compromis précision/performance 44
3.2.3 Compromis portabilité/optimisation 45
3.3 Techniques de précision étendue pour l'étape précise 46
3.3.1 La retenue conservée logicielle 46
3.3.2 Critique de SCSlib pour CRLibm 47
3.3.3 Arithmétiques double-double, triple-double et double-double-étendue 48
3.3.4 Génération de code triple-double pour CRLibm 50
3.4 Les pires cas spéciaux de Lauter 50
3.4.1 Une découverte par hasard 50
3.4.2 Quand les mathématiques ordonnent les statistiques 52
3.4.3 Implications pratiques 54
3.4.4 L'arrondi correct dans les cas de Lauter 55
3.5 Un exemple étendu : le logarithme de CRLibm 56
3.6 Conclusion I : l'arrondi correct normalisé 56
3.7 Conclusion II : vers des fonctions d'intervalle parfaites 57

4 Calcul d'erreur et validation 61
4.1 Bien poser la question 62
4.2 Un exemple de trois lignes 63
4.3 En quoi a-t-on confiance ? 64
4.4 Des progrès de l'automatisation 66
4.5 Une preuve Gappa 68
4.5.1 L'outil Gappa dans les grandes lignes 68
4.5.2 L'énoncé du problème dans Gappa 69
4.5.3 Aide Gappa, et Gappa t'aidera 70
4.5.4 Une page de publicité 72
4.6 Limites et perspectives 73
4.7 Pour conclure 73

5 Conclusions et perspectives 75

Bibliographie 76

Introduction

Ce mémoire reprend quelques résultats obtenus entre 2000 et 2007 au sein du projet Arénaire du LIP. La problématique centrale est l'évaluation de fonctions numériques : étant donnée une fonction réelle $f : \mathbb{R} \to \mathbb{R}$, nous cherchons à construire des opérateurs (matériels au chapitre 2, logiciels au chapitre 3), l'évaluant sous certaines contraintes et avec la meilleure qualité possible sur un certain intervalle I.

La question de la représentation du nombre en entrée et du résultat sous la forme de chaînes de bits est le cœur de l'arithmétique des ordinateurs [60]. Toutefois, ce sera pour nous une donnée du problème : un nombre sera représenté classiquement en virgule fixe ou en virgule flottante. Néanmoins, on utilisera si nécessaire pour les étapes intermédiaires du calcul des représentations un peu moins standard [41, 28].

La fonction étudiée pourra être

- un polynôme,
- une fonction algébrique, comme la racine carrée ou cubique, l'inverse $1/x$, ...
- une fonction élémentaire (exponentielle, logarithme, trigonométrique, etc.),
- une fonction spéciale (telle que les fonctions Gamma, Erreur, etc.)
- toute autre fonction d'une variable réelle présentant certaines bonnes propriétés, essentiellement la dérivabilité jusqu'à un certain ordre sur l'intervalle considéré.

On parle d'*évaluation* de la fonction plutôt que de *calcul* : en effet, il est courant que l'image $f(x)$ d'un nombre x en entrée ne soit pas calculable numériquement en temps fini. C'est le cas des fonctions élémentaires sur la plupart de leurs entrées : un nombre en entrée est un rationnel, et son image sera irrationnelle [112]. Même pour les fonctions algébriques, l'image d'un nombre en entrée ne sera le plus souvent pas représentable exactement dans le format imposé à la sortie. Dans ce cas, le résultat retourné par notre opérateur sera nécessairement une approximation $\widetilde{f(x)}$ du résultat exact $f(x)$.

Dans les temps héroïques de l'informatique, cette approximation imposée par le format a souvent induit un certain laxisme, aboutissant à des approximations bien moins précises que ce que le format permettait. Une constante de ce mémoire, de ce point de vue, est l'exigence de précision : on s'attachera au chapitre 3 à renvoyer le meilleur résultat possible. Pour nos opérateurs matériels du chapitre 2, on saura toujours donner une borne assez fine de l'erreur des opérateurs construits, ce qui permettra par exemple de savoir construire le plus petit opérateur respectant une borne d'erreur donnée.

On voit que la problématique de l'évaluation des fonctions se ramène pour beaucoup à une bonne gestion des erreurs de calcul dans les algorithmes. Si la problématique est ancienne, une contribution importante et transversale de nos travaux dans ce domaine est un travail sur l'automatisation de ces calculs d'erreurs. Ces aspects sont traités au chapitre 4.

CHAPITRE 1

Généralités

On les y retrouva 8 jours après, fort amaigris mais rayonnants : ils avaient, au fond de leur tube, découvert, pour faire l'addition des nombres entiers de 1 chiffre, une méthode facile déduite des propriétés de la spirale logarithmique.

Christophe, *La famille Fenouillard*

Ce mémoire n'a pas la prétention de couvrir toute la problématique de l'évaluation d'une fonction. En particulier, le lecteur intéressé par les aspects algorithmiques est invité à se rapporter à l'ouvrage de référence de Muller [112]. Ce chapitre en expose juste les grandes lignes, et définit le vocabulaire et les notations utilisées par la suite.

Le présent mémoire vient essentiellement en aval du livre de Muller, au sens où nos contributions sont essentiellement du domaine de l'implémentation ou du raffinement de méthodes existantes. Nous cherchons à construire des circuits ou des programmes de qualité pour l'évaluation d'une fonction. Pour cela, il faut d'abord définir ce que l'on entend par qualité. Le plus souvent, différentes mesures de la qualité seront antagonistes, et il faudra gérer les compromis qui apparaissent.

1.1 Les données du problème

Comme la plupart des implémentations, celle d'une fonction devra minimiser une certaine *fonction de coût* sous certaines *contraintes*. Voici un bref survol de ces différents aspects.

D'abord, on se donne le plus souvent une contrainte de *précision*, exprimée

– soit en terme d'erreur absolue de l'approximation par rapport à la fonction :

$$\delta = \widetilde{f(x)} - f(x),$$

– soit en terme d'erreur relative (sous condition que $f(x) \neq 0$) :

$$\epsilon = \frac{\widetilde{f(x)} - f(x)}{f(x)}.$$

Ensuite, lorsqu'on écrit du logiciel, on a des contraintes sur les formats des nombres manipulables (entiers de 32 ou 64 bits, flottants simple ou double précision), et des contraintes

plus subtiles — et plus variables d'un processeur à l'autre [22, 4]— sur le nombre d'opérateurs disponibles, leur capacité à fonctionner en parallèle, la performance de la hiérarchie mémoire, etc. Comme fonction de coût, on pourra s'intéresser au temps d'exécution (mesuré en général en cycles), mais aussi à la taille du code, à la mémoire utilisée, etc.

Lorsqu'on construit du matériel, on a moins de contraintes, notamment sur le format des nombres : on pourra utiliser des additionneurs taillés au plus juste pour des additions d'entiers de 17 ou 42 bits si besoin est. Chaque opérateur a alors un coût en terme de *surface* et en terme de *délai* (mesuré en nanosecondes) qui est une fonction de la précision de l'opérateur. Ces métriques seront présentées plus en détail en 2.3.2. Il existe même souvent plusieurs compromis surface/délai pour chaque opérateur [63]. Par exemple, pour l'addition d'entiers de n bits, on peut utiliser un additionneur à propagation de retenue (surface et temps proportionnels à n) ou différentes variantes [154] d'additionneurs rapides (délai en $\log n$ mais surface en $n \log n$).

Enfin, le *temps d'exécution* sera pour certaines applications une contrainte, pour d'autres il sera une fonction à minimiser.

1.2 Une affaire de compromis

La recette générale pour obtenir des implémentations de qualité est donc d'exhiber une classe d'algorithmes qui semble *a priori* bien adaptée à la cible et aux contraintes du problème, et de bien définir les différents paramètres qui définissent cette classe (degré d'une approximation polynomiale, taille mémoire, précision des différents opérateurs, etc) avec les relations entre ces paramètres, les contraintes et les fonctions de coût du problème.

Ceci définit un *espace des paramètres* dans lequel on cherchera une bonne solution par différentes heuristiques : énumération exhaustive en 2.4, programmation linéaire sous contrainte [17], réduction de réseaux [14, 15], recuit simulé [92], heuristique *ad hoc* en 2.5, *etc*. Naturellement, ces heuristiques sont implémentées par des programmes.

Cette approche présente de plus l'avantage de s'adapter aux évolutions de la technologie. Pour les cibles matérielles du chapitre 2, il sera plus facile de recibler un générateur d'opérateurs (écrit en C++ ou en Java) que de recibler de manière optimisée un opérateur écrit dans un langage de description de matériel. De même, pour les fonctions logicielles du chapitre 3, il sera plus facile de réévaluer pour un nouveau processeur les compromis faits pour l'implémentation d'une fonction élémentaire, si cette recherche de compromis est mécanisée dans un ensemble de programmes capables de produire le code de la fonction mais aussi la preuve de son bon fonctionnement.

Certes, tout n'est pas automatisable, et l'intuition, la vue d'ensemble, l'expertise restent nécessaires. Un des aspects les plus passionnants de ces recherches est justement de formaliser des intuitions pour faire reculer la frontière entre ce qui est automatisable et ce qui reste du domaine de la poésie.

1.3 Les techniques d'évaluation d'une fonction

1.3.1 Approximation polynomiale ou rationnelle

Pour implémenter une fonction, on a à sa disposition un ensemble de briques de bases qui peut se ramener schématiquement à

– des valeurs précalculées stockées dans une mémoire,
– des additions,
– des multiplications.

Il est donc naturel de vouloir approcher la fonction par un polynôme, d'une part parce que le calcul des valeurs d'un polynôme n'utilise que ces opérations de base, d'autre part parce que cette approximation est possible dès lors que la fonction est dérivable jusqu'à un certain ordre, donc en pratique toujours pour les fonctions considérées (au moins par morceaux). Si l'on ajoute la division aux opérateurs de base, on peut utiliser une approximation par une fraction rationnelle, mais la division étant plus coûteuse que l'addition et la multiplication [67, 63], elle ne s'impose que pour des fonctions qui présentent une dérivée tendant vers l'infini, comme l'arcsinus, ou une asymptote, comme tangente et arctangente [100, 112].

Le paramètre principal de l'approximation par un polynôme est le degré du polynôme. La détermination du polynôme peut se faire classiquement par l'algorithme de Remez ou en utilisant les polynômes de Tchebychev [112]. La détermination du *meilleur* polynôme, compte tenu des contraintes de la cible, est une problématique active actuellement [17, 16, 18, 14]. Les travaux présentés dans ce mémoire ont contribué à motiver ces recherches, et en utilisent les résultats.

1.3.2 Réduction d'argument

Cette approximation polynomiale est si possible précédée d'une *réduction d'argument* qui ramène l'argument d'entrée dans un intervalle plus petit. L'intérêt est de permettre l'approximation par un polynôme de plus petit degré, donc moins coûteux à évaluer.

La réduction d'argument la plus simple est la partition de l'intervalle d'entrée en sous-intervalles, et l'utilisation d'un polynôme différent sur chaque sous-intervalle. Plus on a de sous-intervalles, plus ils seront petits, et donc plus les polynômes seront de petit degré (sous la même contrainte de précision). En contrepartie, il faudra stocker les coefficients de tous ces polynômes, ce qui nécessitera plus de mémoire[1]. C'est un exemple du type de compromis évoqué précédemment. Il peut se décliner en de nombreuses variations. Par exemple, on peut utiliser des intervalles de taille identique, ce qui permet de déterminer rapidement à quel intervalle appartient une entrée, mais n'optimise pas globalement la précision des polynômes, ou bien on peut utiliser des intervalles de tailles différentes, ce qui réduit le coût global de l'évaluation polynomiale au prix d'une dichotomie pour déterminer à quel intervalle appartient une entrée [93].

Pour les fonctions élémentaires, il existe de nombreuses techniques de réduction d'argument plus efficaces, exploitant des symétries et des identités remarquables spécifiques à chaque fonction. Le livre de Muller [112] donne un panorama de ces techniques, et le lecteur intéressé trouvera des variations utilisant les spécificités des processeurs Itanium dans les livres de Markstein [102] et Cornea *et al.* [22]. Une idée forte, popularisée par Gal [69] et Tang [137, 138, 139, 140], est de recourir sans complexe à de très grosses tables de valeurs précalculées.

Nous ne détaillerons pas plus avant ces techniques car notre contribution en la matière est minime. À notre connaissance, seule la réduction d'argument trigonométrique en une étape dans CRLibm, notre bibliothèque mathématique qui sera présentée au chapitre 3, est

[1] Defour s'est intéressé au coût de cette mémoire en terme de temps d'exécution dans les hiérarchies mémoires des systèmes actuels [38], mais cette étude est déjà datée.

originale, sans être révolutionnaire [1]. Il n'est d'ailleurs pas certain qu'elle soit toujours plus performante que les approches plus classiques en deux étapes.

1.3.3 Schémas d'évaluation polynomiale

Une fois obtenu un polynôme approchant la fonction et dont les coefficients sont des nombres qu'on sait manipuler en machine [17] (typiquement des *dyadiques*, des nombres de la forme $M \cdot 2^E$ où M et E sont des entiers), il reste à implémenter ce polynôme. Pour cela, il faut choisir un *schéma d'évaluation*, qui décrit l'ordre dans lequel les opérations sont réalisées.

Le schéma le plus simple est le schéma de Horner, qui évalue un polynôme de degré n par une séquence de n étapes comportant chacune une multiplication et une addition :

$$c_0 + x \times (c1 + x \times (c_2 + x \times (\cdots + c_n \times x) \cdots).$$

Ce schéma a de bonnes propriétés en terme de précision [11], mais sa structure séquentielle interdit d'exploiter le parallélisme exposé par les processeurs superscalaires actuels ou le matériel : le schéma de Horner ne peut réaliser qu'une opération à la fois. C'est un compromis lent et économe en matériel.

À l'opposé, l'expression du polynôme développée :

$$c_0 + c_1 \times x + c_2 \times x^2 + \cdots + c_n x^n$$

peut s'évaluer en parallèle en un temps logarithmique en le temps d'une opération, si l'on dispose d'assez d'opérateurs [2] (remarquons que l'expression développée précédente ne constitue pas un schéma d'approximation, il faut y ajouter un parenthésage, qui décrit par exemple comment sont calculés les x^i). Toutefois le coût total en terme d'opérations sera alors en $n \log n$. On a ici un compromis rapide mais peu économe.

Il existe enfin une gamme de compromis intermédiaires. La famille des schémas de Estrin [112] est une sorte de schéma de Horner parallélisé. Par exemple, une version du logarithme de CRLibm évalue un polynôme de degré 7 par le schéma suivant :

$$p(x) = \big((c_0 + c_1 \times x) + x^2 \times (c_2 + c_3 \times x)\big) \; + \; x^4 \times \big((c_4 + c_5 \times x) + x^2 \times (c_6 + c_7 \times x)\big)\,.$$

Ce schéma peut s'exécuter en parallèle :

```
x2=x*x;   p01=c0+c1*x;   p23=c2+x*c3;   p45=c4+x*c5;   p67=c6+x*c7;
x4=x2*x2;   p03=p01+x2*p23;   p47=p45+x2*p67;
p07=p03+x4*p47;
```

On a écrit sur chaque ligne des calculs qui peuvent être réalisés en parallèle. Le nombre total d'opérations est légèrement supérieur à celui d'une évaluation par Horner, mais sur un processeur récent, pipeliné voire superscalaire, l'exploitation du parallélisme offert par le pipeline compense largement. Par exemple, un processeur Itanium-II possède deux FMA (pour *fused multiply-and-add*, un opérateur qui calcule $a \times b + c$ avec un seul arrondi du résultat), chacun pipeliné en 4 cycles, c'est-à-dire qu'une opération dure 4 cycles pendant lesquels d'autres opérations indépendantes peuvent être lancées. Sur un tel processeur,

[2] Lorsqu'on construit du matériel, on peut le construire avec un parallélisme arbitraire, c'est pourquoi la méthode HOTBM, décrite en 2.5.2, part de cette expression développée.

- la première ligne du code précédent peut être lancée en trois cycles consécutifs (mettons les cycles 0, 1 et 2), et les résultats sont disponibles quatre cycles plus tard (cycles 4, 5 et 6).
- Le calcul de `x4` peut donc être lancé au cycle 4, le reste de la seconde ligne au cycle 6.
- La troisième ligne pourra être démarrée au cycle 10, et le résultat sera disponible au cycle 14.

Par comparaison, une évaluation de Horner aurait nécessité $7 \times 4 = 28$ cycles, soit le double. En effet, un seul des deux FMA aurait été utilisé, avec son pipeline rempli au quart seulement. Une étude quantitative plus générale de l'optimisation de l'évaluation par Estrin sur les processeurs Itanium a été réalisée par Harrison *et al.* chez Intel [75, 22].

Un inconvénient du schéma d'Estrin sur le schéma de Horner est que son erreur d'arrondi cumulée est beaucoup plus difficile à borner précisément. Notre contribution en la matière est d'avoir montré que ce problème était soluble dans les outils présentés au chapitre 4.

Tous les schémas d'approximation précédents laissent les coefficients du polynôme inchangés. Il existe enfin des schémas, par exemple le schéma de Knuth et Eve ou celui de Paterson et Stockmeyer [84] qui, au prix d'une réécriture plus radicale du polynôme, peuvent réaliser moins d'opérations que le schéma de Horner. L'idée est qu'un polynôme à racines multiples tel que $(x-1)^2$ peut s'évaluer en moins d'opérations qu'un polynôme arbitraire de même degré. On cherchera donc à ramener par changement de variables le polynôme initial à un polynôme à racines multiples. L'inconvénient est que les coefficients sont modifiés en profondeur. D'une part, lorsqu'on arrondira les nouveaux coefficients à des nombres manipulables en machine, on changera effectivement le polynôme. D'autre part, les ordres de grandeurs des nouveaux coefficients, donc les erreurs d'arrondi à l'évaluation, pourront être très différents du polynôme initial. Guillaume Revy a toutefois montré que ces difficultés pouvaient être gardées sous contrôle [122], et cela a permis d'économiser quelques cycles dans une des fonctions de CRLibm.

Ce bref survol des schémas d'évaluation n'a pas la prétention d'être exhaustif — en particulier les possibilités en terme de circuit ont à peine été évoquées, elles le seront au chapitre 2 — mais de montrer un exemple des compromis auxquels on est confronté lorsqu'on s'intéresse à l'évaluation des fonctions.

1.4 La maîtrise des erreurs

Il est naturel de vouloir évaluer la qualité d'une approximation utilisant l'une des techniques précédentes. Plus précisément, on voudra pouvoir borner l'erreur commise, qui est définie comme la différence entre le résultat calculé et le résultat idéal. Cette différence peut être absolue, ou relative au résultat idéal, comme déjà vu en 1.1.

La notion de «résultat idéal» doit également être formalisée : si pour la sortie ce sera clairement la valeur mathématique de la fonction, pour les valeurs intermédiaires du calcul ce sera moins évident. Cette question sera précisée au chapitre 4.

1.4.1 Les différentes sources d'erreur

Les erreurs qui font que le résultat calculé s'écarte du résultat idéal sont essentiellement de deux types :

- erreurs d'approximation, essentiellement pour nous l'approximation d'une fonction par un polynôme ;
- erreurs d'arrondi, apparaissant lors du calcul du polynôme dans chaque opération non exacte.

Certaines sources d'erreurs peuvent être considérées comme relevant de l'une ou l'autre catégorie. Par exemple,

- si une formule de Taylor donne une certaine erreur d'approximation, ses coefficients devront le plus souvent être arrondis ensuite à des nombres machine. Il est plus simple de compter ces arrondis dans l'erreur d'approximation du nouveau polynôme. C'est aussi plus précis : l'erreur maximale d'approximation de f par p sur un intervalle sera définie par une seule norme infinie $||f - p||_\infty$. On peut même chercher à obtenir directement l'approximation comme un polynôme à coefficients machine [17, 14].
- Lorsqu'une identité mathématique utilisée dans une réduction d'argument met en jeu un nombre transcendant comme e ou π, son implémentation utilise une valeur approchée de cette constante. À nouveau cette erreur sera comptée dans les erreurs d'approximation.
- La même situation se produit lorsque des valeurs sont tabulées, que ce soit en logiciel [139, 69] ou en matériel [134, 24, 124, 37, 52].

On voit qu'on préférera réserver le terme d'erreur d'arrondi à des arrondis dynamiques, qui dépendent de la valeur d'entrée, alors que les arrondis statiques, réalisés une fois pour toutes à la construction de l'opérateur, seront intégrés à l'erreur d'approximation.

Cette terminologie n'est du reste pas très importante : l'important est de n'oublier aucune source d'erreur. Le chapitre 4 montrera comment des techniques en lien avec la preuve formelle aident à y parvenir.

1.4.2 Pour des bornes d'erreur fines

Il peut paraître plus facile de fournir une borne plus lâche de l'erreur totale commise par un calcul, en sommant des surestimations grossières et en prenant de la marge. C'est d'ailleurs comme cela que l'on pense *a priori* un algorithme et son implémentation. Un exemple en sera donné en 4.2. Toutefois, deux arguments plaident en faveur d'une borne la plus fine possible.

Le premier est qu'une borne lâche aura un impact en terme de performance : pour un circuit, cela signifiera qu'on a surdimensionné des opérateurs, des chemins de calculs, donc qu'on obtient un circuit plus gros et plus lent qu'il ne serait possible. Pour une fonction logicielle, on a le même effet, et il s'y ajoute dans le cas de l'arrondi correct un impact sur le temps de calcul moyen qui sera détaillé en 3.2. Enfin, si l'on implémente une fonction d'intervalle [86, 120, 79, 64, 35], c'est la finesse de l'intervalle retourné, autre mesure de performance, qui sera dégradée.

Le second est que dès lors qu'on veut une garantie, une validation, une preuve au sens mathématique, la difficulté est dans la combinatoire des cas à gérer, et non pas dans la finesse des bornes manipulées. Autrement dit, il ne sera pas tellement plus difficile d'obtenir une borne fine qu'une borne lâche.

1.4.3 Le vocabulaire des calculs d'erreurs

On utilisera souvent, comme unité d'erreur sur une valeur calculée, l'*ulp* (pour *unit in the last place*), qui est égal au poids du bit de poids le plus faible de la valeur en question. Par exemple, en virgule fixe, l'ulp d'un entier vaudra 1, l'ulp d'un nombre écrit en binaire $0,b_1b_2...b_8$ vaudra 2^{-8}.

La notion d'ulp existe aussi pour les valeurs flottantes, mais avec plusieurs définitions [111] qui diffèrent au voisinage des puissances de 2, lorsque l'exposant, donc l'ulp, changent de valeur.

Un intérêt de la notion d'ulp est de permettre des définitions générales de différentes *relations d'arrondi*.

Un nombre représentable est l'*arrondi au plus près* d'un réel s'il est le nombre représentable le plus proche de ce réel. L'arrondi au plus près est presque une fonction des réels vers les nombres représentables. Les seuls réels ayant deux arrondis possibles sont ceux qui sont exactement au milieu de deux nombres représentables. En définissant une règle univoque dans ce cas, l'arrondi au plus près devient une fonction, ce qui entre autres simplifiera la gestion des erreurs. Un tel choix est fixé par la norme IEEE-754, qui régira tout le calcul flottant dans ce document. Nous y reviendrons en temps utile.

L'arrondi au plus près correspond à une erreur inférieure ou égale à un demi-ulp, avec en flottant quelques subtilités autour des puissances de 2 suivant la définition de l'ulp choisie [111].

Un arrondi plus relâché est l'arrondi *fidèle*. L'arrondi fidèle d'un réel peut être l'un des deux nombres qui encadrent ce réel. L'arrondi fidèle correspond à une erreur strictement inférieure à un ulp.

CHAPITRE 2

Méthodes d'approximation adaptées au matériel

LE PROFESSEUR
Écoutez-moi, Mademoiselle, si vous n'arrivez pas à comprendre profondément ces principes, ces archétypes arithmétiques, vous n'arriverez jamais à faire correctement un travail de polytechnicien. (...) à calculer mentalement combien font(...), par exemple, trois milliards sept cent cinquante-cinq millions neuf cent quatre-vingt-dix-huit mille deux cent cinquante et un, multiplié par cinq milliards cent soixante-deux millions trois cent trois mille cinq cent huit ?

L'ÉLÈVE, très vite.
Ça fait dix-neuf quintillions trois cent quatre-vingt-dix quadrillions deux trillions huit cent quarante-quatre milliards deux cent dix-neuf millions cent soixante-quatre mille cinq cent huit (...)

LE PROFESSEUR, stupéfait.
Mais comment le savez-vous, si vous ne connaissez pas les principes du raisonnement arithmétique ?

L'ÉLÈVE
C'est simple. Ne pouvant me fier à mon raisonnement, j'ai appris par cœur tous les résultats possibles de toutes les multiplications possibles.

Eugène Ionesco, *La leçon*

2.1 Introduction

L'objet de ce chapitre est de définir l'architecture interne d'un composant matériel dont l'interface externe est donnée par la figure 2.1.

FIG. 2.1 – Opérateur matériel d'évaluation de fonction élémentaire.

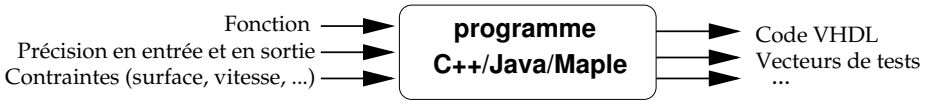


FIG. 2.2 – Générateur d'opérateurs pour les fonctions élémentaires.

L'entrée X et la sortie $\widetilde{f(X)}$ sont en général codées en virgule fixe sur w_I et w_O bits respectivement (une extension à la virgule flottante sera évoquée en 2.6). Les précisions auxquelles nous nous sommes intéressé sont celles qui sont utilisées en traitement du signal : de 8 à 32 bits. Ce ne sont toutefois pas les seuls paramètres d'un tel opérateur. On peut citer aussi :

- les intervalles d'entrée et de sortie : a-t-on besoin d'un sinus sur $[-\pi, \pi]$ ou sur $[0, \pi/4]$?
- la précision de l'approximation recherchée : veut-on l'arrondi correct, l'arrondi fidèle, une autre précision, voire une précision variable en fonction de l'entrée comme dans [55] ?
- La cible technologique (circuit intégré *full-custom*, circuit intégré utilisant des cellules précaractérisées et/ou des blocs des mémoires, circuit reconfigurable/FPGA, ...)
- les contraintes applicatives : doit-on optimiser en surface, en latence, en débit, ... ?

À ces paramètres externes à l'opérateur vont s'ajouter tous les paramètres internes des architectures utilisées, typiquement les tailles des différents chemins de données. Il n'est pas rare qu'une architecture possède plusieurs dizaines de tels paramètres internes.

En pratique, on ne cherchera donc pas à concevoir un opérateur directement : on s'intéressera plutôt à la conception de *générateurs d'opérateurs*, qui sont des programmes prenant en entrée le cahier des charges de l'opérateur recherché et calculant l'architecture la mieux adaptée, au terme d'une navigation dans l'espace des paramètres qui peut être assez complexe (et souvent heuristique).

2.2 Quelques applications

Nous exposons ici trois applications rencontrées qui montrent la variabilité des différents paramètres.

2.2.1 Traitement du signal

Les algorithmes utilisés en traitement du signal (*Digital Signal Processing* ou DSP) ont souvent besoin d'évaluer des fonctions en petite précision. C'est ainsi que l'on trouve dans la littérature des recettes pour, par exemple, contruire une approximation de la racine carrée sur 8 bits (dont 7 significatifs) par un découpage ad-hoc du mot d'entrée suivi par quelques additions et décalages [3]. La première référence connue à la méthode bipartite, qui est l'ancêtre des méthodes développées dans ce chapitre, est une approximation des fonctions trigonométriques pour les générateurs de fonctions utilisés en traitement du signal [134].

Les fonctions utilisées dans la littérature sont essentiellement des sinusoïdes, utilisées pour construire des harmoniques, et des fonctions algébriques (racines carrée et cubique et leurs inverses dans les radars, par exemple). Exponentielle et logarithme peuvent apparaître dans des applications de traitement du signal plus spécialisées, comme le filtrage de signaux provenant de capteurs à base de sources radioactives.

Une autre application est celle des générateurs de nombres pseudo-aléatoires [94] utilisés pour des simulations de type Monte Carlo mais aussi en recherche sur les techniques récentes de télécommunications telles les turbo-codes. On sait fabriquer des générateurs très simples ayant une distribution uniforme (leur architecture comporte typiquement un Xor, un décalage et une addition). Lorsqu'on a besoin d'une autre distribution des nombres (normale par exemple), une solution simple est d'utiliser un tel générateur uniforme en entrée d'une fonction (par exemple gaussienne) qui définit la distribution. Il existe aussi des algorithmes plus subtils, comme la méthode de Box-Muller qui transforme une distribution uniforme en une distribution normale par l'application d'un logarithme, une racine carrée, une rotation (sinus et cosinus) et deux multiplications [84].

Du point de vue de la précision de la sortie, la communauté du traitement du signal a l'habitude de manipuler des signaux bruités, et semble s'accomoder fort bien d'opérateurs de mauvaise qualité, ayant par exemple plusieurs bits de poids faibles non significatifs. Par contre, un critère de qualité important sera que l'erreur d'un opérateur se comporte comme un bruit blanc : si l'erreur d'un opérateur de sinus présente une périodicité dans un rapport entier de celle du sinus, elle introduira des harmoniques qui peuvent perturber de manière importante le signal résultat. Nos méthodes introduiront systématiquement de tels biais si un arrondi plus laxiste que le fidèle est demandé.

2.2.2 Initialisation des itérations quadratiques pour la division et la racine carrée

Il y a deux grandes classes d'algorithmes pour calculer la division et la racine carrée [62, 63] :

- Les algorithmes par récurrence sur les chiffres, parfois appelés SRT en hommage à Sweeney, Robertson et Tocher qui les ont publiés à peu près en même temps à la fin des années 50, généralisent l'algorithme «à la main». Ils utilisent une récurrence à base d'addition qui donne un chiffre à chaque itération. Si la base de numération utilisée est une puissance de 2 (en pratique 4, 8, 16 ou 32) on arrive à obtenir plusieurs chiffres binaires par itération, mais la convergence reste linéaire en le nombre de chiffres.
- Les itérations quadratiques ou de Newton-Raphson, étudiées par Goldschmidt [71] et Flynn [66], sont des variations autour de l'algorithme de Newton pour trouver le zéro d'une fonction. Elles mettent en œuvre des récurrences à base de multiplications qui doublent le nombre de chiffres exacts du résultat à chaque itération. Par rapport aux algorithmes SRT, il faut un nombre logarithmique d'itérations seulement, mais chaque itération est plus coûteuse puisqu'elle demande une ou deux multiplications.

Pour construire un diviseur isolé dans une technologie donnée, les algorithmes SRT sont plus simples et plus performants, mais pour construire le diviseur d'un processeur, les itérations quadratiques sont actuellement la solution préférée. En effet, le processeur contient de toutes manières un ou plusieurs multiplieurs, et le meilleur compromis performance/coût est obtenu par l'utilisation de cette ressource. Si les processeurs IA32 d'Intel ont traditionnellement utilisé l'algorithme SRT (dont une erreur d'implémentation fut à l'origine du bug [109] découvert par T. Nicely[1] dans le Pentium), les processeurs concurrents d'AMD utilisent des itérations quadratiques [81, 114]. De même, le jeu d'instruction IA64 utilisé dans l'Itanium ne comporte pas d'instruction de division ni de racine carrée, mais seulement des instructions qui permettent de démarrer une récurrence quadratique en logiciel.

[1] `http://www.trnicely.net/pentbug/pentbug.html`

Supposons en effet qu'on veuille calculer l'inverse d'un nombre en précision double étendue (64 bits de mantisse). Si l'on part d'une approximation initiale du quotient constante (par exemple 1/2), il faut $\log_2(64) = 6$ itérations pour obtenir cette précision finale (plus une pour l'arrondi correct). Si par contre on part d'une approximation de l'inverse du quotient déjà précise à 16 bits, on économise les quatre premières itérations. C'est ainsi que le K7 contient un opérateur matériel produisant une approximation de l'inverse « précise au moins à 14,94 bits» [114], alors qu'un processeur Itanium contient une table (accessible par l'instruction machine `frcpa`) fournissant une approximation de l'inverse «précise à $2^{-8.86}$» [102, 22].

La précision spécifiée de ces approximation peut sembler étrange : elle est choisie au plus juste, en fonction de l'algorithme quadratique choisi et du matériel qui l'implémentera, pour des divisions performantes dans les précisions importantes (simple, double et double étendue). C'est une des raisons pour laquelle nos générateurs peuvent produire des opérateurs pour une précision arbitraire, pas uniquement pour l'arrondi fidèle.

Les méthodes développées dans ce chapitre fournissent un rapport performance/coût matériel parfaitement adapté à ce type d'application (précision entre 10 et 24 bits).

2.2.3 Opérateurs logarithmiques

L'arithmétique logarithmique, ou LNS, code un nombre réel positif X par son logarithme L_X, habituellement en base 2. Ce logarithme est lui-même codé en virgule fixe sur w_E bits de partie entière et w_F bits de partie fractionnaire.

Cette représentation est concurrente à la virgule flottante : pour le même nombre de bits, un représentation LNS offre une précision et une dynamique comparables à la représentation flottante [6, 21]. Plus précisément, la plage des nombres représentables par un système LNS(w_E, w_F) est comparable à celle offerte par la virgule flottante à w_E bits d'exposant et w_F bits de mantisse.

L'intérêt majeur du système logarithmique est la simplicité des opérateurs pour la multiplication, la division et la racine carrée, réalisées respectivement par addition, soustraction, et décalage à droite des logarithmes des opérandes :

$$\begin{aligned} L_{X\times Y} &= L_X + L_Y, \\ L_{X\,/\,Y} &= L_X - L_Y, \\ L_{\sqrt{X}} &= \tfrac{1}{2} L_X. \end{aligned}$$

Par contre, effectuer une addition ou une soustraction en LNS est une opération bien plus complexe qu'en virgule flottante, car elle nécessite l'évaluation des deux fonctions non linéaires $f_\oplus$ et $f_\ominus$ définies comme suit et tracées à la figure 2.3 (X et Y sont ici des nombres positifs et tels que $X > Y$) :

$$\begin{aligned} L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\ &= L_X + f_\oplus(L_Y - L_X), \qquad \text{avec} \quad f_\oplus(r) = \log_2(1 + 2^r), \\ L_{X-Y} &= \log_2(2^{L_X} - 2^{L_Y}) \\ &= L_X + f_\ominus(L_Y - L_X), \qquad \text{avec} \quad f_\ominus(r) = \log_2(1 - 2^r). \end{aligned}$$

Il est aussi possible d'utiliser une réduction d'argument pour se ramener uniquement aux fonctions 2^x et $\log_2(x)$. Cette technique réduit la surface mais augmente le chemin critique [91].

On voit que dans tous les cas, le calcul de l'addition LNS nécessite l'évaluation de fonctions non linéaires. Les précisions mises en jeu sont également de l'ordre de 8 bits (traitement de signial audio) à 24 bits (précision comparable à la virgule flottante simple précision).

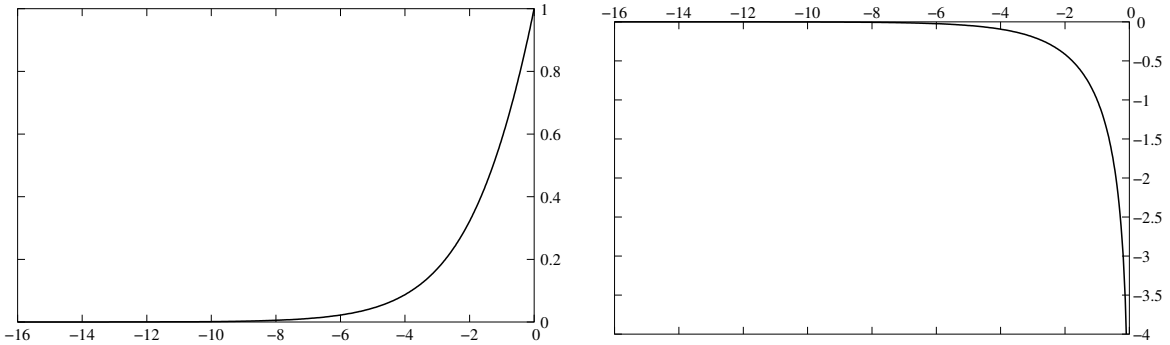
2.3 Les cibles technologiques

2.3.1 ASIC ou FPGA

Par matériel, on entend dans ce chapitre deux technologies bien distinctes. La première est celle des circuits intégrés classiques (*application-specific integrated circuit* ou ASIC). Nous n'avons toutefois pas réalisé de circuit ASIC incluant les travaux présentés dans ce chapitre. Une raison en est leur coût : la réalisation d'un tel circuit se chiffre en dizaines de milliers d'euros, et le prix des outils pour les concevoir est en proportion. Une autre raison est que les opérateurs que nous concevons n'ont pas vocation à être des circuits intégrés complets, mais plutôt de petites parties d'un circuit plus grand.

Pour valider nos algorithmes et nos architectures, nous les avons donc implémentés sur des circuits reconfigurables, ou FPGA pour *field-programmable gate array*. Ce sont des circuits dans lesquels on peut implanter par programmation, donc relativement facilement et rapidement, des circuits logiques arbitraires. Les FPGA permettent un compromis coût-flexibilité intermédiaire entre circuit ASIC, très performant mais très coûteux à concevoir et peu flexible, et microprocesseur, peu coûteux et très flexible mais moins performant. Les FPGA sont des composants du commerce produits en grande série, donc immédiatement disponibles et relativement peu coûteux. L'implantation d'un circuit dans un FPGA présentera des performances inférieures à celles qu'aurait une réalisation sous forme d'ASIC, mais très supérieures à une émulation logicielle. Ces circuits sont donc économiquement intéressants pour de petites séries d'applications. Pour les grandes séries, le coût initial supérieur d'un circuit intégré spécifique pourra être amorti. Leurs applications vont aussi du prototypage rapide [128] à l'accélération de calcul numérique [145, 44, 12, 146, 90, 7, 101, 144, 72, 46, 59].

Concrètement, ces FPGA sont constitués d'un grand nombre de *blocs logiques* dont la fonction est programmable (figure 2.5), reliés par un *réseau de connexion* également confi-



(a) La fonction $f_{\oplus}$ (b) La fonction $f_{\ominus}$

FIG. 2.3 – Les fonctions mises en jeu dans l'arithmétique logarithmique.

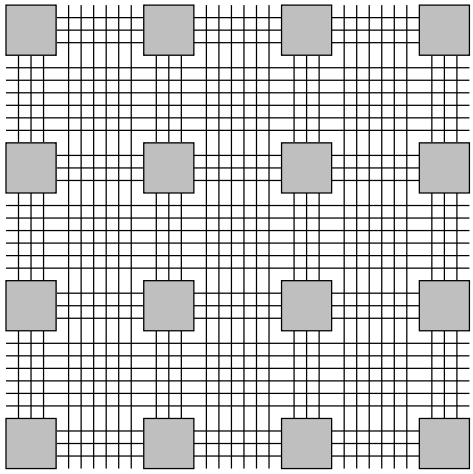


FIG. 2.4 – Vue schématique d'un FPGA d'architecture «insulaire».

gurable (figure 2.4). Ces schémas seront détaillées dans la suite.

2.3.2 Les métriques du matériel et les compromis architecturaux

On mesure la qualité d'un circuit à sa vitesse, à la quantité de matériel qu'il occupe, et parfois à sa consommation électrique. Chacune de ces métriques se décline selon plusieurs modalités

- La surface d'un circuit ASIC est à terme la taille du rectangle de silicium qui le contiendra[2]. Lors de la conception, on l'évalue en termes d'*équivalent-portes*. Pour les FPGA, nos opérateurs arithmétiques ayant vocation à être des sous-circuits dans des applications plus conséquentes, la surface qu'ils occupent doit également être minimisée. Pour une application complète, la situation est légèrement différente : soit le circuit rentre dans le FPGA, soit il ne rentre pas. On mesure la surface en «îles» effectivement occupées. Pour les FPGA de la firme Xilinx que nous utilisons principalement, nous suivons la terminologie de la marque : une île s'appelle une *slice*. Il est à noter que certains des FPGA récents incluent de petits multiplieurs (18 bits par 18 bits) en plus des *slices* [152]. Si nous les utilisons, ils seront intégrés dans la mesure de la quantité de matériel occupée.
- La vitesse est mesurée à l'identique pour les technologies ASIC et FPGA : on peut considérer la *latence* (en secondes), qui est le délai total entre l'entrée de la donnée et la sortie du résultat. Son inverse est la fréquence de fonctionnement. Lorsqu'on pipeline l'opérateur, le délai augmente en principe peu, la fréquence augmente et la latence se mesure alors en *cycles*, qui sont aussi une mesure de la profondeur du pipeline.

Pour des applications plus spécialisées, d'autres métriques pourront être prises en compte. Par exemple, le niveau d'émissions électromagnétiques est un critère de qualité pour un circuit devant prendre place dans un téléphone portable. Pour un circuit utilisé en cryptogra-

[2]Le coût unitaire de fabrication d'un gros circuit intégré est plus que proportionnel à cette surface. En effet, le processus de fabrication du circuit n'est pas parfait : il subsiste une certaine densité de défauts au mm^2, et il faudra jeter tous les circuits contenant un défaut. Ainsi, plus les circuits fabriqués sont gros, plus le pourcentage qu'il faudra jeter est élevé. Le coût des circuits ainsi perdus s'ajoute au coût des circuits qu'on garde, proportionnel lui à la surface.

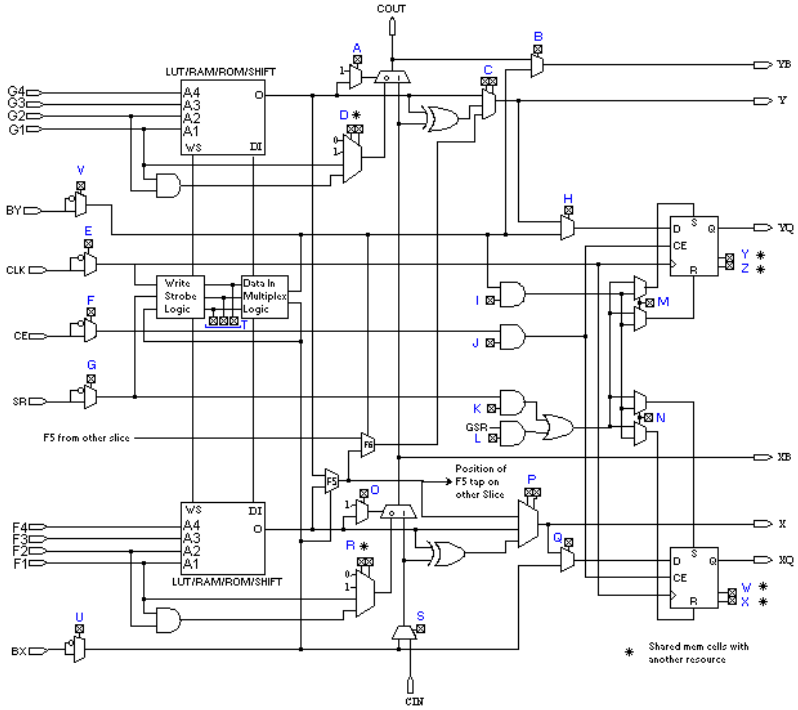


FIG. 2.5 – Vue schématique d'une île (ou *slice*) de FPGA Xilinx Virtex, d'après la documentation constructeur. Une petite boîte munie d'une croix représente un bit de configuration.

phie, un critère capital est que l'information privée qu'il contient ne puisse être déduite de mesures externes (variations de sa consommation ou de ses émissions, par exemple). Ces aspects n'ont pas concerné nos travaux.

2.3.3 Les briques de base et leur coût

La suite de cette section détaille les briques de bases disponibles dans chacune de ces technologies, c'est-à-dire les opérateurs que nous pouvons espérer trouver facilement en bibliothèque, et que nous n'aurons donc pas à construire. Nous insistons sur leurs coûts et performances relatives, car ces métriques influencent le choix des algorithmes et l'exploration architecturale. Ainsi, si certaines des techniques que nous avons développées s'appliquent parfaitement à la technologie ASIC comme à celle des FPGA, les paramètres optimaux de l'architecture pourront être très différents d'une cible à l'autre.

Portes logiques

D'un point de vue théorique, la porte logique NON-ET est suffisante pour construire toutes les autres. Toutefois, pour construire un circuit intégré dédié performant, il est plus efficace de se donner un ensemble plus conséquent de portes, incluant les portes NON, ET, OU et OU exclusif (ou XOR), leurs versions complémentées et à une entrée complémentée, leurs versions à trois ou quatre entrées, les multiplexeurs et démultiplexeurs, etc. Les coûts respectifs de ces différentes portes varient suivant leurs caractéristiques électriques (notamment le *fan-out*, qui mesure le nombre de portes qu'une porte peut alimenter), lesquelles sont également liées à leur vitesse et consommation électrique [135, 136]. Ainsi, une bibliothèque de base pour une technologie ASIC donnée pourra comporter plusieurs centaines de portes.

Toutefois, il est possible de nos jours d'ignorer cette complexité – et c'est même sans doute le plus sage. Les outils de synthèse associés aux langages de description de matériel comme VHDL et Verilog sont extrêmement performants lorsqu'il s'agit de compiler une description de matériel vers une technologie donnée. Pour les circuits relativement petits que sont les opérateurs arithmétiques décrits dans ce chapitre, on fait confiance à l'outil de synthèse pour leur implémentation optimale. Et si l'on continue de mesurer les surfaces en termes de «nombre de portes équivalent», c'est l'outil qui fournit ces chiffres.

Lorsque l'on cible des circuits reconfigurables, la porte de base est typiquement la petite mémoire adressable (*look-up table* ou LUT) à 4 entrées, et l'on raisonne et mesure en termes de LUT. Mais ici aussi, le circuit effectivement ciblé est plus complexe, comme l'indique la figure 2.5 : on y voit qu'il y a deux LUT par *slice*, et que les LUT sont entourées d'une circuiterie plus ou moins orthogonale qui permet d'implanter efficacement des sous-circuits courants comme des additionneurs, des mémoires... Nous y reviendrons. De plus, les contraintes sur le routage sont très variables d'un FPGA à l'autre – et pas toujours documentées. Ici aussi, on s'en remet donc également à l'outil qui compile notre description VHDL.

En conclusion, quelle que soit la technologie, on écrira des AND et des XOR en VHDL, et on les dessinera sur les figures représentant nos architectures, mais ces opérations pourront apparaître de manière très différente – et, on l'espère, plus efficace – dans l'implementation physique de ces architectures, et ce en fonction de la technologie visée.

Les quatre opérations

L'addition de deux entiers est l'opération arithmétique de base à partir de laquelle on peut construire les autres (multiplications, divisions, ...). Comme les opérateurs logiques, elle peut s'exprimer par un simple + en VHDL. Lorsque l'on cible les FPGA, c'est la bonne approche : l'architecture des FPGAs modernes comporte une logique dédiée à la propagation des signaux de retenue (pour les Virtex, de l'entrée `CIN` à la sortie `COUT` sur la figure 2.5). Cette circuiterie permet d'implémenter l'additionneur le plus simple, qui propage les retenues comme on le fait à la main. Ce n'est pas le plus rapide asymptotiquement : son temps est en $O(n)$ où n est le nombre de bits à additionner, alors qu'il existe tout un bestiaire d'additionneurs en $O(\log(n))$ [154, 63]. Toutefois ceux-ci sont moins performants en pratique jusqu'à des valeurs de n très grandes, tout en occupant plus de surface. La raison en est qu'ils utilisent le routage général du FPGA (les canaux de fils visibles sur la figure 2.4), qui est comparativement plus lent que le routage dédié d'un `COUT` au `CIN` suivant. En effet, chaque signal allant d'une *slice* à une autre à travers le routage général doit traverser les nombreux transistors – plusieurs dizaines – qui assurent la programmabilité du routage [27].

Lorsqu'on considère l'implémentation d'une addition pour un circuit ASIC, on a beaucoup plus de choix, de l'additionneur à propagation de retenue (vitesse $O(n)$, surface $O(n)$) aux additionneurs rapides (vitesse $O(\log(n))$, surface $O(n\log(n))$ avec toute une gamme de compromis [154]). Les outils de synthèse offrent en général un choix d'additionneurs.

Dans les architectures que nous développons, on a le plus souvent à additionner plus de deux termes. Il existe des techniques spécifiques pour ce problème de *multiaddition*, utilisant des représentations internes redondantes, comme la retenue conservée *(carry-save)* [63]. Les additions dans une telle représentation sont pleinement parallèles, donc très rapides (pas de propagation de retenue), et l'opérateur correspondant est petit. L'inconvénient est qu'il faut ensuite convertir le résultat en la représentation binaire classique, ce qui nécessite un additionneur rapide. Toutefois, pour les applications qui nous intéressent, il arrive souvent que l'opérateur destinataire du résultat s'accomode également d'une représentation redondante. C'est le cas pour les approximations initiales pour les diviseurs, par exemple [24]. Dans ce cas, on économise l'additionneur rapide final.

Toutefois, pour les raisons déjà citées, l'utilisation de représentations redondantes est rarement avantageuse lorsqu'on cible les FPGA. Dans ce cas, une multi-addition arborescente naïve est la meilleure solution.

Les opérateurs plus gros (multiplication, division) présentent une diversité architecturale encore plus grande – le lecteur intéressé se rapportera aux ouvrages de référence [85, 63].

Il est toujours intéressant de savoir spécialiser un opérateur pour son contexte. Ainsi, nous utiliserons des opérateurs *ad-hoc* lorsqu'il s'agira de multiplier par une constante [19, 95, 34, 147]. Lorsqu'on cible les FPGA comportant de petits multiplieurs, leur taille spécifique orientera la conception de nos algorithmes [8].

Mémoires et tables

Nos architectures feront souvent appel à des tables de valeurs précalculées. En fait, l'architecture la plus simple pour implémenter une boîte noire pour une fonction telle que représentée à la figure 2.1 consiste à tabuler les 2^{w_I} valeurs en entrée. Cette approche est praticable (et pratiquée, par exemple pour les cœurs trigonométriques vendus par Xilinx [153]) pour des valeurs de w_I ne dépassant pas 16.

Les outils de synthèse pour ASIC fournissent des générateurs de mémoires mortes aboutissant à des architectures extrêmement compactes et optimisées en fonction de la technologie. Pour les FPGA, les mémoires mortes de l'architecture sont implémentées par des mémoires vives dans le circuit cible. Si l'on compare le coût d'une addition donnée et d'une mémoire morte donnée, on constatera donc qu'une grosse mémoire est comparativement plus coûteuse dans un FPGA. Toutefois, cette généralité doit être nuancée. D'une part, les FPGA récents offrent des blocs de mémoire comme ils offrent de petits multiplieurs. On est alors contraint par la taille de ces blocs, et l'on cherchera à adapter l'architecture à cette contrainte. D'autre part, les petites mémoires peuvent être implémentées par des arbres de LUT ce qui offre une flexibilité supplémentaire : en effet, les outils pourront appliquer des techniques d'optimisation logique [125] au contenu de la mémoire, ce qui peut réduire la taille dans un facteur considérable – mais dépendant des valeurs stockées [30]. Ces techniques ne sont pas applicables lorsqu'on cible un générateur de mémoire morte en ASIC ou un bloc mémoire dans un FPGA.

2.4 Méthodes d'ordre 1

Cette section détaille les différentes méthodes pour construire un opérateur utilisant une approximation de la fonction par des segments de droite, ou en termes plus mathématiques par un développement de Taylor d'ordre 1. Le nombre en entrée écrit en binaire comme un mot de w_I bits $X = \sum_{i=p}^{p+w_I-1} 2^i x_i$ peut se décomposer en deux sous-mots A et B de respectivement α et β bits : $X = A + 2^{-\beta}B$ (voir la figure 2.8). La valeur de la fonction à évaluer, $f(X) = f(A + 2^{-\beta}B)$, peut alors être approchée par un développement de Taylor à l'ordre 1 :

$$f(A + 2^{-\beta}B) \approx f(A) \; + \; 2^{-\beta}Bf'(A). \tag{2.1}$$

Les articles dont nous nous sommes inspirés [134, 24, 110, 131] exprimaient le problème en termes de telles approximations de Taylor. Nous avons préféré raisonner sur des représentations graphiques qui sont plus parlantes, et surtout permettent une maîtrise plus fine des erreurs d'approximations.

2.4.1 La méthode bipartite

Cette méthode a été développée indépendamment par Sunderland *et al.* pour les fonctions trigonométriques [134] et, dix ans plus tard, par Das Sarma et Matula pour la fonction $1/x$ [24]. Elle a été généralisée par Schulte and Stine [124, 131] puis Muller [110]. Dans son principe, cette méthode approche la fonction par des segments de droite, comme illustré par la figure 2.6.

Les 2^α segments (16 sur la figure) sont indicés par les α bits de poids forts du mot d'entrée (représenté figure 2.8). Pour chaque segment, une valeur initiale est tabulée, et les autres valeurs sont interpolées par l'addition d'une correction calculée à partir des $\beta = w_I - \alpha$ bits restant.

L'idée de la méthode bipartite est de regrouper les 2^α intervalles d'entrée en 2^γ intervalles plus grands ($\gamma < \alpha$). Sur la figure 2.6, $\gamma = 2$. Sur chacun de ces grands intervalles, on fixe une pente constante : là est l'approximation bipartite. C'est cette pente constante qui est tabulée pour chaque grand intervalle. Ainsi, il y a seulement 2^γ tables de correction, chacune

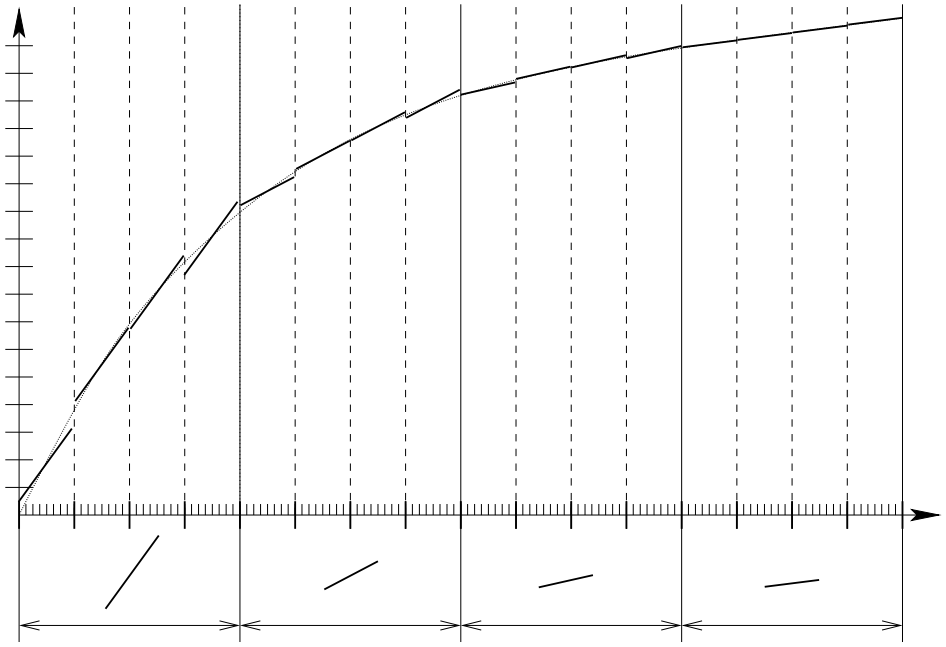


FIG. 2.6 – L'approximation bipartite ($\alpha = 4$, $\beta = 2$, $\gamma = 2$).

contenant 2^β valeurs de correction. Au total, on n'a besoin de tabuler $2^\alpha + 2^{\gamma+\beta}$ valeurs, au lieu de $2^{w_I} = 2^{\alpha+\beta}$ pour une tabulation simple de la fonction. L'architecture correspondante est décrite par la figure 2.7.

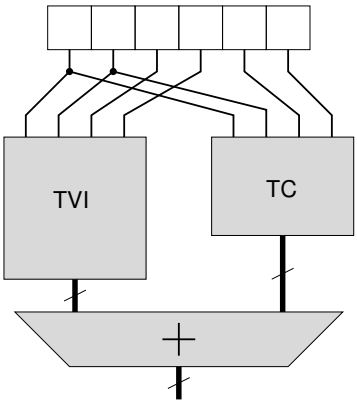


FIG. 2.7 – Une architecture bipartite.

Dans toute la suite, nous dénoterons *table de valeur initiale* ou TVI la table indicée par A, et par *table de corrections* ou TC la table qui contient les corrections.

Le raisonnement en termes d'approximations de Taylor indique que pour $\gamma \approx \beta \approx \alpha/2$, l'erreur d'approximation reste «dans des limites acceptables» [110, 131]. Une de nos contribution sera de déterminer les valeurs optimales (au sens de la minimisation de la taille totale des tables) de α, β et γ.

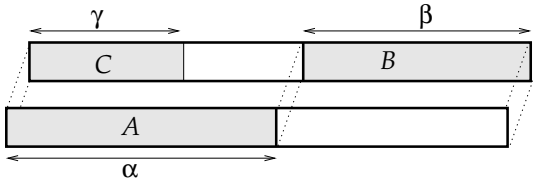


FIG. 2.8 – Décomposition bipartite du mot d'entrée.

2.4.2 Une optimisation : l'exploitation de la symétrie

Une contribution de Schulte et Stine [124] fut de remarquer que les segments approchant la courbe, s'ils sont bien centrés de manière à minimiser la distance à la courbe, présentent une symétrie (approximative, à l'ordre 2) par rapport à cette courbe. La figure 2.9, qui est un agrandissement de la figure 2.6, décrit cette situation. Cette remarque permet de diviser par deux la taille des tables de correction : si l'on stocke dans la TVI la valeur de la fonction au milieu de l'intervalle, alors la TC contient les corrections pour une moitié de la courbe, et les corrections pour l'autre moitié se déduisent par symétrie. Le calcul des opposés se fait par quelques portes XOR, dont le coût est largement compensé par la diminution de moitié de la TC.

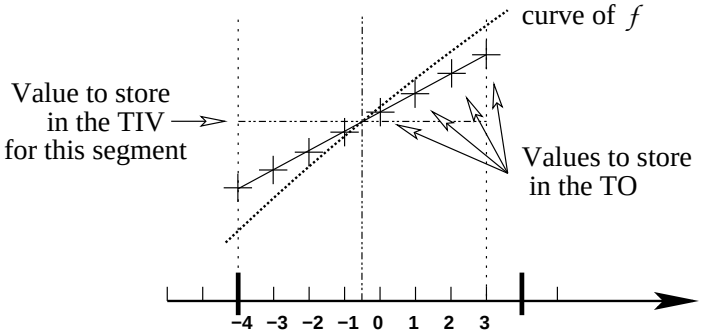


FIG. 2.9 – Symétrie sur un segment.

2.4.3 Les méthodes multipartites

Travaux précédents

Schulte et Stine [131] ont ensuite remarqué que la taille totale des TC peut encore être réduite : en effet cette table contient une fonction linéaire $\mathrm{TC}(CB) = s(C) \times B$, où $s(C)$ est la pente du segment. On peut décomposer B à nouveau, comme illustré par la figure 2.10, en m sous-mots B_i de tailles β_i, pour $0 \leq i < m$:

$$B = B_0 + 2^{\beta_0} B_1 + ... + 2^{\beta_0+\beta_1+...+\beta_{m-2}} B_{m-1}.$$

Définissons $p_0 = 0$, et $p_i = \sum_{j=0}^{i-1} \beta_j$ pour $i > 0$. La fonction calculée par la TC est alors :

$$\begin{aligned} \mathrm{TC}(CB) &= s(C) \times \sum_{i=0}^{m-1} 2^{p_i} B_i \\ &= \sum_{i=0}^{m-1} 2^{p_i} s(C) \times B_i \\ &= \sum_{i=0}^{m-1} 2^{p_i} \mathrm{TC}_i(CB_i). \end{aligned} \tag{2.2}$$

Par conséquent, on peut distribuer cette TC en m tables plus petites $\mathrm{TC}_i(CB_i)$, ce qui réduira encore la taille totale (la symétrie existe toujours dans les tables obtenues). Ceci coûte $m-1$ additions. Cette amélioration implique donc deux compromis :

- un sur le coût, entre la réduction de la taille des tables et l'augmentation du nombre des additionneurs ;
- un sur la précision : l'équation (2.2) n'est pas une approximation, mais conduira à plus d'erreurs de discrétisation lors du remplissage des tables.

Il y a encore une amélioration possible. En effet, on constate dans l'équation (2.2) que pour $j > i$, le poids du dernier bit significatif de TC_j vaut $2^{p_j - p_i}$ fois le poids du dernier bit significatif de TC_i. En d'autres termes, TC_i est bien plus précise que TC_j. Peut-être est-elle trop précise ? On peut donc essayer de réduire la précision de certaines TC_i en y stockant une approximation plus grossière de la pente $s(C)$, obtenue en ignorant les bits les moins significatifs de C. Ceci aboutira à des tables encore plus petites.

Cette idée d'équilibrer les précisions des différentes tables est présente, pour un cas spécial, dans un article de Muller [110]. Sa méthode *multipartite* décompose le mot d'entrée en $2p+1$ sous-mots X_1,...,X_{2p+1} de tailles identiques. Un travail sur les formules de Taylor montre qu'on obtient des erreurs d'approximation du même ordre de grandeur pour une table adressée par X_{2p+1} et une pente déterminée seulement par X_1, une table adressée par X_{2p} et une pente déterminée par X_1X_2, et en général une table adressée par X_{2p+2-i} et les i sous-mots de poids forts. Cette analye d'erreur reste bien loin d'une application architecturale : la décomposition est rigide, les majorations des erreurs d'approximation sont grossières, et les erreurs d'arrondi ne sont pas considérées.

La décomposition multipartite généralisée

Si l'on cherche à fusionner les approches de Schulte et Stine et de Muller, le plus simple est de considérer toutes les décompositions possibles du mot d'entrée (les deux approches précédentes étant des cas particuliers), et de mener une étude architecturale autour de chaque décomposition. Plus précisément, une *décomposition multipartite* est définie par les paramètres suivants (voir la figure 2.10) :

- Le mot d'entrée est d'abord coupé en deux sous-mots A et B de tailles respectives α et β, avec $\alpha + \beta = w_I$.
- Le mot de poids fort, A, adresse la TVI.
- Le mot de poids faible, B, sera utilisé pour adresser $m \geq 1$ tables de corrections.
  - B est décomposé à son tour en m sous-mots B_0,..., B_{m-1}, B_0 étant celui de poids le plus faible.

- Un sous-mot B_i commence au p_i-ème bit, et se compose de β_i bits. On a $p_0 = 0$ et $p_{i+1} = p_i + \beta_i$.
- Le sous-mot B_i est utilisé pour adresser la TC_i, avec une pente adressée par un sous-mot C_i de A composé des γ_i bits les plus significatifs de A.

– On notera $\mathcal{D} = \{\alpha, \beta, m, (\gamma_i, p_i, \beta_i)_{i=0\ldots m-1}\}$ une telle décomposition.

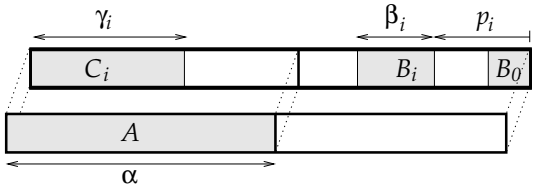


FIG. 2.10 – Une décomposition multipartite.

Exploration architecturale

L'exploration architecturale qui permet de déterminer l'architecture optimale selon les critères recherchés (surface, vitesse ou autre) consiste à énumérer l'ensemble des décompositions, et à évaluer pour chaque décomposition l'architecture qu'elle représente. La décomposition ne définit pas entièrement l'architecture (représentée figure 2.11). Il faut aussi déterminer le nombre g de bits de garde qui permettront à cette architecture de fournir la précision requise (nous supposerons ici que c'est l'arrondi fidèle, c'est-à-dire un résultat qui s'éloigne de la valeur mathématique de moins d'un ulp, ou 2^{-w_O} si l'intervalle image est $[0, 1[$). C'est l'objet de l'analyse d'erreur que nous survolons maintenant.

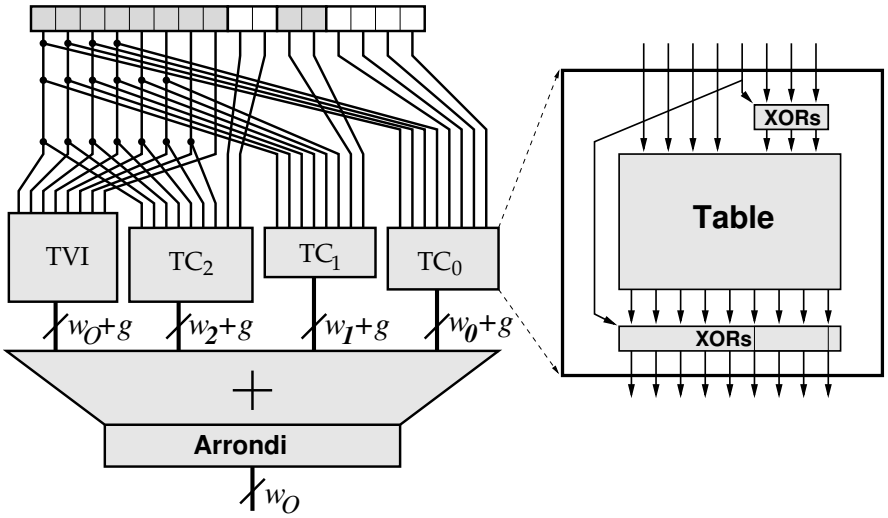


FIG. 2.11 – Une architecture multipartite, avec exploitation de la symétrie.

On remarque d'abord que l'arrondi final peut faire perdre un demi-ulp[3].

[3] Au lieu de 2^{-w_O-1}, l'arrondi final peut voir son erreur abaissée à $2^{-w_O-1} - 2^{-w_O-g-1}$ si l'on utilise une petite

Reste un budget d'un demi-ulp pour assurer l'arrondi fidèle. Pour une décomposition donnée et une fonction f donnée sur un intervalle donné, il est possible de calculer exactement les valeurs optimales à stocker dans chaque TC_i et l'erreur d'approximation maximale en résultant, en fonction de f et des paramètres de cette TC_i, à savoir (γ_i, p_i, β_i). Le calcul est assez simple, puisque la TC_i contient un segment de droite, et que le graphe de f est parfaitement connu. On tient compte dans ce calcul de la symétrie. Ensuite, il ne reste qu'à sommer les erreurs d'approximation de chaque TC_i pour obtenir l'erreur d'approximation totale pour la décomposition multipartite $\mathcal{D}$.

On ne retient que les décompositions pour lesquelles cette erreur est inférieure à un demi ulp. Ceci constitue le premier filtre de l'exploration architecturale. Il faut noter que des branches entières de l'énumération peuvent être coupées à ce stade : si une décomposition donnée est incapable de fournir l'arrondi fidèle, toutes celles qui par exemple ont un γ_i plus petit que celle-ci sont moins précises, et donc ne sont pas évaluées.

Pour toutes les décompositions retenues, il existe une architecture multipartite offrant l'arrondi fidèle : il suffit de calculer avec une précision intermédiaire assez grande pour que les erreurs d'arrondis intermédiaires deviennent assez petites pour entrer dans le budget d'erreur. Autrement dit, il suffit de prendre g assez grand. Naturellement, plus g est grand, plus les additionneurs sont coûteux, et plus la mémoire requise par l'architecture est importante.

Nous donnons dans [37] une formule qui donne g en fonction de l'erreur d'approximation de la décomposition. Une fois g connu, l'architecture correspondant à une décomposition est parfaitement déterminée, et on peut lui appliquer une fonction de coût qui permet de retenir la meilleure. Cet algorithme d'exploration est résumé par la figure 2.12, dans laquelle les flèches représentent des dépendances de données.

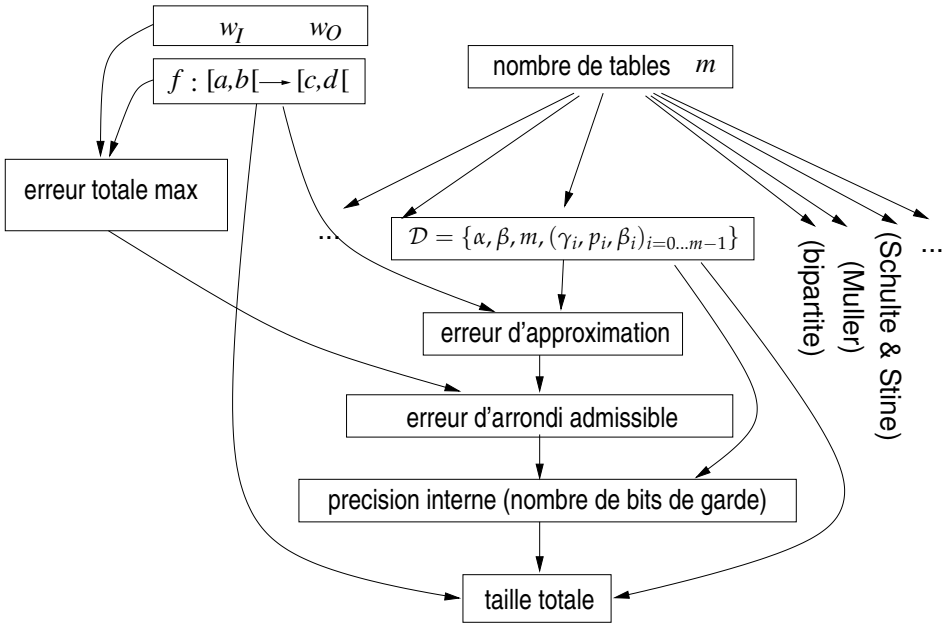


FIG. 2.12 – L'exploration des architectures multipartites.

astuce due à DasSarma et Matula [24] et expliqué en détail dans [37] mais que nous ne reprendrons pas ici)

En pratique, la taille totale des tables donne une bonne estimation du coût de l'architecture en terme de surface. En terme de délai, une règle générale est que les architectures les plus petites sont aussi les plus rapides, mais on peut préférer limiter le nombre de tables, donc le nombre d'additions. On retient ainsi quelques candidats que l'on synthétise, et pour lesquels on obtient des estimations très précises du coût et de la performance.

Observations et conclusions

L'ensemble de la méthodologie est détaillée dans [37] et a été automatisée, exploration comprise, jusqu'à la production de code VHDL. Nous avons également étudié la possibilité d'utiliser des techniques d'optimisation logique pour réduire encore la surface des tables [30].

Des exemples que nous avons traités, on peut retenir que la méthode est adaptée à des précisions de 12 à 24 bits, que les valeurs de g restent modestes (entre 2 et 6), et que l'optimal consiste souvent en une décomposition dans laquelle tous les β_i sont égaux à 2 (la plus petite valeur possible) ou 3. Les tables 2.2 et 2.3 donnent quelques résultats de décompositions optimales pour les fonctions décrites par la table 2.1.

On observe aussi que la méthode multipartite ainsi généralisée ne nécessite en général pas plus de volume de table que si l'on avait implémenté une approximation d'ordre 1 au moyen de multiplieurs, alors que nous faisons l'économie de ceux-ci. Plus précisément, on peut définir $\alpha_{\min}$ la plus petite valeur de α telle qu'il est possible de définir une approximation linéaire par morceaux fidèle de f par une partition de l'intervalle d'entrée en segments de taille 2^{α}. Ainsi, $2^{\alpha_{\min}}$ est un minorant du nombre d'entrées pour toute table de coefficients d'une approximation d'ordre 1 fidèle. Or, on constate que, pour la plupart des fonctions testées, la décomposition multipartite optimale a pour paramètre $\alpha = \alpha_{\min}$.

Enfin, une dernière observation, moins mathématique, est que, au moins lorsque l'on cible les FPGA, une architecture multipartite est toujours plus rapide, pour les précisions supérieures à 8 bits, qu'une tabulation simple de la fonction – qui pourtant ne réalise pas d'addition. Ce n'est pas si surprenant au vu des facteurs de compression des tables en question par la méthode multipartite : En première approximation, le temps de traversée d'une grosse table, implémentée en LUT et en routage général, est proportionnel à la racine carrée de sa surface [152]. La compression de cette surface obtenue (d'un facteur 100 par rapport à la table simple pour 16 bits) compense donc largement le coût des additions.

Toutefois, c'est une situation inhabituelle en architecture de ne pas avoir de compromis entre temps et surface. Cela a motivé nos travaux pour passer à des approximations d'ordre 2 [48], puis d'ordres supérieurs [52], jusqu'à voir apparaître un tel compromis. La suite survole

Fonction	Intervalle d'entrée	Intervalle de sortie	w_I	w_O
sin	$[0, \pi/4[$	$[0, 1[$	16	16
2^x	$[0, 1[$	$[1, 2[$	16	15
$1/x$	$[1, 2]$	$[1/2, 1]$	15	15

TAB. 2.1 – Les fonctions testées. Il peut y avoir un bit implicite en entrée et en sortie, c'est pourquoi les valeurs de w_I et w_O pour une précision de 16 bits sont données.

f	m	α	β	γ_i	β_i	g	tables	taille	taille selon [131]
sin	1	10	6	5	6	1	$17.2^{10} + 6.2^{10}$	23552	32768
	2	8	8	6 5	4 4	3	$19.2^8 + 10.2^9 + 6.2^8$	11520	20480
	3	8	8	7 5 4	2 3 3	2	$18.2^8 + 9.2^8 + 7.2^7 + 4.2^6$	8064	17920
	4	8	8	6 6 5 5	2 2 2 2	3	$19.2^8 + 10.2^7 + 8.2^7 + 6.2^6 + 4.2^6$	7808	na
2^x	1	10	6	5	6	1	$16.2^{10} + 6.2^{10}$	22528	24576
	2	8	8	7 5	3 5	2	$17.2^8 + 9.2^9 + 6.2^9$	12032	14592
	3	8	8	7 6 4	2 3 3	2	$17.2^8 + 9.2^8 + 7.2^8 + 4.2^6$	8704	13568
	4	8	8	7 6 5 4	2 2 2 2	2	$17.2^8 + 9.2^8 + 7.2^7 + 5.2^6 + 3.2^5$	7968	na
$1/x$	1	10	5	7	5	1	$16.2^{10} + 6.2^{11}$	28672	24576
	2	9	6	7,6	3,3	3	$18.2^9 + 9.2^9 + 6.2^8$	15360	16896
	3	9	6	8,7,5	2,2,2	2	$17.2^9 + 8.2^9 + 6.2^8 + 4.2^6$	14592	15872

TAB. 2.2 – Meilleures décompositions multipartites pour des opérandes de 16 bits

ces recherches, avec moins de détails car ceux-ci sont disponibles dans la thèse de Jérémie Detrey [47].

2.5 Méthodes d'ordre supérieur

2.5.1 Travaux antérieurs

Après les approximations linéaires, les approximations polynomiales sont une idée naturelle pour la construction d'opérateurs matériels en virgule fixe. Cette idée a d'ailleurs été utilisée avant nous pour différentes applications particulières :

- Piñeiro *et al.* [119] utilisent la forme développée d'un développement d'ordre 2 par morceaux de la fonction (sur chaque intervalle, $f(X) \approx a_0 + a_1 X + a_2 X^2$) et la projettent relativement directement en matériel. L'architecture obtenue comporte donc trois tables de coefficients, une unité de mise au carré, et un arbre de multiplication qui réalise aussi l'addition finale. Ce travail cible quelques fonctions spécifiques (de la forme $f(X) = X^n$), mais l'approche est en principe tout-à-fait générale.
- Chez Defour *et al.* [42], un développement d'ordre 5 par morceaux est simplifié, ses termes sont regroupés et les termes qui peuvent être négligés le sont. L'architecture obtenue comporte des tables, des additions et de petits multiplieurs, «petit» signifiant que l'une des entrées est de taille $w_I/5$ bits seulement si bien que leurs surface et délai sont comparables à ceux de quelques additions.
- Parmi les nombreuses implémentations de l'addition/soustraction en LNS présentée en 2.2.3, on trouve des approximations d'ordre 0 [142], d'ordre 1 [99] et d'ordre 2 [21, 91]. Ces articles utilisent toutefois des techniques spécifiques aux fonctions $f_\oplus$ et $f_\ominus$.

On trouve de plus dans la littérature quelques tentatives de méthodes plus générales, s'appliquant à une fonction arbitraire et à des précisions arbitraires :

- Une méthode due à Hassler et Tagaki [76] utilise des sommes de produits partiels de la manière suivante. On approche d'abord la fonction par un polynôme $\sum A_i X^i$ (l'article original utilise un développement de Taylor d'ordre arbitraire). On écrit ensuite l'entrée X sous forme de sommes de bits pondérées, $X = \sum x_j 2^{-j}$. On fait de même pour

f	m	α	β	γ_i	β_i	g	tables	taille
sin	1	15	9	7	9	3	$27.2^{15} + 11.2^{15}$	1245184
	2	13	11	10 6	4 7	3	$27.2^{13} + 13.2^{13} + 9.2^{12}$	364544
	3	12	12	10 9 6	3 4 5	4	$28.2^{12} + 15.2^{12} + 12.2^{12} + 8.2^{10}$	233472
	4	12	12	10 10 8 7	2 2 4 4	4	$28.2^{12} + 15.2^{11} + 13.2^{11} + 11.2^{11} + 7.2^{10}$	201728
	5	12	12	10 10 9 7 6	2 2 2 3 3	4	$28.2^{12} + 15.2^{11} + 13.2^{11} + 11.2^{10} + 9.2^{9} + 6.2^{8}$	189440
	6	12	12	10 10 9 8 7 5	2 2 2 2 2 2	4	$28.2^{12} + 15.2^{11} + 13.2^{11} + 11.2^{10} + 9.2^{9} + 7.2^{8} + 5.2^{6}$	190016
2^x	1	15	9	8	9	1	$24.2^{15} + 9.2^{16}$	1376256
	2	13	11	10 8	5 6	2	$25.2^{13} + 12.2^{14} + 7.2^{13}$	458752
	3	12	12	11 9 7	3 4 5	3	$26.2^{12} + 14.2^{13} + 11.2^{12} + 7.2^{11}$	280576
	4	12	12	11 10 8 8	2 3 3 4	3	$26.2^{12} + 14.2^{12} + 12.2^{12} + 9.2^{10} + 6.2^{11}$	234496
	5	12	12	11 10 9 8 8	2 2 2 3 3	3	$26.2^{12} + 14.2^{12} + 12.2^{11} + 10.2^{10} + 8.2^{10} + 5.2^{10}$	211968
	6	12	12	11 10 9 8 8 8	2 2 2 2 2 2	3	$26.2^{12} + 14.2^{12} + 12.2^{11} + 10.2^{10} + 8.2^{9} + 6.2^{9} + 4.2^{9}$	207872
$1/x$	1	15	8	9	8	5	$28.2^{15} + 13.2^{16}$	1769472
	2	14	9	11 8	3 6	3	$26.2^{14} + 12.2^{13} + 9.2^{13}$	598016
	3	13	10	12 10 8	2 3 5	4	$27.2^{13} + 14.2^{13} + 12.2^{12} + 9.2^{12}$	421888
	4	13	10	11 11 10 9	2 2 3 3	4	$27.2^{13} + 14.2^{12} + 12.2^{12} + 10.2^{12} + 7.2^{11}$	382976
	5	13	10	11 11 10 9 8	2 2 2 2 2	5	$28.2^{13} + 15.2^{12} + 13.2^{12} + 11.2^{11} + 9.2^{10} + 7.2^{9}$	379392

TAB. 2.3 – Meilleures décompositions multipartites pour des opérandes de 24 bits

chaque coefficient A_i, que l'on écrit comme une somme de puissances de 2. On peut alors distribuer chaque terme $A_i X^i$. On obtient ainsi une grosse somme de produits de certains des x_j pondérés par des puissances de 2. On peut simplifier cette somme de différentes manières. Par exemple un produit de x_j est en fait un ET logique, donc on a $x_j^k = x_j$. Une seconde approximation consiste à négliger certains de ces termes, de manière à pouvoir regrouper les termes restant par paquets dans lesquels certains des x_j n'apparaissent pas : les sommes correspondantes peuvent être rangées dans des tables.

C'est une idée très séduisante, puisqu'elle définit un espace de recherche très vaste en termes mathématiques très simples et très faciles à manipuler. Par exemple, on obtient directement une expression analytique de l'erreur maximale due à la seconde approximation. Le problème est qu'on sombre assez vite dans des heuristiques qui perdent toute élégance dès lors qu'on cherche à naviguer dans cet espace. C'est le cas de l'heuristique décrite dans l'article original. Il est d'ailleurs remarquable que cette heuristique trouve presque toujours une décomposition multipartite – qui ne peut donc qu'être moins performante que celle que choisit notre méthode multipartite généralisée, d'autant que l'idée de la symétrie en est absente.

Si l'on veut en tirer une morale générale, ce sera qu'il faut trouver, lorsqu'on fait de l'exploration architecturale, le juste compromis entre la richesse de l'espace exploré et la possibilité de l'explorer de manière raisonnablement exhaustive en un temps raisonnable.

– Un exemple opposé est donné par Lee *et al.* [92] qui présentent une technique d'exploration architecturale des approximations d'ordre supérieur en utilisant une architecture simpliste d'évaluation du schéma de Horner. Ils réalisent une exploration archi-

tecturale exhaustive dans un espace très restreint, et l'optimal trouvé dans cet espace n'est certainement pas optimal dans un sens plus général.

2.5.2 La méthode HOTBM

La méthode HOTBM (pour *Higher Order Table-Base Method* est une évolution naturelle de la méthode multipartite. L'approche est la même :

- On définit un espace architectural autour de l'approximation d'ordre supérieur, puis on navigue dans cet espace en cherchant l'architecture optimale en terme de ressources sous contraintes de précision.
- On utilise une forme développée du polynôme pour arriver à une architecture sous la forme d'une somme de termes. Cela permettra d'exprimer plus de parallélisme que la forme de Horner.
- On définit pour chaque terme des sous-architectures paramétrées, de manière à pouvoir évaluer précisément les différents termes d'erreur en fonction de ces paramètres
- On exploite systématiquement les propriétés mathématiques de chaque terme (symétrie, continuité)

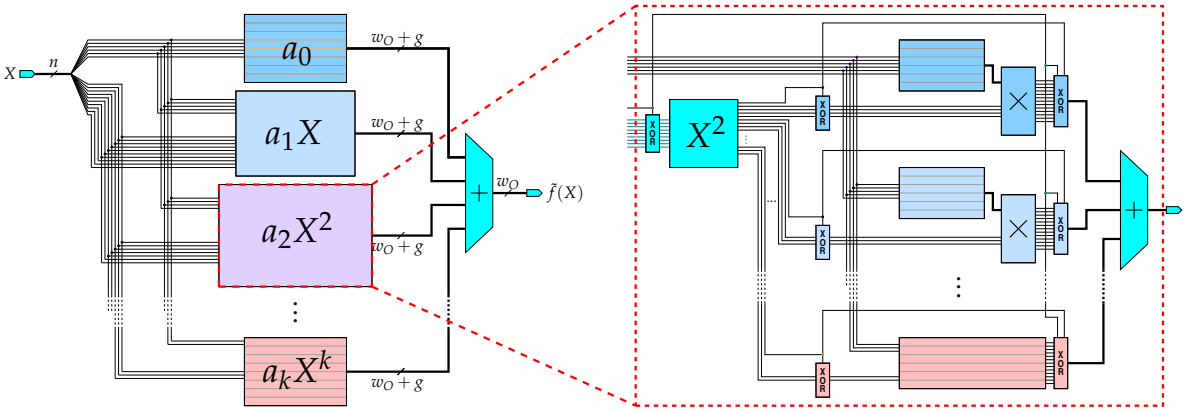


FIG. 2.13 – Une architecture HOTBM.

Cette méthode a fait l'objet d'une publication [52] et est décrite dans tous les détails dans la thèse de Jérémie Detrey [47]. Je ne la décrirai donc pas plus avant ici. La figure 2.14 donne un exemple des résultats obtenus, et montre que les méthodes d'ordre supérieur permettent des opérateurs très rapides pour des précisions jusqu'à 32 bits.

Nous nous étonnions en conclusion de 2.4.3 que les opérateurs obtenus par la méthode multipartite étaient toujours plus rapides lorsqu'ils étaient plus petits, ce qui ne correspond pas à l'idée que l'on se fait d'un compromis. Comme l'illustre la figure 2.14, l'utilisation d'approximations d'ordre supérieur fait bien apparaître un compromis entre surface et délai : une approximation d'ordre supérieur donne un opérateur plus petit mais plus lent.

2.5.3 Applications aux opérateurs logarithmiques

Notre bibliothèque d'opérateurs matériels FPLibrary utilise les méthodes d'approximation présentées dans ce chapitre pour l'évaluation des fonctions $f_\ominus$ et $f_\oplus$ définies en 2.2.3. Le lecteur intéressé trouvera tous les détails dans [55] et dans la thèse de Jérémie Detrey [47].

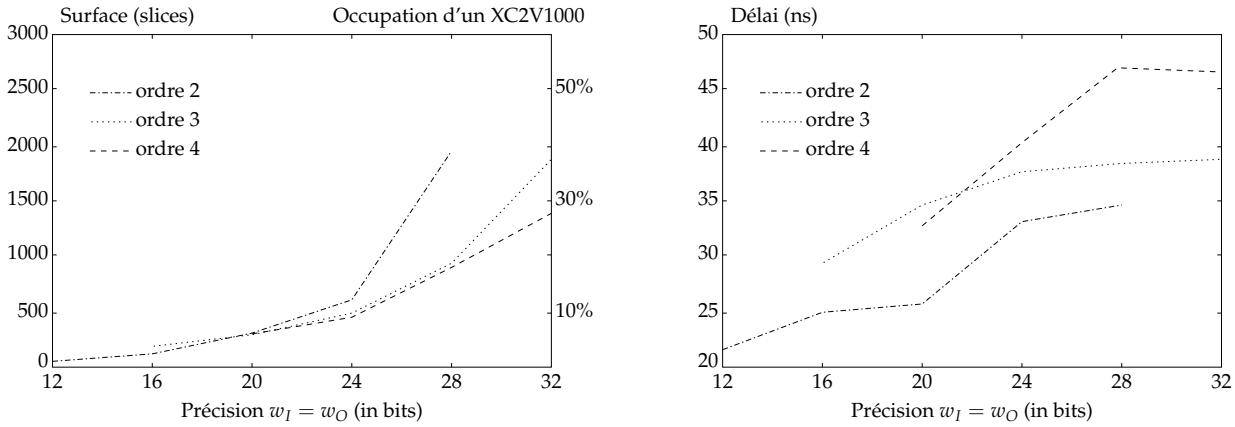


FIG. 2.14 – Surface et délai d'un opérateur pour sin x en fonction de la précision et de l'ordre.

2.5.4 Perspectives sur l'évaluation de fonctions en virgule fixe

Nous avons aussi exploré différentes méthodes utilisant des additions avant les lectures de tables (méthodes ATA pour addition-table-addition, le terme est d'un article de Wong et Goto [150] que nous avons généralisé). Ce sont des méthodes d'ordre 1 ou 2, et elles s'avèrent moins performantes que la méthode multipartite. Elles sont décrites dans [37] par souci d'exhaustivité.

Nous avons toujours partitionné l'intervalle d'entrée en segments de taille identique et égale à une puissance de 2. Pour des fonctions très irrégulières, comme la fonction $f_\ominus$ de l'arithmétique logarithmique, il est intéressant de considérer des décompositions moins régulières. Pour cela, la littérature offre deux approches, synthéthisées par Lee *et al.* [93].

La première est une décomposition en morceaux de tailles différentes. Ceci complique l'adressage de la table, et un bon compromis est un découpage en morceaux qui sont des puissances de 2 successives, si bien que les tables de coefficients sont adressées par le résultat d'un *compteur de zéros en tête* du mot d'entrée (ou *LZC* pour *leading zero counter*).

La seconde est l'utilisation d'une segmentation à deux niveaux, où un premier niveau de table, adressé par les poids forts de l'entrée, donne un pointeur vers une seconde table adressée par les poids faibles. On trouve cette idée dans une architecture de Lewis pour l'arithmétique logarithmique [98], et dans la méthode bipartite indirecte de Matula [104].

Ces deux idées sont extrêmement intéressantes, et Lee *et al.* en proposent une synthèse très prometteuse, assortie d'une méthodologie d'optimisation. Par contre, l'architecture d'évaluation polynomiale utilisée dans [93] est assez primitive. De plus, les fonctions utilisées comme étude de cas dans cet article sont trop exotiques pour qu'il soit possible d'évaluer quantitativement l'intérêt de cette approche. Toutefois, c'est certainement une piste à explorer.

Outre ces questions de découpage d'intervalle, on peut encore explorer certaines pistes, en particulier la méthode d'Hassler et Takagi [76] déjà évoquée. Cette méthode est très générale, très simple et très ouverte. Sa limite est qu'elle utilise une représentation de laquelle ont disparu toutes les propriétés de haut niveau de la fonction (continuité, etc). En revanche elle peut s'appliquer à une fonction à deux arguments ou plus.

Les fonctions multivariées sont d'ailleurs sans doute le prochain objet auquel nous nous intéresserons. Des exemples utiles sont les normes, comme $\sqrt{x^2 + y^2 + z^2}$, la fonction

$\arctan(x/y)$, composée de deux fonctions chacune relativement coûteuse, et bien d'autres, en fait presqu'autant qu'il y a de sous-circuits composés de plusieurs blocs de calcul dans les circuits numériques... Il faudrait donc développer pour ce type de fonctions des méthodologies aussi générales que la méthode multipartite ou HOTBM. C'est un projet de recherche ambitieux mais dont les applications ne manqueront pas.

2.6 Fonctions élémentaires en virgule flottante pour FPGAs

Une application potentielle de nos générateurs d'opérateurs d'évaluation de fonction en virgule fixe est naturellement la construction d'opérateurs pour la virgule flottante. Nous avons ainsi avec J. Detrey exploré des architectures pour l'exponentielle, le logarithme et les fonctions sinus et cosinus. Ce fut un travail original, voire en avance sur son temps : la densité d'intégration sur FPGA ne permettait pas de l'envisager avant le XXIe siècle.

L'idée de départ était d'utiliser une réduction d'argument pour se ramener à de la virgule fixe. L'objet d'une réduction d'argument est toujours de se ramener dans un petit intervalle [112], nous cherchons en plus à nous ramener dans un intervalle où l'exposant est constant (ou ne peut prendre qu'un petit nombre de valeurs). La mantisse est alors techniquement un nombre en virgule fixe, et on peut réaliser la suite de l'évaluation par les méthodes vues précédemment.

2.6.1 Les degrés de liberté du matériel

En explorant ces idées avec J. Detrey, il est apparu rapidement qu'il était très profitable de s'éloigner des méthodes classiques utilisées par les bibliothèques mathématiques (`libm`) qui évaluent les fonctions élementaires en logiciel. La raison de fond en est la liberté qu'a le concepteur d'architecture par rapport au concepteur de logiciel mathématique : alors que le dernier n'a à sa disposition que quelques opérateurs de base, de taille fixe, le premier peut construire exactement les opérateurs dont il a besoin.

Voici quelques points par lesquels ce travail s'est écarté des techniques utilisées dans les `libm`.

- Alors qu'une `libm` utilise essentiellement du calcul en virgule flottante, nous réussissons à utiliser uniquement du calcul entier ou en virgule fixe.
- En contrepartie, il faut se soucier de la normalisation du résultat final. Celle-ci est automatique dans une `libm`, puisque les opérateurs flottants utilisés renvoient toujours un résultat normalisé[4].
- Il faut utiliser des réductions d'argument inédites, car l'objectif est de se ramener à de la virgule fixe, et de surcroît sur peu de bits.
- On utilise la flexibilité des méthodes vues dans ce chapitre en ce qui concerne la borne d'erreur en sortie.
- Le calcul d'erreur est en virgule fixe aussi, ce qui est plus simple. Nos opérateurs sont fidèles et retournent l'arrondi correct dans une large majorité des cas.
- On voit apparaître des multiplications par des constantes, sur lesquelles le projet Arénaire a travaillé [95, 34].

[4] Le problème se pose tout de même lorsqu'on doit construire un résultat dénormalisé dans une `libm`. Par exemple, la partie du code d'une exponentielle qui gère les cas dénormalisés en sortie est traditionnellement complexe et inélégante.

Pour ces raisons, ce travail [54], que j'envisageais au départ comme essentiellement applicatif, est devenu tout à fait original.

Nous ne connaissons que deux tentatives avant la nôtre, un sinus [117] et une exponentielle [58]. Les deux sont de simples transpositions d'algorithmes développés pour le logiciel qui n'exploitent pas la flexibilité du FPGA. Les performances de nos opérateurs sont meilleures d'un ordre de grandeur en espace *et* d'un ordre de grandeur en temps.

2.6.2 L'exemple du logarithme

Le calcul du logarithme commence par une réduction d'argument qui est exactement celle qui est donnée par Markstein [100] et également utilisée dans la bibliothèque CRLibm (voir aussi la section 3.1 p. 39).

Soit $1,F$ une mantisse avec son 1 implicite (la taille de F est de w_F bits), et $X = 2^{E-E_0}\,1,F$ un nombre flottant en entrée. La réduction d'argument et la reconstruction se font comme suit :

$$\log(2^{E-E_0}\,1,F) = \begin{cases} \log(1,F) + (E-E_0)\cdot\log 2 & \text{si } 1,F \in [1,\sqrt{2}[, \\ \log\left(\dfrac{1,F}{2}\right) + (1+E-E_0)\cdot\log 2 & \text{si } 1,F \in [\sqrt{2},2[. \end{cases} \tag{2.3}$$

On fait deux cas, car utiliser uniquement le premier conduirait à une perte catastrophique de précision (cancellation) lors de la reconstruction lorsque X est proche de 1 par valeurs inférieures : on aurait une mantisse $1,F$ proche de 2, dont le logarithme serait proche de $\log(2)$, auquel la reconstruction ajouterait $-\log(2)$ pour obtenir le logarithme recherché, proche de zéro. La décomposition proposée contourne se problème en recentrant l'intervalle de l'argument réduit autour de 1.

Il faut remarquer que cette réduction d'argument, même avec deux cas, est exacte : la décomposition en mantisse et exposant est bien sûr exacte, et la division par 2 ainsi que l'addition de 1 peuvent facilement être implantées exactement en binaire. Le nombre $\sqrt{2}$, qui fait la frontière entre les deux cas, devra être approché, mais cela ne pose pas de problème : n'importe quelle approximation fera l'affaire tant que l'on reste suffisamment loin de 1, puisque les deux alternatives de (2.3) sont des identités sur $1,F \in [1,2[$.

Ayant posé $M = 1,F$ ou $M = \dfrac{1,F}{2}$ selon le cas, il reste à calculer $\log M$ pour $M \in [\sqrt{2}/2, \sqrt{2})$. Nous présentons ici deux techniques. La première utilise HOTBM : elle est très rapide mais sa surface croît exponentiellement avec la précision. Elle ne peut donc guère dépasser la simple précision. La seconde technique utilise une réduction d'argument itérative, également à base de tables. Elle est plus lente, mais sa surface croît seulement quadratiquement, ce qui lui permet d'atteindre la double précision.

Utilisation de HOTBM pour le logarithme de la mantisse

On pourrait utiliser directement la méthode HOTBM pour ceci. Il faudrait toutefois alors un opérateur avec en gros $2w_F$ bits en sortie. En effet, au voisinage de 1, la normalisation du nombre flottant final peut se traduire par un décalage pouvant aller jusqu'à w_F bits de mantisse puisque $\log M$ se rapproche de zéro. Pour compenser, il faudrait donc doubler la précision en sortie de l'opérateur $\log M$, ce qui serait coûteux.

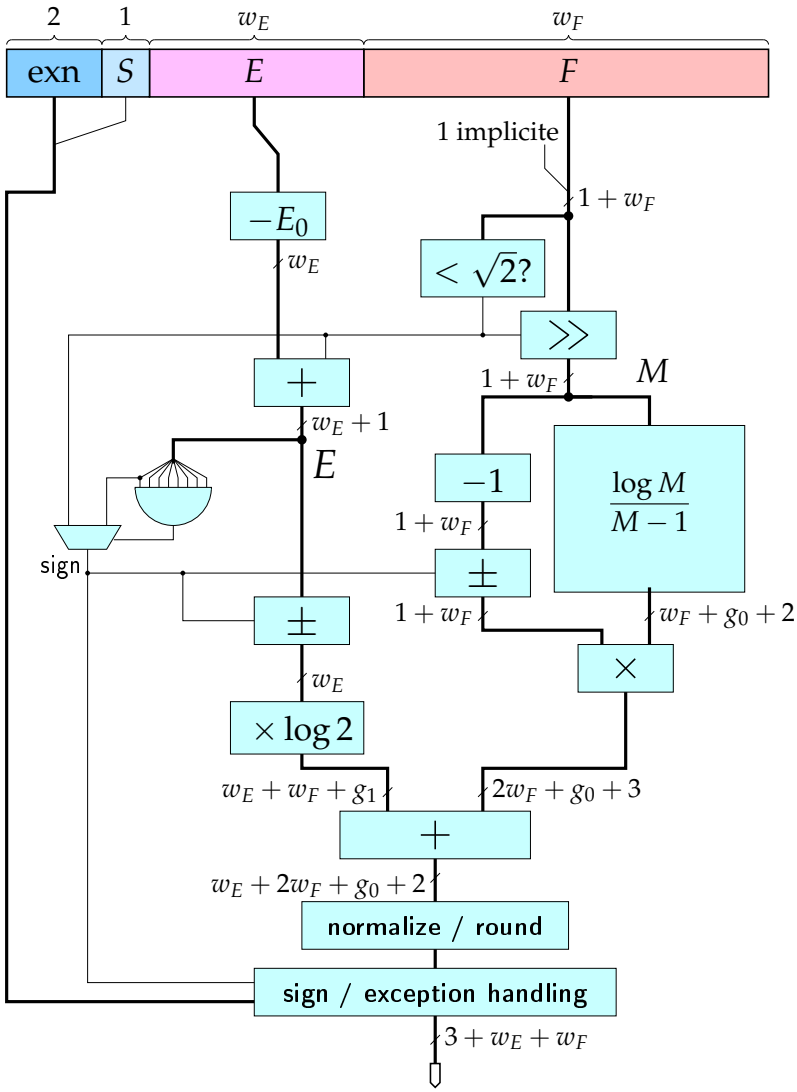


FIG. 2.15 – Architecture du logarithme flottant.

On obtiendra donc une précision meilleure à moindre coût en exploitant le développement limité du logarithme au voisinage de 1. On tabulera $\frac{\log M}{M-1}$ et on multipliera le résultat par $M-1$, qui est lui aussi calculé exactement. L'avantage est que $\frac{\log M}{M-1}$ ne passe pas par zéro : l'erreur absolue de la tabulation est ainsi transformée par la multiplication par $M-1$ en une erreur relative également constante. Un autre point de vue est plus architectural : on multipliera deux nombres précis ayant *grosso modo* la taille de la mantisse. En gardant tous les chiffres de ce produit, on aura un nombre de taille double, et s'il est très proche de zéro, sa normalisation (pour renvoyer un flottant dont le premier bit de la mantisse est un 1) pourra décaler de la taille de la mantisse et renvoyer néanmoins une mantisse entièrement significative. Le lecteur intéressé trouvera dans [54] une analyse d'erreur détaillée qui précise quantitativement cette réflexion.

L'architecture obtenue est décrite en détail sur la figure 2.15. Voici quelques commentaires explicatifs.

- Le format flottant utilisé comporte deux bits (notés *exn* sur la figure) codant les cas exceptionnels (infinis, zéros et NaN). Le traitement de ces cas est très simple pour le logarithme.
- Comme déjà dit, la frontière entre les deux cas n'a pas à être très proche de $\sqrt{2}$. Pour économiser du matériel, on va calculer la comparaison entre $1,F$ et cette constante sur le moins de bits possible. Combien exactement ? On va considérer les intervalles de M et $\log M$ (qui ne sont plus $M \in [\sqrt{2}/2, \sqrt{2}[$ ni $\log M \in [-\log 2/2, \log 2/2[$), et on va accepter qu'ils augmentent tant que la taille en bit des nombres correspondant n'augmente pas. Ainsi, l'économie de matériel due au relâchement de l'approximation de $\sqrt{2}$ ne se traduira pas en surcoût dans les étapes ultérieures du calcul.
- Le signe du résultat (qu'il faut naturellement aussi calculer) sera le signe de E tant que $E \neq 0$. Lorsque $E = 0$, il est donné par celui de $M-1$.
- La normalisation finale nécessite un compteur de zéros en tête *(Leading Zero Counter)*, une brique classique de l'architecture flottante, suivie d'un décalage qui peut aller jusqu'à w_F bits à gauche et w_E bits à droite.
- Le paramètre g_0 est un nombre de bits de garde. Il est déterminé par l'analyse d'erreur [54].

Cette architecture a été synthétisée et testée exhaustivement sur une carte munie d'un FPGA VirtexII reliée à un PC hôte, et ce pour toutes les valeurs de paramètres définies par $w_E \in [3,8]$ et $w_F \in [6,23]$. Ce test a montré que le résultat était toujours fidèle, et arrondi correctement pour 98% des nombres en entrée.

Le tableau 2.4 donne des résultats de synthèses pour différentes précisions. Le lecteur intéressé trouvera plus de détails dans [54]. La surface comme le délai dépendent peu de la dynamique (taille des exposants). En revanche, la surface croît exponentiellement, et le délai linéairement, avec la précision (taille de la mantisse) : c'est la signature de l'opérateur en virgule fixe sous-jacent. Pour les FPGA de 2006, les architectures proposées sont pertinentes jusqu'à la simple précision (24 bits de mantisse) et un peu au delà.

Utilisation d'une réduction d'argument itérative

De nombreux articles (en particulier [61, 149] et les variations autour de Cordic et de l'algorithme BKM présentées en détail dans [112]) utilisent des variations de l'itération suivante

pour calculer exponentielle et logarithme. Soient (x_i) et (l_i) deux suites de réels telles que $\forall i, e^{l_i} = x_i$. Il est possible de définir deux nouvelles sequences (x'_i) and (l'_i) comme suit : l'_0 et x'_0 sont choisis tels que $x'_0 = e^{l'_0}$, et

$$\forall i > 0 \left\{ \begin{array}{rcl} l'_{i+1} & = & l_i + l'_i \\ x'_{i+1} & = & x_i \times x'_i \end{array} \right. \tag{2.4}$$

Cette itération conserve l'invariant $x'_i = e^{l'_i}$, puisque $x'_0 = e^{l'_0}$ et $x_{i+1} = x_i x'_i = e^{l_i} e^{l'_i} = e^{l_i + l'_i} = e^{l'_{i+1}}$.

Si donc x est donné et que l'on veut calculer $l = \log(x)$, on peut prendre $x'_0 = x$, puis lire dans une table une suite (l_i, x_i) telle que la suite correspondante (l'_i, x'_i) converge vers $(0, 1)$. L'iteration en x'_i est calculée à i croissant, jusqu'à un certain n tel que x'_n soir suffisamment proche de 1. On peut alors calculer son logarithme par la série de Taylor $l'_n \approx x'_n - 1 - (x'_n - 1)^2/2$, voire même $l'_n \approx x'_n - 1$. Puis on en déduit le logarithme $\log(x) = l = l'_0$ par la recurrence (2.4) sur l'_i, à i décroissant de n à 0. Une approche duale permet de calculer l'exponentielle.

Dans cette double itération, on a une addition, une multiplication, et une valeur précalculée à extraire d'une table. Les nombreuses variations évoquées plus haut visent à minimiser le coût de la multiplication. Il faut en particulier contraindre les x_i à être de petits nombres – typiquement des chiffres dans la base de calcul, voire des bits. Les algorithmes varient aussi dans la manière dont les (x_i, l_i) sont choisis, en fonction de l'état courant de l'itération. Typiquement on veut des tables adressées par peu de bits.

Nous avons reciblé ces algorithmes pour les FPGAs [56], en utilisant le fait que la brique de base des FPGA est justement la table à 4 ou 5 entrées. On obtient des opérateurs plus petits mais plus lents que l'approche utilisant HOTBM, comme le montre le tableau 2.4. Surtout, la croissance de la surface est *grosso modo* quadratique avec la précision et non plus exponentielle, ce qui permet d'atteindre la double précision dans une fraction des FPGA courants (bien moins d'un dixième d'un FPGA haut de gamme de 2007). Autre comparaison, ils sont moins de deux fois plus gros que le multiplieur double précision mentionné dans [144].

Ces opérateurs sont également pipelinables, mais il reste encore à les pipeliner.

2.6.3 Performance : le FPGA surpasse le processeur

La comparaison des performances de nos fonctions en virgule flottante avec leurs homologues simple précision dans les processeurs est intéressante. Comme le montre le tableau 2.5, nos opérateurs ont une latence comparable à l'implémentation logicielle dans un processeur contemporain[5]. Toutefois, en version pipelinée, leur débit est dix fois meilleur. De plus, ils occupent une fraction d'un FPGA moyen de gamme, et ce débit pourrait donc être encore décuplé par l'utilisation de parallélisme.

Ce qui est intéressant dans ces chiffres, plus que leur valeur absolue, c'est qu'ils montrent qu'à partir d'une certaine complexité, la souplesse de programmation du FPGA lui permet de rattraper son retard de performance intrinsèque sur le processeur [27]. Il faut en effet comparer ces chiffres avec les performances relatives des opérateurs de base (quatre opérations

[5]Pour être honnête, l'Itanium est capable de latences 5 à 10 fois meilleures que le Pentium 4 auquel nous nous comparons.

Exponentielle				
Format	approche HOTBM		approche itérative	
(w_E, w_F)	Surface	Délai	Surface	Délai
(7, 16)	480	69	472	118
(8, 23)	948	85	728	123
(9, 38)	–	–	1242	175
(11, 52)	–	–	2045	229
Logarithme				
Format	approche HOTBM		approche itérative	
(w_E, w_F)	Surface	Délai	Surface	Délai
(7, 16)	627	56	556	70
(8, 23)	1368	69	881	88
(9, 38)	–	–	1893	149
(11, 52)	–	–	3146	182

TAB. 2.4 – Surface (Virtex-II slices) et délai (ns) des fonctions élémentaires flottantes sur un Virtex-II 1000.

	2.4 GHz Intel Xeon			100 MHz Virtex-II FPGA		
Fonction	Cycles	Latence (ns)	Débit (10^6 op/s)	Cycles	Latence (ns)	Débit (10^6 op/s)
Logarithme	196	82	12	11	64	100
Exponentielle	308	128	8	15	85	100

TAB. 2.5 – Comparaison de performances pour exponentielle et logarithme pour la précision simple, entre la `glibc` sur un processeur Intel PIV Xeon (2,4GHz) et notre implémentation FPGA Virtex-II (XC2V1000-4).

et racine carrée) en virgule flottante : dans ce cas, le FPGA est au moins dix fois plus lent que le processeur [127, 57, 7, 90, 49], et il faut déployer un parallélisme massif pour rattraper cet écart [101, 46, 59]. Dans notre application, point besoin de parallélisme, c'est la granularité plus fine de sa programmation qui permet au FPGA de surpasser le processeur.

2.7 Conclusion : le retour des fonctions élémentaires en matériel

Faut-il consacrer du silicium des processeurs à des unités dédiées au calcul de fonctions élémentaires ? Si cette question a fait débat [118], avec la publication au siècle dernier de nombreuses architectures pour l'évaluation de fonctions en matériel [61, 151, 65, 23, 148, 149, 150], le consensus actuel est que ce silicium sera mieux employé à améliorer les unités flottantes de base et augmenter leur nombre. Les fonctions élementaires sont donc essentiellement implémentées en logiciel [139, 102, 112], et ce même pour les processeurs compatibles x86 qui, par héritage, offrent des instructions de calcul des principales fonctions élémentaires

[4]. Il faut noter que la bibliothèque mathématique GNU `glibc` utilisée dans la comparaison de la table 2.5 fait appel pour les deux fonctions étudiées aux instructions machines micro-codées `fyl2x`, `fyl2xp1` et `f2xm1`, et pâtissent donc de cet héritage, comme l'illustre la comparaison avec l'Itanium.

La question se pose différemment dès lors que l'on cible un FPGA : il est reconfigurable, donc un opérateur matériel d'évaluation d'une fonction n'y occupera des ressources que dans une application qui en a effectivement besoin. De plus, il sera possible de tailler cet opérateur au plus juste pour les besoins de l'application. C'est vrai pour la précision de la fonction. C'est aussi vrai pour la fonction elle-même : les méthodes à base de table s'adaptent ainsi à des fonctions composées qui seraient très coûteuses à implémenter par composition de fonctions élémentaires. Nous insistons sur le fait que nous avons tout mis en œuvre pour que cette adaptation soit automatique.

Enfin, nous avons montré que la flexibilité du FPGA lui permet même de surpasser en débit, pour une fonction élémentaire, l'implémentation correspondante dans le processeur, là où les opérateurs de base avaient un débit dix fois plus faible. On obtient de telles performances par l'utilisation d'algorithmes matériels spécifiques, dont nous avons donné des exemples.

Certains supercalculateurs, vendus par exemple par Silicon Graphics et Cray, contiennent des FPGA qui sont actuellement utilisés pour accélérer des calculs exotiques, dont les opérations de base ne sont pas disponibles dans les processeurs (cryptographie, génomique, etc). Nos opérateurs sont un pas de plus vers leur utilisation également comme accélérateurs flottants. La publication d'opérateurs pour les quatre opérations en simple [127, 101], puis en double [46, 59] précision avait permis de montrer que le parallélisme massif pouvait se traduire par une performance supérieure à celle du processeur [144, 72, 77]. Nous offrons des implémentations des fonctions élémentaires [50, 51, 53, 54, 56] compatibles avec les fonctions de la bibliothèque mathématique standard, ce qui permet de transposer rapidement un code C ou Matlab sur FPGA sans crainte d'obtenir un comportement numérique différent. D'une part, nous avons montré que la performance était au rendez-vous. D'autre part, même si ce n'était pas le cas, la bande passante entre le FPGA et le processeur est une ressource précieuse, et on préférera qu'une évaluation de fonction soit implémentée au plus près du reste du calcul, sur le FPGA.

Il nous reste à faire la démonstration d'une telle accélération sur une application réelle. Une première collaboration a commencé avec l'université technique de Cluj Napoca pour l'accélération d'une simulation du champ magnétique produit par un ensemble de bobines, dont les équations mettent en jeu la fonction logarithme.

CHAPITRE 3

L'arrondi correct dans la bibliothèque mathématique

Cependant, les astronomes étaient en séance pour discuter la 72ème décimale du logarithme Népérien de 0,000 000 042 (...)

Christophe, *La famille Fenouillard*

3.1 Enjeux et difficultés

3.1.1 La virgule flottante dans les calculateurs modernes

Jusqu'en 1985, la virgule flottante était très hétérogène d'un modèle d'ordinateur à l'autre. D'une part, cela rendait l'écriture d'un programme portable extrêmement difficile. D'autre part, la preuve de propriétés numériques de programmes, même simples, était également très complexe : à la combinatoire des cas du programme à prouver, il fallait ajouter celle des comportements possibles des systèmes sur lesquels ce programme serait amené à s'exécuter [45]. Enfin, une conséquence indirecte de cette absence d'une spécification commune était souvent l'absence de spécification tout court. Le lecteur intéressé par ces questions peut se référer à la page web de William Kahan à Berkeley.

De ce point de vue, la définition de la norme IEEE-754 [5] a été un grand progrès. Cette norme définit

- des formats de nombres flottants, dont les formats simple précision sur 32 bits (un bit de signe, 8 bits d'exposant et 24 bits de mantisse dont un implicite) et double précision sur 64 bits (11 bits d'exposant et 53 bits de mantisse dont un implicite),
- des modes d'arrondi : le plus utilisé est l'arrondi au plus près, mais il y a aussi trois arrondis *dirigés* (vers le haut, vers le bas et vers 0), qui peuvent servir à l'*arithmétique d'intervalle* [106] qui sera évoquée en 3.7,
- et enfin le comportement des quatre opérations et de la racine carrée, incluant les cas exceptionnels.

Ces opérations doivent renvoyer *l'arrondi correct* du résultat mathématique : la valeur retournée est celle qu'on obtiendrait en faisant le calcul avec une précision infinie, puis en arrondissant ce résultat infiniment précis[1]. Compte tenu du format de représentation fini,

[1] La norme spécifie qu'au cas où ce résultat infiniment précis est exactement au milieu de deux nombres

c'est le meilleur arrondi possible. Il a en particulier l'avantage d'être parfaitement spécifié et déterministe, ce qui était rarement le cas des implémentations antérieures à cette norme, même de très bonne qualité. Par exemple, garantir une «erreur inférieure au poids du dernier chiffre de la mantisse» (ce que nous avons fait dans le chapitre précédent) est numériquement presque aussi bon que l'arrondi correct, mais spécifie plusieurs résultats admissibles.

D'abord implantée dans le coprocesseur mathématique 80387 d'Intel, acceptée comme une norme ISO/IEC, la norme IEEE-754 a été rapidement adoptée par tous les calculateurs. Il faut noter que cette norme définit l'arithmétique d'une manière qui peut être transposée en logiciel, en matériel, ou (le plus souvent) en une combinaison des deux. Par parenthèse, la tendance n'est pas forcément à faire de plus en plus d'opérations en matériel. Ainsi, l'architecture IA64 définie récemment par HP et Intel et implémentée dans la famille de processeurs Itanium n'a pas de diviseur flottant. Cela se justifie essentiellement par la rareté relative de l'opération de division dans les applications [115], et aussi par l'existence des algorithmes quadratiques très performants évoqués en 2.2.2, qui de plus peuvent être optimisés pour différents contextes. HP et Intel ont ainsi rivalisé d'ingéniosité pour définir chacun une gamme de routines de division flottante [102, 22].

3.1.2 Des fonctions élémentaires imparfaitement spécifiées

La norme IEEE-754, telle que définie en 1985, ne mentionne pas les fonctions élémentaires telles que sinus, cosinus, exponentielle, logarithme, etc. Pour celles-ci, les normes des langages comme C99 [82], Fortran2003 [83] donnent typiquement des listes de fonctions élémentaires, avec leurs formats d'entrée et de sortie, mais restent très lâches quant à la qualité de l'implémentation. Le comportement dans certains cas limites reste sujet à discussion [43]. Surtout, l'absence d'une spécification stricte a longtemps laissé la porte ouverte à des implémentations franchement imprécises. Cette situation s'améliore toutefois au fil des ans, et les fabricants publient désormais des chiffres de précision [132, 100], typiquement une erreur relative inférieure à $0,501$ ulp, alors que l'arrondi correct serait une erreur strictement inférieure[2] à $0,5$ ulp. Une formulation équivalente, pour des raisons vues plus bas, est de parler de 99% d'arrondi correct. Toutefois il s'agit encore de mesures non exhaustives, et non de garanties.

En pratique, en 2007, même avec ces implémentations de qualité, la portabilité d'un programme flottant d'un ordinateur à l'autre n'est pas assurée, ce qui oblige à des acrobaties lorsque cette portabilité est nécessaire, par exemple lors de la distribution d'un calcul cahotique sur une réseau de stations de travail hétérogène [141, 36].

3.1.3 Le dilemme du fabricant de tables

La raison principale de cette absence d'une spécification stricte est qu'on ne savait pas, en 1985, assurer l'arrondi correct des fonctions élémentaires à coût raisonnable. Ce problème est appelé le «dilemme du fabricant de table», car les premiers fabricants de tables de logarithmes y ont été confrontés. Voici de quoi il s'agit.

flottants, l'arrondi au plus près est défini comme celui des deux dont la mantisse se termine par un zéro (arrondi au plus près pair). Ce choix évite le biais statistique qu'on aurait si on arrondissait ces cas tous dans la même direction.

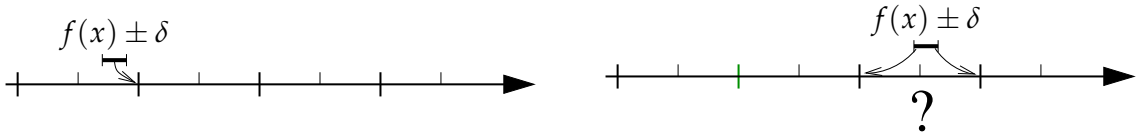
[2] Le *strictement* est une caractéristique des fonctions transcendantes, comme cela sera vu en 3.1.5.

On évalue une fonction élémentaire comme le logarithme par un schéma d'approximation (par exemple un développment de Taylor). On choisit un schéma dont on est capable de borner la précision *a priori*. Le calcul d'une telle borne sera l'objet du chapitre 4. Mettons nous par exemple à la place d'un fabricant de table de logarithme qui désire construire une table $T_{\log}(x)$ précise à 5 chiffres décimaux. Il va utiliser un schéma d'approximation légèrement plus précis, par exemple à 7 chiffres. L'expression «précis à 7 chiffres» est un raccourci de langage [3] pour dire que son schéma d'approximation commet une erreur relative inférieure à 10^{-7}, ou en termes mathématiques, que $\left|\frac{T_{\log}(x)-\log(x)}{\log(x)}\right| < 10^{-7}$.

Le plus souvent, ce schéma suffira pour déterminer l'arrondi correct du logarithme. Par exemple, pour $x = 17,42$, le fabricant de table évaluera $\log(x) = 2,857619 \pm 10^{-7}$, ce qui permet de garantir que l'arrondi correct à 5 chiffres de cette valeur est $T_{\log}(x) = 2,8576$.

Mais considérons le nombre $x = 3.6297$. Notre fabricant de table calculera que $\log(x) = 1.289150 \pm 10^{-7}$, et sera incapable d'en déduire l'arrondi à 5 chiffres. Cela pourrait être aussi bien 1.2891 que 1.2892.

Le lecteur se convaincra aisément que ce problème se transpose parfaitement à la virgule flottante binaire. La figure 3.1 illustre la situation pour l'arrondi au plus près.



(a) Cas facile à arrondir (b) Cas difficile à arrondir

FIG. 3.1 – Le dilemme du fabricant de table. La droite réelle est représentée avec les flottants en gras, et les milieux de deux flottants en traits fins. Toute évaluation de fonction élémentaire est associée à un intervalle d'erreur, et l'arrondi correct au plus près ne peut être décidé si cet intervalle contient le milieu de deux nombres flottants. Pour les arrondis dirigés, le problème se pose de la même manière lorsque l'intervalle d'erreur contient un nombre flottant.

Pour le fabricant de table, la situation où l'on ne sait pas arrondir ne posait pas vraiment problème : dans ce cas, le logarithme est très proche du milieu entre deux nombres consécutifs à 5 chiffres, ce qui signifie que les deux candidats, l'arrondi supérieur et l'arrondi inférieur, sont presque aussi bons, ou plutôt presque aussi mauvais l'un que l'autre. N'importe lequel des deux fera l'affaire, et son erreur sera proche de la limite théorique imposée par la tabulation de 5 chiffres seulement[4]. De même, il est difficile de justifier l'exigence de l'arrondi correct par la simple amélioration de la précision : par rapport aux meilleures bibliothèques courantes qui calculent «avec une erreur relative inférieure à 0,501 ulp», le gain en précision sera également en gros d'un centième d'ulp une fois sur cent. Redisons-le, les justifications de l'arrondi correct sont d'une part la portabilité numérique, d'autre part la

[3] La valeur 0.999999999 est une approximation «précise à 7 chiffres» de la valeur 1, bien qu'aucun chiffre ne concorde.

[4] D'après Muller [112], les fabricants de table arrondissaient alors au hasard, mais enregistraient dans un cahier spécial le choix fait. Cela permettait ainsi de détecter les plagiats : une table calculée honnêtement par un concurrent devait faire des choix différents. Il s'agit probablement là du premier algorithme de signature numérique.

cohérence avec les quatre opérations, et plus généralement de fermer la porte au laxisme qu'une absence de spécification laisse toujours possible.

3.1.4 L'arrondi correct par raffinement de précision

Un algorithme simple pour déterminer l'arrondi correct est le suivant : on évalue d'abord une approximation y_1 de la fonction $f(x)$ avec une borne sur l'erreur relative totale commise $\overline{\epsilon}_1$. Autrement dit, on sait garantir que la valeur exacte de $f(x)$ se trouve dans l'intervalle $[y_1/(1+\overline{\epsilon}_1), y_1/(1-\overline{\epsilon}_1)]$ (on suppose ici y_1 positif). On réalise ensuite un *test d'arrondi* : il s'agit de tester si cet intervalle contient un nombre flottant dans le cas de l'arrondi dirigé, ou le milieu de deux nombres flottants dans le cas de l'arrondi au plus près. Si ce n'est pas le cas, il suffit d'arrondir y_1 à la précision cible pour obtenir l'arrondi correct de $f(x)$. Dans le cas contraire, on recommence avec une évaluation plus précise ($\overline{\epsilon}_2 \ll \overline{\epsilon}_1$). Cet algorithme est dû à A. Ziv [155], qui a surtout remarqué que les étapes plus précises sont exécutées très rarement : pour revenir à notre fabricant de table et à son schéma d'approximation précis à 7 chiffres pour une précision cible de 5 chiffres, le dilemme n'arrive que lorsque les deux derniers chiffres sont exactement 50, soit environ une fois sur 100 – en supposant une répartition aléatoire uniforme des deux derniers chiffres, ce qui n'est vrai qu'approximativement. Si notre fabricant de table avait utilisé un schéma précis à 8 chiffres, il serait confronté au dilemme une fois sur 1000. Transposé à la virgule flottante binaire en double précision, un schéma d'approximation précis à $53+p$ bits ($\overline{\epsilon}_1 = 2^{-53-p}$) verra le test d'arrondi échouer avec une probabilité de l'ordre de 2^{-p}. Même si les étapes d'évaluation plus précises sont beaucoup plus coûteuses, leur contribution au temps d'exécution moyen pourra rester très faible.

En résumé, l'arrondi correct, pour une fonction élémentaire, se ramène à

- savoir programmer des schémas d'approximation pour les fonctions élémentaires, à différentes précisions. Sur ce point, la littérature était déjà très riche. Notre contribution concerne essentiellement les techniques de calcul entre 100 et 1000 bits de précision [41, 28] ;
- savoir borner l'erreur maximale totale commise par un schéma d'approximation. C'est une problématique bien distincte de la précédente, ce qui justifie que le chapitre 4 de ce mémoire lui soit consacré ;
- savoir réaliser rapidement le test d'arrondi, qui a lieu à l'exécution, pour chaque évaluation de la fonction – le calcul de ces bornes d'erreur est lui réalisé *a priori*, une fois pour toutes.

3.1.5 Quelle précision pour l'arrondi correct ?

Il reste une question de fond : l'algorithme de Ziv termine-t-il ? La réponse, dans le cas général, est non : par exemple, si l'image d'un nombre flottant est *exactement* le milieu de deux nombres flottants, on a une itération infinie pour l'arrondi au plus près. Heureusement, il existe une collection de théorèmes classiques [112] qui démontrent que l'image d'un rationnel par une fonction transcendante n'est pas un rationnel à quelques exceptions triviales près, comme $\log(1) = 0$. Un nombre flottant étant un rationnel, son image par la fonction est irrationnelle, donc l'écriture binaire de cette image ne présente pas de suite infinie de 1 ou de 0, ce qui garantit que l'itération termine. Toutefois, pour des fonctions comme $\log_{10}$ et

$\log_2$ (logarithmes en base 10 et 2), le nombre de cas exacts est plus élevé (23 et 2047 respectivement pour la double-précision IEEE-754) et il convient de filtrer avant le test d'arrondi. Pour la fonction `power` qui calcule x^y, le nombre de cas exacts est encore plus grand et ce préfiltrage est complexe [89].

Une fois établi que l'itération termine, quand termine-t-elle ? Cette question est importante car elle conditionne le temps d'exécution au pire cas. Les bornes théoriques existantes sont tellement élevées qu'elles sont inutiles en pratique [112]. Toutefois, un argument statistique dû à Gal [68] donne une estimation de la précision au pire cas. Revenons pour le présenter à notre fabricant de table. Si ce dernier construit une table à 10^6 entrées, et sous l'hypothèse précédente de répartition uniforme des chiffres, parmi les 10^6 valeurs du logarithme correspondantes, il en existera probablement une dont l'écriture décimale infinie est xxxxx500000xxx.... Par conséquent, on calculera avec $5 + 6 + p$ chiffres de précision cible : 5 chiffres de précision cible, 6 de plus parce qu'on a 10^6 entrées à la table, et encore p chiffres supplémentaires. On s'attend alors à ce que la probabilité qu'il existe une entrée dont on ne puisse décider l'arrondi soit de l'ordre de 10^{-p}, et on peut la faire baisser arbitrairement en choisissant p. On ne cherchera pas à formaliser en termes statistiques précis ce raisonnement puisque l'une de ses hypothèses (la répartition uniforme des chiffres) est invalide. Si on le transpose à la virgule flottante binaire double-précision de la norme IEEE-754, on s'attend à ce que la précision nécessaire pour déterminer l'arrondi correct, au pire cas parmi les 2^{64} flottants en entrée (un nombre flottant s'écrit sur 64 bits), soit de l'ordre de $2^{-53-64} = 2^{-117}$.

Ce fut le sujet de la thèse de Vincent Lefèvre [96] de chercher à calculer exactement la précision au pire cas pour les fonctions élémentaires les plus courantes. Selon la discussion précédente, il suffit en principe d'évaluer la fonction sur toutes ses entrées avec une précision de l'ordre de 200 bits, ce qu'on ferait d'ailleurs en utilisant le raffinement de précision de Ziv. Malheureusement, il y a trop d'entrées possibles (de l'ordre de 2^{64} flottants différents), ce qui rend cet algorithme inutilisable en pratique : un ordinateur à 10GHz calculant une fonction en 10 cycles – on en est loin – testerait de l'ordre de 2^{30} entrées par seconde, et mettrait donc environ cinq siècles pour le test d'une fonction. Certaines fonctions sont plus faciles que d'autres. La plus facile est 2^x qui n'a besoin d'être testé que sur une binade, soit 2^{53} entrées : cela reste intractable en pratique.

Lefèvre a donc défini des algorithmes rapides selon les principes suivants :

- Détermination d'intervalles sur lesquels la fonction peut être approchée localement par une droite, dont les coefficients sont calculés en grande précision ;
- Élimination rapide de la majorité des intervalles par des considérations de théorie des nombres sur les coefficients, pour prouver que le segment ne s'approche pas suffisamment près d'un point de la grille flottante ;
- Évaluation rapide d'un polynôme approchant la fonction en tous les points des intervalles restant, en utilisant le fait que le calcul de valeurs successives d'un polynôme en des points équidistants peut se faire incrémentalement uniquement par des additions [84].

Il faut noter que ces algorithmes calculent les pires cas pour une fonction et sa réciproque, puisque le problème est ramené à minorer la distance entre un segment de droite et les points d'une grille à deux dimensions.

Ces algorithmes ont ensuite été améliorés par Lefèvre, Stehlé et Zimmermann pour, notamment, utiliser des approximations d'ordre supérieur de la fonction, et ramener le problème à une réduction de réseaux [130, 129]. Ils permettent désormais de tester une fonction complètement en quelques mois de calculs seulement.

Début 2007, les pires cas sont ainsi connus pour exp, log, 2^x, $\log_2$, 10^x, $\log_{10}$, cosh, sinh arcsin, arccos et arctan sur l'ensemble de leur domaine, et sin, cos, tan et x^y sur des domaines partiels. Les résultats sont conformes aux prédictions statistiques de 2^{-117} (entre 2^{-115} et 2^{-120}), sauf dans certains cas particuliers sur lesquels nous reviendrons en 3.4.

3.2 CRLibm, vers une bibliothèque mathématique parfaite

3.2.1 Deux étapes de Ziv

Connaissant la précision au pire cas requise pour l'arrondi correct, il a été possible de simplifier l'algorithme de Ziv pour qu'il ne fasse plus que deux itérations, dont la seconde est plus précise que le pire cas, ce qui garantit qu'elle retourne toujours l'arrondi correct. L'avantage principal est qu'on n'a plus besoin de multiprécision arbitraire, laquelle est algorithmiquement plus coûteuse qu'une précision étendue fixée de l'ordre de 2^{-120}. En pratique, on gagnera un facteur 100 sur la performance. De plus, il sera plus facile de prouver du code en deux étapes que du code à un nombre arbitraire d'étapes reposant sur des algorithmes de multiprécision arbitraire.

Voici posées les bases de la bibliothèque CRLibm. Pour chaque fonction, il s'agit

- de programmer un schéma d'approximation précis à une soixantaine de bits (cette première étape sera dans la suite souvent dénommée *étape rapide*),
- de borner formellement l'erreur maximale commise,
- d'utiliser cette borne dans un test d'arrondi
- de programmer un schéma d'approximation plus précis que le pire cas nécessaire (cette seconde étape sera dénommée *étape précise*),
- et de prouver formellement que cette seconde étape est plus précise que le pire cas.

Par rapport à des bibliothèques précédentes, en particulier celle de Ziv, CRLibm propose également des fonctions pour les trois arrondis dirigés, et quelques fonctions d'intervalle [35] sur lesquelles nous reviendrons en 3.7.

3.2.2 Compromis précision/performance

Si l'on note T_1 et T_2 les temps d'exécution respectifs des deux étapes, mesurés typiquement en cycles du processeur, T_t le coût du test d'arrondi, et p_2 la probabilité d'avoir besoin de l'étape précise, la performance moyenne sera de l'ordre de

$$T_{moyen} = T_1 + T_t + p_2 T_2. \tag{3.1}$$

Dans cette formule,

- T_1 est comparable au temps d'exécution d'une bibliothèque standard, sans arrondi correct mais «précise à 0.501 ulp».
- T_t est de l'ordre de quelques cycles seulement. Pour le minimiser, nous avons étudié différents tests d'arrondi — certains purement en flottant, certains passant par des manipulations de bits de la mantisse [31, 1]. Pour certaines fonctions et certains processeurs on arrive à calculer le test d'arrondi en parallèle du calcul de l'arrondi lui-même, si bien que T_t est nul.

- T_2 est dans un facteur 2 à 10 de T_1 (à nouveau, cela dépend des fonctions et du processeur utilisé), si bien qu'il suffit que p_2 soit de l'ordre de 10^{-2} à 10^{-3} pour que la contribution de $p_2 T_2$ au temps moyen soit négligeable.
- On fait varier p_2 en jouant sur la précision de l'étape rapide : comme vu précédemment, si celle-ci est précise à 2^{-53-p}, alors p_2 sera de l'ordre de 2^{-p}. Bien sûr, une première étape plus précise sera sans doute plus lente (T_1 est plus élevé), et il faut donc étudier le compromis représenté par l'équation (3.1) au cas par cas des fonctions.

3.2.3 Compromis portabilité/optimisation

Il reste un aspect qui n'a pas été évoqué, c'est celui de la portabilité de CRLibm. L'objectif premier est une bibliothèque portable sur tout système supportant la double-précision IEEE-754. Toutefois, les bibliothèques constructeurs utilisent souvent des optimisations non portables, que ce soit la précision double étendue (format de 80 bits dont 64 bits de mantisse), le FMA, ou les extensions de type SSE2 qui permettent d'exécuter plusieurs opérations flottantes en parallèle [100, 22, 143, 4]. Nous avons cherché à intégrer ces optimisations de deux manières.

Pour tout ce qui relève de l'exploitation du parallélisme d'instruction offert par les processeurs récents, nous choisissons de nous reposer sur le compilateur (avec certes un succès mitigé jusque là pour le parallélisme de type SSE2, mais la communauté de la compilation est très active sur ce sujet). Notre rôle en tant que programmeur se limite à écrire du code parallélisable. Par exemple, nous préférerons un schéma de Estrin à un schéma de Horner [31, 112] pour l'évaluation d'un polynôme, ou nous choisirons un test d'arrondi qui puisse être recouvert par du calcul. Ainsi, l'évaluation de la phase précise de l'exponentielle se termine par une multiplication par une puissance de 2. Le test d'arrondi peut être réalisé sur la valeur avant cette multiplication, et donc s'exécuter en parallèle avec elle (ce n'est pas vrai dans le cas où le résultat est dénormalisé, qui nécessite un traitement spécifique).

En ce qui concerne les possibilités non portables des processeurs (précision double étendue et/ou FMA), nous cherchons à les intégrer de manière transparente lorsque c'est possible. Dans le code des fonctions, nous utilisons des macros génériques dont la définition dépend du processeur et est choisie à la compilation. Le but est de n'avoir qu'un seul code pour la fonction et que les aspects non-portables soient proprement encapsulés, afin notamment que les preuves de propriétés sur la fonction s'appuient sur des lemmes prouvés pour chaque implémentation des macros. Cette approche fonctionne assez bien pour l'utilisation du FMA : lorsqu'il est disponible, on remplace par exemple l'algorithme de multiplication exacte de Dekker [45] (17 opérations) par un algorithme plus simple utilisant juste deux FMA. Cette modification d'une macro apporte un bénéfice de performance considérable à toute l'arithmétique double-double et triple-double qui l'utilise (ces arithmétiques seront présentées en 3.3.3), sans nécessiter le moindre changement ni au code des fonctions, ni à leur preuve.

Par contre, l'utilisation de la précision double étendue nécessite de réécrire le plus gros du code, et n'a été étudiée que de manière ponctuelle et expérimentale [31].

Au bout de trois années de développement de CRLibm, nous arrivons à des fonctions dont le temps d'exécution moyen est comparable à celui des fonctions de la bibliothèque standard sans arrondi correct. Le temps d'exécution au pire cas est souvent inférieur à celui de la bibliothèque standard, donc en tout état de cause acceptable même pour des applications temps réel ou critiques.

3.3 Techniques de précision étendue pour l'étape précise

L'un des intérêts principaux d'un schéma en deux étapes est de permettre l'optimisation au plus juste de la seconde étape précise. Cela passe en particulier par la définition d'une arithmétique adaptée à la précision requise pour cette seconde étape, soit environ 130 bits. Nous avons utilisé différents formats, suivant les briques de base utilisées.

3.3.1 La retenue conservée logicielle

Présentons d'abord SCSlib, bibliothèque multiprécision écrite essentiellement par David Defour dans sa thèse sous ma direction.

Cette bibliothèque, utilisant uniquement de l'arithmétique entière, utilise le principe de la retenue conservée logicielle, ou SCS pour *software carry save*. Sa précision est paramétrable, de quelques dizaines à quelques milliers de bits. L'idée est de représenter une mantisse en précision étendue comme un tableau d'entiers, mais en laissant quelques bits à zéros dans ces entiers pour absorber les retenues intermédiaires qui se produiront dans les calculs. Par exemple, les paramètres utilisés dans SCSlib sont de représenter un nombre comme un tableau de 8 chiffres compris entre 0 et $2^{30} - 1$, chaque chiffre étant stocké dans un entier 32 bits [41].

On réalise l'addition comme à la main, par addition chiffre à chiffre et propagation de retenue de droite à gauche. Un premier intérêt d'avoir mis deux bits de côté est que la détermination de la retenue se fait par un calcul de masques en C portable. En effet, bien que tous les processeurs 32 bits possèdent des instructions permettant de détecter et de propager des retenues sur des additions 32 bits, il n'y a pas de manière portable d'utiliser ces instructions. C'est la raison que met en avant Brent, qui est à notre connaissance le premier à avoir décrit ce type d'arithmétique [13]. La bibliothèque de Ziv utilise une bibliothèque multiprécision arbitraire qui utilise les mêmes principes, mais en utilisant comme chiffres des nombres flottants, ce qui rend les opérations plus coûteuses. La bibliothèque GMP [70] préfère recourir à de l'assembleur pour cette propagation de retenue. C'est plus performant mais cet assembleur doit être écrit pour chaque processeur connu[5].

Un second intérêt, si l'on a mis plus d'un bit de côté, est de pouvoir additionner plusieurs nombres pour une seule propagation de retenue. C'est surtout utile pour l'algorithme de multiplication, dont une représentation abstraite est donnée par la figure 3.2.

Dans SCSlib, les paramètres choisis sont tels que toutes les additions de tous les produits partiels représentés sur cette figure pourront se faire sans propagation de retenue. Avec des mots de 30 bits, le produits partiels (notés $y_j x_i$ sur la figure) occupent 60 bits, et on peut en additionner 16 en arithmétique 64 bits sans dépassement de capacité. Il ne reste ensuite qu'à redécouper les nombres 64 bits obtenus (les c_i sur la figure) pour en refaire des chiffres de 30 bits, par une unique propagation de retenue.

Depuis la norme C99 [82], il est possible d'exprimer de manière portable ce type de mélanges d'arithmétiques entières 32 et 64 bits, et les compilateurs font du bon travail pour utiliser au mieux les ressources des processeurs sous-jacents.

Un intérêt supplémentaire que nous avons mis en avant [28] est que l'arithmétique SCS permet d'exposer du parallélisme présent dans les processeurs modernes. Le nom que nous

[5] Ceci fait du code source de GMP un monument unique à l'écodiversité des langages assembleurs.

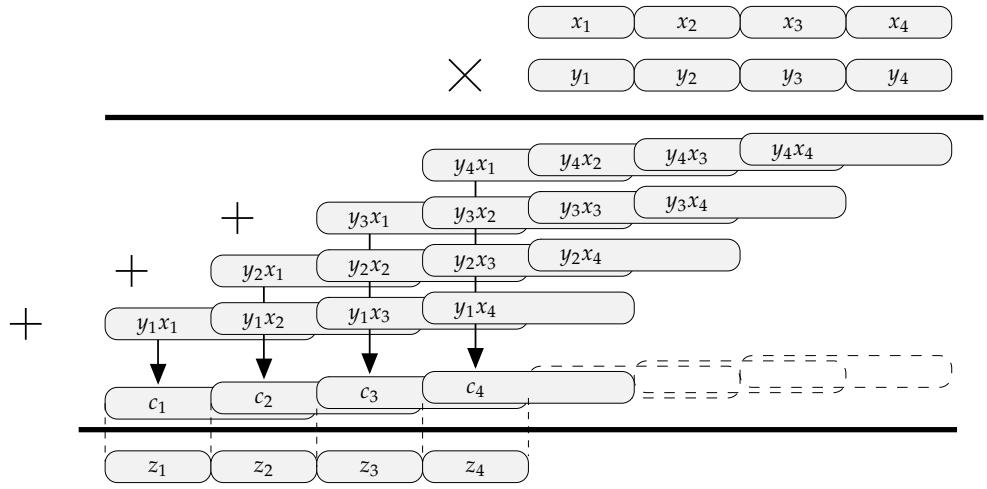


FIG. 3.2 – La multiplication en grande base avec retenue conservée

avons donné à cette bibliothèque fait référence à la retenue conservée, technique couramment utilisée en matériel dans le but d'exposer du parallélisme de chiffre.

Le format de SCSlib est un format flottant autour d'une mantisse SCS de 8 chiffres de 30 bits, soit une précision pratique de 210 bits, car le chiffre de poids fort peut n'avoir qu'un seul bit significatif. Cette précision est très supérieure à ce qu'exige l'arrondi correct (158 bits d'après Lefèvre et Muller) : l'idée était de se donner de la marge pour que nos premières preuves soient simples. Ce format a aussi la propriété que l'ensemble des nombres représentés est un sur-ensemble strict de l'ensemble des flottants IEEE-754. Nous avons implémenté essentiellement les opérations de base : addition, soustraction, multiplication, et les conversions.

La performance est au rendez-vous, puisque notre bibliothèque SCS surpasse GMP, la référence en la matière, sur toutes les opérations de base [28]. À dire vrai, c'est surtout parce que notre bibliothèque a sa précision fixée statiquement, ce qui permet des optimisations qui ne sont pas possibles pour GMP, qui gère une multiprécision arbitraire. Toutefois, l'exploitation du parallélisme a aussi un impact non négligeable, en particulier sur Itanium [28]. De plus, GMP évolue pour exprimer lui aussi du parallélisme.

Appliqué aux fonctions élémentaires, nous avons été capables d'améliorer la performance mesurée au pire cas sur la bibliothèque de Ziv d'un facteur 10. Naturellement, nous apportions en plus la garantie que cette performance est vraiment celle du pire cas, ce qui est pour certaines applications bien plus important.

3.3.2 Critique de SCSlib pour CRLibm

Constatons d'abord que SCS est un format peu économe tant en mémoire qu'en temps d'exécution. Par exemple, lors de l'évaluation d'un polynôme de degré 20 (typique d'une étape précise) sur un argument $x \ll 1$ par le schéma de Horner, il est bien connu qu'il n'est pas nécessaire de représenter tous les coefficients sur 120 bits pour obtenir une précision finale de 2^{-120} [11]. Les coefficients de plus grands degrés, qui seront multipliés par un x^i très petit, peuvent se permettre d'être bien moins précis. Cet effet est facile à quantifier. Or dans SCSLib, la précision (8 chiffres de 30 bits) est fixée à la compilation, et est la même pour

tous les nombres et tous les opérateurs. Remarquons au passage que la multiplication de nombres de n chiffres coûte de l'ordre de n^2 opérations. Remarquons aussi que l'entrée de notre évaluation polynomiale, même pour l'étape précise, est souvent un flottant de 53 bits seulement. L'approche SCS va donc l'injecter dans un nombre SCS de 240 bits, puis le multiplier par un coefficient également sur 240 bits alors qu'une multiplication d'une soixantaine de bits aurait le plus souvent suffit.

Certes, on aurait pu définir une arithmétique SCS offrant plusieurs précisions, toutes sélectionnées statiquement. Ainsi on aurait toujours l'avantage de performance sur GMP que donne du code statique, et moins de gâchis. Toutefois, cette approche aurait encore un inconvénient : les deux étapes d'évaluation, l'étape rapide et l'étape précise, utilisent des arithmétiques complètement différentes, flottante double précision pour la première, SCS basé sur des entiers pour la seconde. Ceci interdit de réutiliser des résultats intermédiaires de l'étape rapide vers l'étape précise. Cela interdit également de partager des tables entre les deux étapes, sauf à payer de coûteuses conversions.

Quitte à utiliser une arithmétique adaptable en précision, donc complexe d'utilisation, pour notre étape précise, autant qu'elle puisse étendre harmonieusement celle de l'étape rapide. Voilà pourquoi nous utilisons à présent dans CRLibm une approche de la précision multiple uniquement basée sur des flottants.

3.3.3 Arithmétiques double-double, triple-double et double-double-étendue

Ces arithmétiques représentent un nombre en grande précision par une somme non évaluée de nombres flottants, de même qu'on peut représenter le nombre $123,456$ par la somme de flottants à deux chiffres décimaux suivants : $123,456 = 1,2 \cdot 10^2 + 3,4 + 5,6 \cdot 10^{-2}$.

L'arithmétique double-double, qui représente donc un nombre comme une somme non évaluée de deux flottants, ici double-précision, est attribuée à Dekker [45]. Les algorithmes pour ses opérations de base sont bien connus [84] et bien prouvés [9]. Elle est utilisée au moins depuis Gal [69] pour des fonctions élémentaires précises. Dans CRLibm, elle est utilisée pour représenter (et donc pour calculer) le résultat de l'étape rapide, qui par définition doit être plus précis qu'un double. Avec une précision de 106 bits, elle est en fait bien trop précise pour cette étape rapide, mais pas assez pour l'étape précise qui a besoin d'environ 120 bits (voir ci-dessous en 3.4). Nos premiers développements portables (logarithme, exponentielle, fonctions trigonométriques et hyperboliques) utilisaient un format double-double pour l'étape rapide, et un format SCS pour l'étape précise. Nous avons vu les inconvénients.

Lorsque le processeur dispose de précision double-étendue, on peut utiliser pour l'étape précise une arithmétique double-double-étendue ou DDE, qui fournit au moins 128 bits de précision. Dans ce cas le code sera très efficace, d'autant que l'étape rapide peut être calculée en double-étendue, donc uniquement avec l'arithmétique native du processeur [31]. Mais il ne sera pas portable [6].

[6] Depuis que la firme Apple équipe ses Macintosh de processeurs Intel, on peut affirmer que l'écrasante majorité des calculateurs vendus possède une unité flottante en double-étendue. Mais la portabilité sur le plus petit dénominateur commun, la double-précision, ne doit pas être négligée pour autant. Par exemple, parmi les 5 premiers du top500 des supercalculateurs de novembre 2006, on trouve 4 machines IBM à base de processeurs Power, donc sans précision double-étendue en matériel. De plus, les unités SSE2 des processeurs IA32 récents ne fonctionnent qu'en précisions simple et double. Comme ces unités sont bien plus performantes, et de surcroît plus simples à utiliser, que l'unité «historique» qui offre la double-étendue, elles sont devenues la cible préférée d'un nombre croissant de compilateurs, dont les versions récentes de `gcc`.

Pour permettre une meilleure intégration des codes de l'étape rapide et de l'étape précise, nous avons entrepris de développer une arithmétique triple-double ad-hoc. Ce fut le sujet de la thèse de *Diplom* (l'équivalent allemand du Master) de Christoph Lauter [88] sous ma direction. Il s'agit de représenter un nombre comme la somme non évaluée de trois flottants double-précision. Les principes sont inspirés des *expansions flottantes* étudiées entre autre par Bailey [78], Shewchuck [126], Daumas et ses étudiants [25, 26, 10], Rump *et al.* [116].

Pour l'objectif de l'arrondi correct, il y a toutefois un saut qualitatif de difficulté par rapport à l'arithmétique double-double, essentiellement pour la raison suivante : une addition flottante IEEE-754 donne par définition l'arrondi correct de la somme de deux doubles, donc l'arrondi correct d'un double-double. Par contre, obtenir l'arrondi correct de la somme de trois doubles, ou même un arrondi avec une borne d'erreur bien prouvée, s'avère beaucoup plus complexe. S'il n'y avait qu'un nouvel opérateur à demander dans les processeurs pour faciliter l'implémentation de fonctions avec arrondi correct, ce serait celui-là : une addition de trois doubles avec arrondi correct. Il est intéressant de constater qu'indépendamment, D. Defour a montré qu'un tel opérateur serait justifié par des raisons de performances [40]. Dans son optique, il ne faudrait toutefois pas l'arrondi correct, mais l'arrondi correspondant à deux additions IEEE-754 successives pour préserver la sémantique séquentielle exigée notamment par le langage C. Il reste à proposer une architecture efficace pour un opérateur d'addition de trois doubles capable de ces deux comportements, arrondi correct ou double arrondi séquentiel. Du point de vue de l'assembleur des processeurs, cet opérateur aura besoin de trois arguments, ce qui est une grosse difficulté pour les jeux d'instruction anciens et compacts comme IA32. Toutefois, le FMA a aussi besoin de trois arguments : ces opérateurs sont donc plus facilement envisageables dans les processeurs disposant déjà d'un FMA.

Cette parenthèse refermée, la spécificité de notre implémentation triple-double est de permettre que les mantisses des trois doubles qui représentent un nombre se «chevauchent». Un exemple de chevauchement, avec des flottants à deux chiffres décimaux, est $123,45 = 1,2 \cdot 10^2 + 3,2 + 2,5 \cdot 10^{-1}$. Dans cet exemple, les chiffres 2 des deux flottants $2,5 \cdot 10^{-1}$ et $3,2$ ont le même poids : on dit qu'il se chevauchent.

Le chevauchement pose plusieurs problèmes. D'une part la représentation n'est plus unique, par exemple on a aussi $123,45 = 1,2 \cdot 10^2 + 3,4 + 5,0 \cdot 10^{-2}$. C'est un problème pour faire des comparaisons, mais pas vraiment pour nos applications, qui vont utiliser des triples-doubles essentiellement pour des évaluations polynomiales. D'autre part, pour un même nombre de flottants, la précision du format complet diminue avec le chevauchement : dans notre exemple à trois flottants, on n'a plus que 5 chiffres décimaux au lieu de 6.

Pour ces raison, les auteurs qui ont étudié les expansions flottantes ont particulièrement étudié des *renormalisations* qui suppriment le chevauchement. Ces renormalisations sont nécessaires, car du chevauchement va apparaître dans les implémentations efficaces des opérations arithmétiques.

Toutefois, une telle renormalisation est coûteuse. Par ailleurs, le format triple-double sans chevauchement présente, avec 159 bits, un excès de précision pour les besoins de l'arrondi correct. L'approche originale de Lauter est donc de vivre avec le chevauchement introduit par les opérations intermédiaires, plutôt que de le supprimer après chaque opération. On n'introduit une renormalisation dans le code que lorsque le chevauchement interdit d'obtenir la précision qu'on veut.

Lauter a réalisé une bibliothèque qui comporte des opérateurs pour les quatre opérations pour toutes les combinaisons d'arguments (double, double-double et triple-double) qui se sont avérées utiles lors du développement de fonctions dans CRLibm. Il a prouvé des théo-

rèmes sur les erreurs relatives commises dont un exemple (choisi au hasard dans [88]) est donné figure 3.3. Ces théorèmes donnent des bornes d'erreur en fonction, entre autres, des chevauchements en entrée et en sortie.

Le rapport technique [88] a été repris, corrigé et étendu, et est maintenu à jour dans la documentation de CRLibm [1]. Nous ne désespérons pas de tenter de le publier un jour. Le problème est la taille des preuves, nous le discuterons au chapitre 4. Par exemple, la preuve de l'arrondi final correct d'un triple double vers un double fait une dizaine de pages – alors que, on l'a vu, la preuve de l'arrondi correct d'un double-double vers un double est une conséquence directe de la norme IEEE-754.

3.3.4 Génération de code triple-double pour CRLibm

Muni de cette collection de théorèmes, Lauter a pu mécaniser la construction de code évaluant des polynômes d'approximation de fonctions par le schéma de Horner. Le code obtenu utilise la plus petite précision possible à chaque étape, parmi double, double-double ou triple-double. L'outil gère automatiquement les chevauchements et insère des renormalisations quand nécessaire. Il produit de surcroît un script de preuve qui permet de prouver une borne d'erreur de l'évaluation totale commise, en utilisant les outils qui seront décrits au chapitre 4.

On a oublié ici les schémas de Estrin, car les briques de base de l'arithmétique double-double et triple-double exposent suffisamment de parallélisme pour bien remplir les pipelines (voir par exemple la figure 3.3) – et consommer déjà plus de registres que les jeux d'instructions historiques comme IA32 ne peuvent offrir.

Au final, la performance est à nouveau meilleure d'un facteur 10 que SCS. Pris par l'autre bout, toutes les fonctions que Lauter a écrites en utilisant la triple-double ont un temps d'exécution au pire cas dans un facteur 10 du temps moyen, et souvent dans un facteur 5.

3.4 Les pires cas spéciaux de Lauter

Une contribution à la théorie a été l'étude d'une anomalie dans les précisions au pire cas par rapport aux prédictions statistiques de Gal. Rappelons que, sous l'hypothèse que les bits du résultat obéissent à une répartition uniforme, au moins à partir du $53^{\text{ème}}$, on s'attend à ce que la précision requise au pire cas soit de l'ordre de 2^{-117}. Or les calculs de Vincent Lefèvre montraient par exemple pour l'exponentielle un pire cas à 2^{-158}. Ce fut mis, sans plus de réflexion, sur le compte de l'invalidité de l'hypothèse d'uniformité, qui encore une fois n'est justifiée par aucune démonstration mathématique rigoureuse.

3.4.1 Une découverte par hasard

En 2003, Christoph Quirin Lauter, alors en maîtrise, partit faire un stage au laboratoire d'Intel à Nijni-Novgorod en Russie. Il s'agit d'un des deux laboratoires de ce constructeur qui s'occupe d'implémenter des fonctions élémentaires optimisées pour les processeurs maison. L'objet de son stage, co-encadré par Andrey Naraikin et moi-même, était de réaliser une fonction élémentaire avec arrondi correct pour Itanium, optimisée avec tout le savoir-faire de cette équipe. En particulier, il s'agissait d'utiliser la précision double-étendue de l'Itanium, ses FMA, etc, alors que CRLibm se voulait portable et s'interdisait donc ces ressources. À

Algorithm Add233
In : a double-double number $a_h + a_l$ and a triple-double number $b_h + b_m + b_l$
Out : a triple-double number $r_h + r_m + r_l$
Preconditions on the arguments :

$$\begin{aligned} |b_h| &\leq 2^{-2} \cdot |a_h| \\ |a_l| &\leq 2^{-53} \cdot |a_h| \\ |b_m| &\leq 2^{-\beta_o} \cdot |b_h| \\ |b_l| &\leq 2^{-\beta_u} \cdot |b_m| \end{aligned}$$

Algorithm :

$$\begin{aligned} &(r_h, t_1) \leftarrow \textbf{Fast2Sum}\,(a_h, b_h) \\ &(t_2, t_3) \leftarrow \textbf{Fast2Sum}\,(a_l, b_m) \\ &(t_4, t_5) \leftarrow \textbf{Fast2Sum}\,(t_1, t_2) \\ &t_6 \leftarrow t_3 \oplus b_l \\ &t_7 \leftarrow t_6 \oplus t_5 \\ &(r_m, r_l) \leftarrow \textbf{Fast2Sum}\,(t_4, t_7) \end{aligned}$$

Theorem (Relative error of algorithm Add233)
Let be $a_h + a_l$ and $b_h + b_m + b_l$ the values taken in argument of algorithm **Add233**. Let the preconditions hold for this values.
So the following holds for the values returned by the algorithm r_h, r_m and r_l

$$r_h + r_m + r_l = ((a_h + a_l) + (b_h + b_m + b_l)) \cdot (1 + \epsilon)$$

where ϵ is bounded by

$$|\epsilon| \leq 2^{-\beta_o - \beta_u - 52} + 2^{-\beta_o - 104} + 2^{-153}.$$

The values r_m and r_l will not overlap at all and the overlap of r_h and r_m will be bounded by :

$$|r_m| \leq 2^{-\gamma} \cdot |r_h|$$

with

$$\gamma \geq \min\left(45, \beta_o - 4, \beta_o + \beta_u - 2\right).$$

FIG. 3.3 – Addition d'un nombre double-double avec un nombre triple-double, résultat en triple-double. La preuve est disponible dans [88, 1]. L'opération Fast2Sum est l'addition exacte de deux doubles, qui renvoie un double-double. On peut aussi la considérer comme la renormalisation d'un double-double. Le Fast2Sum ne fonctionne qu'à la condition que l'un des deux doubles en entrée soit plus grand en valeur absolue que l'autre, ce qui explique certaines des préconditions.

l'époque, une seule fonction était terminée dans CRLibm (justement l'exponentielle), et ses performances étaient très médiocres. Le but de ce stage était de déterminer dans quelle mesure ces faibles performances étaient dues à notre souci de portabilité (sans parler de notre relatif amateurisme de l'époque), et, plus important, quel était le surcoût intrinsèque de l'arrondi correct [31].

Considérant le pire cas à 2^{-158}, Lauter définit une stratégie en trois étapes. La première étape calculait en précision double-étendue (DE), soit 64 bits de mantisse, et se contentait de reprendre un code existant chez Intel et d'y ajouter un test d'arrondi. Utilisant un polynôme d'approximation précis à 64 bits, sa borne d'erreur totale était de l'ordre de 2^{-62}. La seconde étape calculait en précision double-double-étendue ou DDE, en utilisant les algorithmes bien connus de calcul en précision doublée, déjà évoqués [45, 84]. Sa borne d'erreur était de l'ordre de 2^{-125}. Enfin, la troisième étape devait être l'étape SCS de CRLibm, précise à 2^{-165}, ou bien juste une série de tests sur une table des pires cas connus.

Lauter avait construit une batterie de tests utilisant en particulier des nombres en entrée très difficiles à arrondir, fournis par Vincent Lefèvre. En testant sa seconde étape, il eut la surprise de constater qu'elle renvoyait toujours l'arrondi correct. Après maintes remises en cause des procédures de test et des valeurs des pires cas, il fut décidé qu'il consacrerait la fin de son stage à prouver que le code retournait effectivement toujours l'arrondi correct, et si possible que cette propriété n'était pas un coup de chance.

Le code de l'exponentielle en question était une succession de FMA en précision double-étendue, et il était naturel de chercher d'abord à expliquer la propriété par le FMA. C'était une fausse piste : le FMA n'est pas utile, car la propriété repose uniquement sur des propriétés de la fonction à implémenter (ici l'exponentielle, mais le cosinus, le sinus, et d'autres sont également concernées), et sur une séquence de quelques additions flottantes. Dans le code de Lauter, cette séquence était dissimulée dans des FMA éparpillés dans le code de la fonction.

3.4.2 Quand les mathématiques ordonnent les statistiques

La première partie de l'explication est que l'hypothèse de la répartition uniforme des bits de $f(x)$, pour la plupart des fonctions élémentaires f, est spectaculairement fausse pour les valeurs de x autour de zéro. En effet, pour ces fonctions, f admet souvent un développement limité simple : pour l'exponentielle il s'agit de

$$e^x = 1 + x + x^2/2 + O(x^3)$$

Il est alors facile de construire des pires cas artificiels, non aléatoires, par des annulations de bits entre un x négatif et un $x^2/2$ positif. Par exemple, le pire cas de l'exponentielle, qui nécessite une précision supérieure à 2^{-158}, se construit comme suit : soit $x = 2^{-52}(1 - 2^{-53})$. On a alors $x^2 = 2^{-104}(1 - 2 \cdot 2^{-53} + 2^{-106})$, et

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2} + ... \\ &= 1 + 2^{-52} - 2^{-105} \ + \ 2^{-105} - 2^{-157} + ... \\ &= 1 + 2^{-52} - 2^{-157} + ... \end{aligned}$$

On voit que e^x est un nombre machine $(1 + 2^{-52})$ suivi d'une très longue suite de 0 avant le premier bit significatif, ici négatif : cette valeur sera, au sens précédent, très difficile à arrondir pour les modes d'arrondi dirigés.

Cet exemple est simple à analyser, mais la même situation se produit pour de très nombreuses valeurs moins triviales. De plus, elle peut se produire également pour des développements limités de terme constant nul, comme $\sin(x) = x - x^3/6 + x^5/120 + O(x^7) = x \cdot (1 - x^2/6 + x^4/120 + O(x^6))$.

Une conclusion de ces premières observations est qu'il ne faut pas donner la «précision au pire cas nécessaire pour une fonction» de manière monolithique, mais plutôt par des énoncés détaillés de la forme suivante :

Théorème 1 *Soit x un nombre flottant double-precision et $y = e^x$ la valeur exacte de son exponentielle. Soit y^* une approximation de y telle que la distance relative entre y et y^* soit majorée par ϵ. On a alors :*
- *pour $2^{-54} \leq |x| < 2^{-44}$, si $\epsilon < 2^{-53-104-1} = 2^{-158}$, quel que soit le mode d'arrondi, arrondir y^* à un flottant double précision est équivalent à arrondir y ;*
- *pour $2^{-44} \leq |x| < 2^{-30}$, si $\epsilon < 2^{-53-84-1} = 2^{-138}$, quel que soit le mode d'arrondi, arrondir y^* à un flottant double précision est équivalent à arrondir y ;*
- *pour $|x| \geq 2^{-30}$, si $\epsilon < 2^{-53-59-1} = 2^{-113}$, quel que soit le mode d'arrondi, arrondir y^* à un flottant double précision est équivalent à arrondir y.*

On semble observer ici une relation linéaire entre l'exposant minimal de x sur l'intervalle considéré, et la précision requise au pire cas sur cet intervalle. Cette propriété est vraie pour toutes les fonctions pour lesquelles Lefèvre et Muller ont calculé des pires cas. Elle peut se résumer ainsi :

Propriété 3.1 *Pour chaque fonction étudiée*
- *soit la précision au pire cas requise pour l'arrondi correct sur tout le domaine de f est de l'ordre de 2^{-117} (la valeur prédite par l'argument statistique)*
- *soit il existe un intervalle I autour de zéro sur lequel la précision requise au pire cas est bien plus élevée que 2^{-117}. Toutefois, on est alors dans l'un des cas suivants, suivant le premier terme du développement de Taylor de la fonction en 0 :*
  - *Le premier terme du développment de Taylor en zéro est 1 (cas de e^x ou $\cos(x)$). Il existe alors une précision relative ϵ de l'ordre de 2^{-117} telle que, lorsqu'on approche $f(x) - 1$ par un polynôme $p(x)$ avec une précision relative meilleure que ϵ, alors sur tout I la précision relative de l'approximation de $f(x)$ par la somme non évaluée $1 + p(x)$ est meilleure que la précision au pire cas déterminée par Lefèvre et Muller.*
  - *Le premier terme du développement de Taylor est x (cas de $\sin(x)$, $\arcsin(x)$, etc...). Il existe alors une précision relative ϵ de l'ordre de 2^{-117} telle que, lorsqu'on approche $f(x) - x$ par un polynôme $p(x)$ avec une précision relative meilleure que ϵ, alors sur tout I la précision relative de l'approximation de $f(x)$ par la somme non évaluée $x + p(x)$ est meilleure que la précision au pire cas déterminée par Lefèvre et Muller.*

Ce que cette propriété dit en pratique, c'est qu'on n'aura jamais besoin de mener des calculs intermédiaires avec une précision très supérieure aux 2^{-117} prévus par l'argument statistique.

Nous avons dans la propriété précédente deux situations où des pires cas spéciaux apparaissent : pour des fonctions de type $1 + p(x)$, et pour des fonctions de type $x + p(x)$. En principe la situation serait la même si le premier terme du développement de Taylor était une autre constante c représentable sur un flottant, ou le produit de x par une autre puissance de

2, bref dès que le premier terme du développement de Taylor en 0 est représentable par un flottant. Pour les fonctions étudiées jusque là, nous n'avons rencontré que les situations de type $1 + p(x)$ et $x + p(x)$.

Il faut insister sur le fait que 3.1 est une propriété observée sur les valeurs de pires cas déterminés par énumération exhaustive par Lefèvre and Muller [97], et non pas une propriété démontrée par une preuve mathématique explicite. L'intuition statistique, illustrée pour l'exponentielle, est que pour $0 < x < 2^{-k}$, on a également $e^x - 1 < 2^{-k+1}$, donc l'écriture binaire de e^x commence par un 1 suivi de $k - 1$ zéros. Par conséquent l'argument statistique de répartition uniforme des bits doit être compté sur les bits de $p(x)$, mais certainement pas sur les bits de $f(x)$. Encore une fois il ne s'agit que d'une intuition : la preuve est dans l'énumération des pires cas, qui sont résumés dans le tableau 3.1.

$f(x)$	précision au pire cas brute	type	précision nécessaire en pratique
e^x	2^{-158}	$1 + p(x)$	2^{-115}
$\cos(x)$	2^{-142}	$1 + p(x)$	2^{-112}
$\sin(x)$	2^{-126}	$x + p(x)$	2^{-118}
$\arcsin(x)$	2^{-126}	$x + p(x)$	2^{-118}
$\tan(x)$	2^{-126}	$x + p(x)$	2^{-113}
$\arctan(x)$	2^{-126}	$x + p(x)$	2^{-113}
$\sinh(x)$	2^{-126}	$x + p(x)$	2^{-109}
$\text{arcsinh}(x)$	2^{-126}	$x + p(x)$	2^{-118}

TAB. 3.1 – Quelques fonctions dont l'évaluation bénéficie de la propriété 3.1.

Rétrospectivement, on peut se demander toutefois si la recherche exhaustive des pires cas était nécessaire autour de zéro : peut-être était-il possible de prouver que les pires cas ne peuvent être que des pires cas de Lauter, que l'on aurait construits. C'est d'ailleurs un argument similaire qui a été utilisé pour arrêter la recherche de pires cas à un certain exposant, en utilisant des considérations sur le développement de Taylor à l'ordre 0 ou 1 : par exemple pour l'exponentielle, si $0 < |x| < 2^{-54}$, on sait que la valeur correctement arrondie est 1, éventuellement plus ou moins un ulp selon le mode d'arrondi. On peut peut-être pousser cet argument à l'ordre 2.

3.4.3 Implications pratiques

Considérant le coût des opérations de calcul en précision étendue présentées en 3.3, il n'est pas anodin de réduire la précision des calculs intermédiaires ne serait-ce que d'une dizaine de bits, ce qui permettra de réduire le degré des polynômes utilisés.

Toutefois, le gain principal concerne les fonctions optimisées pour utiliser la précision double-étendue, disponible sur les jeux d'instruction IA32 et IA64. En effet, aucune des précisions de la colonne de droite de la table 3.1 ne dépasse 2^{-120}, ce qui signifie que l'arrondi correct de toutes ces fonctions pourra être déterminé uniquement en utilisant de l'arithmétique double-double-étendue. Nos expériences [87, 31, 33] montrent que le temps d'exécution de la seconde étape devient alors seulement deux fois celui de la première étape, en raison du partage de code entre les deux étapes.

Avec de telles performances sur l'étape précise, et en revenant à (3.1) page 44, il devient possible d'envisager une nouvelle stratégie d'optimisation : on pourrait rendre l'étape

rapide encore plus rapide en réduisant délibérément sa précision. En particulier, on peut utiliser une étape rapide bien moins précise que les fonctions au standard des `libm` actuelles, sachant que l'étape précise rattrapera si nécessaire. Cela ouvre la possibilité d'une fonction avec arrondi correct plus rapide en moyenne que son équivalent moins précis de bibliothèque standard.

Cela sera rarement possible, car les différents compromis entre taille de table, degré de polynôme, etc, définissent un espace d'implémentation très discret. Sur les trois fonctions utilisant la précision double-étendue que nous avons étudiées (exp, log et arctan), nous n'avons pas encore réussi à exploiter cette idée.

3.4.4 L'arrondi correct dans les cas de Lauter

La propriété 3.1 dit que si l'on a une approximation de $f(x) - 1$ ou de $f(x) - x$ précise à environ 2^{-120}, on a toute l'information nécessaire à l'arrondi correct de $f(x)$. Toutefois, elle ne dit pas comment calculer effectivement cet arrondi correct. Lauter était tombé par hasard sur la séquence suivante (bien cachée derrière des FMAs), qui retourne l'arrondi correct de $1 + p(x)$ dans le mode d'arrondi rx, lorsque $p(x)$ est approché par un double-double $p_h + p_l$:

$$\begin{aligned} t_h + t_l &\leftarrow \text{Fast2Sum64}(1, p_h) \\ r &\leftarrow t_l \overset{64}{\underset{rn}{\oplus}} p_l \\ z &\leftarrow t_h \overset{53}{\underset{rx}{\oplus}} r \end{aligned}$$

Ici les notations $\overset{64}{\underset{rn}{\oplus}}$ et $\overset{53}{\underset{rx}{\oplus}}$ representent respectivement l'addition avec arrondi au double-étendu le plus proche, et l'addition avec arrondi vers un double dans le mode d'arrondi désiré au final. Cette séquence totalise 5 additions flottantes dépendantes (le Fast2Sum64 s'implémente en 3 additions en double-étendue). La figure 3.4 montre dans le cas général les différentes mantisses qu'elle calcule.

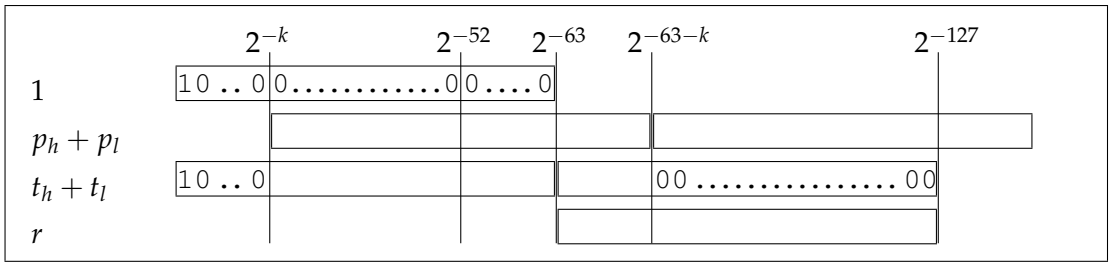


FIG. 3.4 – Arrondi de $1 + p_h + p_l$ vers un double, dans le cas $p_h \geq 0$.

Une preuve du comportement de cette séquence se trouve dans le rapport de recherche [29]. Cette preuve présente sans doute des lacunes, et Lauter est en train de la reprendre en la généralisant au cas où $p(x)$ est représenté comme un triple-double. Sa structure est la suivante :

- Soit $1 + p_h + p_l$ est facile à arrondir, plus précisément les bits de poids 2^{-53} à 2^{-63} de t_h décident l'arrondi. C'est le cas le plus général et le plus simple.

– Soit $1 + p_h + p_l$ est difficile à arrondir, c'est-à-dire que les bits de poids 2^{-53} à 2^{-63} de t_h sont nuls. Alors l'information qui décide l'arrondi final est uniquement le signe de r. Il suffit de montrer que ce signe est conservé par les arrondis réalisés.

Dans les détails, comme toujours dans ce genre de preuve, il faut faire une étude de cas soignée des différentes possibilités de signes, de chevauchement des mantisses, gérer le cas de l'arrondi pair sur les puissances de deux, etc. Le tout heureusement seulement pour les deux dernières opérations de la séquence, car le Fast2Sum est utilisé uniquement à travers ses propriétés.

Il faut enfin remarquer que cette séquence d'arrondi, relativement coûteuse, ne sera utilisée que pour les très petits arguments dans le code d'une fonction élémentaire.

3.5 Un exemple étendu : le logarithme de CRLibm

Le lecteur trouvera dans l'article [33] une description complète de l'implémentation d'une fonction dans CRLibm. Le logarithme a été un banc d'essai pour les différentes techniques évoquées jusqu'ici, et nous en avons des implémentations en SCS, en double-double-étendue, en triple-double, utilisant le FMA ou pas. La table 3.2 reprend les mesures de temps obtenues.

Bibliothèque	temps moyen	temps au pire cas
Sun's `libmcr`	1055	831476
IBM's `libultim`	677	463488
CRLibm portable, utilisant `scslib`	706	55804
CRLibm portable utilisant l'arithmétique triple-double	634	5140
CRLibm utilisant l'arithmétique double-étendue	339	4824
bibliothèque par défaut, sans arrondi correct	*323*	*8424*

TAB. 3.2 – Comparaisons de différentes implémentations du logarithme en double précision (temps en cycles sur un Pentium 4 Xeon / Linux Debian 3.1 / gcc 3.3).

Nous espérons que la version « CRLibm portable utilisant l'arithmétique triple-double » va voir ses performances s'améliorer sur les processeurs IA32 qui, de plus en plus nombreux, possèdent des unités SSE2, à mesure que les compilateurs apprendront à exploiter ce parallélisme. Anderson *et al.*, chez Intel, ont montré (mais en écrivant du code SSE2 explicite) qu'un facteur 2 était possible [4]. Toutefois, début 2007, la performance qu'on obtient avec gcc par des options du compilateur ciblant les unités flottantes SSE2 (`-msse2 -mfpmath=sse`) est équivalente, à quelques cycles près, à celle que l'on obtient par défaut.

3.6 Conclusion I : l'arrondi correct normalisé

À l'heure où j'écris ces lignes, la norme IEEE-754 est en train d'être révisée, et l'une de ses annexes mentionne que les fonctions élémentaires devraient *(should)* renvoyer l'arrondi correct de leur argument. L'arrondi correct n'y est pas strictement obligatoire, et heureusement. Il reste trop de fonctions pour lesquelles on ne sait pas encore le garantir : les fonctions

trigonométriques, les fonctions de deux variables `atan2` et `power`, les fonctions spéciales... Mais il est présenté comme l'idéal vers lequel on doit tendre, et les implémentations non correctement arrondies seront sous-standard. De plus, les implémentations seront tenues d'expliciter quelle est la qualité de leur résultat. Comme pour les étiquettes sur les yaourts, cette obligation de transparence va tirer dans le sens de la qualité : dès lors que l'un des acteurs majeur offrira l'arrondi correct par défaut[7], sans surcoût pour l'utilisateur, les autres s'aligneront.

C'est un grand progrès vers plus de portabilité et plus de confiance dans l'environnement flottant, et il n'est pas exagéré de dire que CRLibm a joué un rôle majeur dans cette avancée. CRLibm n'a pas été la première bibliothèque avec arrondi correct, celle de Ziv ainsi que, dans une autre catégorie, MPFR l'ont précédée. CRLibm y a ajouté, grâce aux travaux de Lefèvre et Muller, une borne au temps d'exécution qui est indispensable pour des fonctions tellement basiques qu'elles vont se retrouver dans la plupart des systèmes critiques. Elle a permis de mesurer quel était le surcoût intrinsèque de l'arrondi correct [31]. Elle a aussi apporté un souci de preuve qui, même s'il n'est sur bien des points qu'une prétention modeste, comme le prochain chapitre va le montrer, a le mérite d'exister, dans un contexte marqué par le souvenir du bug du Pentium. Nous avons d'ailleurs, dans les premiers temps, trouvé des erreurs dans la bibliothèque de Ziv.

Le comité chargé de la révision de la norme était au départ plutôt hostile à l'idée de mentionner l'arrondi correct dans la norme, estimant cette notion prématurée. Je crois qu'il y a eu un tournant avec la publication de l'article [31]. Beaucoup des membres influents du comité, à commencer par William Kahan, le père de la norme de 1985, ont assisté à la présentation de cet article. Je ne sous-estime pas pour autant le poids de Jean-Michel Muller, qui a lancé le projet CRLibm pour me le donner sur un plateau, et n'a par la suite jamais cessé de le porter de toute son énergie.

L'étape suivante sera d'intégrer CRLibm dans la `glibc` du projet GNU, qui définit la bibliothèque mathématique des systèmes d'exploitations libres comme Linux. Actuellement, c'est un patchwork de contributions diverses. On y trouve des morceaux de code dérivés de la FDLibm de Sun, d'autres dérivés de la bibliothèque de Ziv, qui a atterri là depuis qu'elle n'est plus officiellement supportée par IBM, du vieil assembleur Itanium offert par Intel, etc. La sélection se fait au cas par cas des systèmes. Si le code de CRLibm n'est pas toujours au niveau des standards GNU, par d'autres côtés, notamment la qualité de sa documentation, CRLibm est très au dessus du lot. L'inclusion se fera par morceaux, fonction par fonction.

Mais pendant que de l'énergie est consacrée à ces questions politiques, la science avance aussi, notamment sous l'impulsion de Christoph Lauter qui est en train de grignoter petit à petit la fonction `pow` qui calcule x^y [89].

3.7 Conclusion II : vers des fonctions d'intervalle parfaites

L'arithmétique d'intervalle [106] est une technique qui permet d'obtenir des garanties sur les calculs réalisés. Une valeur réelle n'est plus représentée par une approximation sous forme d'un nombre machine, mais par un intervalle. Tous les intervalles intervenant dans un calcul doivent posséder la *propriété d'inclusion* : la valeur représentée est incluse avec

[7] C'est déjà le cas de certaines fonctions, dérivées de la bibliothèque de Ziv, dans la `glibc` du projet GNU, pour les architectures Power/PowerPC et AMD64 notamment.

certitude dans l'intervalle la représentant. Les intervalles utilisés sont le plus souvent des couples de flottants, éventuellement en précision arbitraire [121].

Pour garantir cette propriété d'inclusion dans un calcul complexe, il suffit de n'utiliser que des opérateurs qui la propagent. Par exemple, l'implémentation pour l'arithmétique d'intervalle d'une fonction élémentaire prend un intervalle I_e et retourne un intervalle I_s, et doit assurer la propriété $f(I_e) \subset I_s$: pour tout réel x de I_e, son image par f, réelle également, doit appartenir à I_s. Nous insistons dans cette formulation sur le fait que l'arithmétique d'intervalle, bien que son support soit fini, permet de garantir des résultats sur des nombres réels.

Nous nous sommes intéressés à l'implémentation de fonctions élémentaires d'intervalle d'abord parce que c'est l'application la plus évidente (et peut-être la seule[8]) des fonctions avec arrondi correct dirigé dans CRLibm. En effet, considérons pour commencer une fonction croissante sur tout son domaine, comme l'exponentielle. L'image d'un intervalle $I_e = [x_i, x_s]$ par une telle fonction est $f(I_e) = [f(x_i), f(x_s)]$. Pour obtenir un intervalle $I_s = [y_i, y_s]$ vérifiant la propriété d'inclusion, il suffit de garantir que $y_i < f(x_i)$ et $y_s > f(x_s)$. L'intervalle le plus petit représentable vérifiant la propriété d'inclusion est obtenu par $y_i = \bigtriangledown(f(x_i))$ et $y_s = \bigtriangleup(f(x_s))$, où $\bigtriangledown$ et $\bigtriangleup$ représentent respectivement les fonctions d'arrondi vers le bas et vers le haut définies par la norme IEEE-754.

Les implémentations précédentes [120, 79, 123] des fonctions d'intervalle, ne disposant pas de l'arrondi correct des fonctions élémentaires, retournent un intervalle qui peut être plus grand que ce plus petit intervalle possible. Cela est parfaitement valide : en arithmétique d'intervalles, l'important est la garantie de la propriété d'inclusion plus que la précision du résultat. De ce point de vue, on peut préférer l'implémentation relativement imprécise et lente, mais munie d'une preuve publiée et mécanisée, de `fi_lib` [79] ou d'Intlab [123] à celle plus efficace, tant en vitesse qu'en finesse, de Sun [120] pour laquelle une telle preuve, à notre connaissance, n'est pas disponible.

On peut obtenir un intervalle valide au sens de la propriété d'inclusion à partir de n'importe quelle approximation y de $f(x)$, à condition de disposer également d'une borne δ sur l'erreur totale commise lors de cette évaluation. Par exemple, $[\bigtriangledown(y-\delta), \bigtriangleup(y+\delta)]$ fera l'affaire. On voit donc que la problématique du calcul d'une fonction d'intervalle rejoint celle de l'évaluation d'une fonction avec arrondi correct : dans les deux cas, il y a nécessité de prouver une borne d'erreur totale sur l'implémentation de la fonction. De plus, plus la borne d'erreur est serrée, meilleure est la qualité de la fonction.

En collaboration avec Sergey Maidanov du laboratoire Intel de Nijniy Novgorod, nous avons étudié l'implémentation d'une fonction d'intervalle «parfaite» [35] :

- grâce à l'arrondi correct, elle retourne le plus petit intervalle possible ;
- la propriété d'inclusion est prouvée ;
- le temps d'exécution peut être très inférieur à celui de deux appels de la fonction scalaire, car le calcul des deux bornes est indépendant et peut exploiter mieux le parallélisme des processeurs modernes, comme remarqué par Priest [120].

Aucune des implémentations précédentes n'offre cette combinaison de qualités. L'un de nos arguments est le suivant : quitte à faire une preuve de borne d'erreur, ce qui est de toutes manières nécessaire pour une fonction d'intervalle, autant faire celle qui permet d'obtenir

[8] Avoir à l'esprit que l'arithmétique d'intervalle est l'application essentielle des arrondis dirigés permet de trancher [43] les cas où l'arrondi correct est en conflit avec une propriété importante de la fonction, comme par exemple le fait que l'arctangente doit appartenir à $[-\pi/2, \pi/2]$.

l'arrondi correct, et donc le meilleur intervalle possible. De plus cette preuve est la même que pour l'arrondi correct au plus près.

Une autre remarque importante apparue lors de ce travail est que la quête des pires cas n'est pas critique pour les fonctions d'intervalles : Pour garantir la propriété d'inclusion en l'absence de pires cas, il faudra juste un test d'arrondi supplémentaire à la fin de l'étape précise, dont le coût moyen sera négligeable en vertu de l'équation (3.1) p. 44. Ce test élargira l'intervalle retourné d'un ulp dans le cas (statistiquement très improbable) où l'on ne sait décider l'arrondi à la fin de l'étape précise. Par exemple, pour le sinus, dont on ne connaît les pires cas que sur un petit intervalle, on saura néanmoins construire ainsi une fonction d'intervalle «probablement parfaite». Ici l'adverbe «probablement» porte uniquement sur le fait que la fonction retourne toujours le plus petit intervalle possible, pas sur la garantie de la propriété d'inclusion.

Ce travail est resté expérimental jusqu'ici. Les expériences réalisées sur Itanium sont très satisfaisantes [35], car ce processeur dispose d'un support flottant proche de la perfection avec deux FMA dont on peut choisir le mode d'arrondi et la précision (jusqu'à la double étendue) instruction par instruction, alors qu'un changement de l'un de ces paramètres coûte plusieurs dizaines de cycles sur les processeurs classiques. L'Itanium offre également une pléthore de registres flottants, assez pour que le calcul en parallèle de deux fonctions élémentaires ne pose aucun problème. Malheureusement, l'exploitation de toutes ces spécificités ne peut, par définition, être portable.

Nous avons également essayé de convertir des fonctions portables de CRLibm en fonctions d'intervalles, et là les résultats, s'ils sont meilleurs tout de même que tout ce qui existe, ne présentent pas la performance prédite par Priest. Nous avons constaté que, sur un Pentium, l'approche naïve[9] qui consiste à calculer $\triangledown(f(x_i))$ puis $\triangle(f(x_s))$, est plus rapide que l'approche qui essaye de calculer les deux bornes en parallèle. La raison en est sans doute le petit nombre de registres du jeu d'instruction IA32, qui oblige le compilateur à stocker de nombreuses variables en mémoire *(spill)* au lieu de les conserver dans des registres.

Ces travaux sont un peu en sommeil actuellement. L'avenir est à une approche intégrée que jusqu'ici seul Sun propose [2] : il faut un support dans les processeurs qui permet de choisir dynamiquement le mode d'arrondi, comme font l'Itanium et les SPARC récents [133], et il faut pour l'exploiter une collaboration harmonieuse des langages, des compilateurs et des systèmes d'exploitation qui ne va pas de soi. L'approche intégrée de Sun est séduisante, mais malheureusement isolée. Elle n'est possible que parce que Sun est un constructeur (sans doute le dernier) qui maîtrise toute la chaîne : processeur, calculateur, système d'exploitation et compilateur. Ils ont toutefois du apporter des extensions non standard aux langages pour y ajouter le support des intervalles. Pour un support des intervalles efficace et portable sur des systèmes plus hétérogènes, il faudra définir des normes pour l'arithmétique d'intervalle, en commençant par les langages de programmation. Les progrès dans cette direction sont lents.

Un impact positif de ces travaux a toutefois été de clarifier des questions d'interfaces qui se posaient à la comparaison des trois bibliothèques offrant l'arrondi correct dirigé des fonctions élémentaires. Par exemple, quelle est la bonne interface pour spécifier le mode d'arrondi ? Pour MPFR [107], c'est un argument à la fonction. Pour CRLibm, on a une fonction différente par mode d'arrondi. Pour `libmcr`, la petite dernière, publiée par Sun en 2002,

[9] Nous escamotons ici les questions liées à la réduction d'argument, pour que par exemple la fonction sinus renvoit $[-1, 1]$ pour tout intervalle d'entrée plus grand que 2π.

le mode d'arrondi est lu dans le registre de drapeaux du processeur. Quel est le bon choix ? Nous proposons à présent de répondre ainsi : à quoi servent les arrondis dirigés ? Essentiellement à faire des intervalles. Quelle interface pour les arrondis dirigés veulent les gens qui font des intervalles ? Eh bien, plutôt que des fonctions avec arrondi dirigé, ils voudront sans doute directement des fonctions d'intervalles, pour lesquelles il n'y a pas de question d'interface. D'ailleurs, depuis Priest, on sait que ce sera plus efficace.

CHAPITRE 4

Calcul d'erreur et validation

It makes me nervous to fly on airplanes since I know they are designed using floating-point arithmetic.

A. Householder

Ce chapitre présente la problématique transversale de la validation d'un code ou d'un circuit numérique. Le plus souvent, les bonnes propriétés à prouver sont de deux types :

- On veut montrer que les valeurs manipulées restent dans un certain intervalle. Parfois on cherche ainsi à montrer une propriété intéressante de l'objet simulé numériquement, mais souvent il s'agit juste de montrer l'absence de débordements de capacités : on veut montrer que le matériel reste dans les conditions dans lesquelles il fait un calcul utile, autrement dit éviter les situations absorbantes (les NaN et les infinis de la virgule flottante IEEE-754), la perte de précision du format (les subnormaux IEEE-754), ou les fonctionnements aberrants (le modulo 2^n de l'arithmétique entière n bits).
- On veut montrer que la valeur calculée n'est pas trop loin de la «vraie» valeur.

Une première idée forte est que ces deux situations se ramènent à montrer qu'une valeur est dans un certain intervalle : dans le premier cas il s'agit de montrer qu'une variable du programme ou qu'un signal du circuit reste dans certaines limites. Dans le second cas, il s'agit de montrer qu'une erreur, définie comme à la section 1.1 par une différence entre valeur calculée et valeur «vraie», reste dans un certain intervalle de confiance.

Cette idée que la plupart des propriétés qu'on voudra prouver se ramènent à «une valeur appartient à un intervalle» est au cœur de l'arithmétique d'intervalle [106] déjà évoquée en 3.7, et en particulier de l'outil Gappa [32] développé par Guillaume Melquiond durant sa thèse [105]. Ce chapitre fera beaucoup de réclame pour cet outil, et en tout cas pour les principes qui le sous-tendent. Par contre il parlera peu de Gappa lui-même, essentiellement à travers un exemple en 4.5. Le lecteur intéressé se reportera directement à la thèse de Melquiond.

Avant cela il faut toutefois commencer par définir la «vraie» valeur que doit approcher le programme ou le circuit. Le lecteur attentif aura remarqué qu'on utilise désormais l'expression «vraie valeur», alors qu'au chapitre 1 on parlait de «résultat idéal». C'est anecdotique, mais cela marque une volonté de se rapprocher du monde de la logique.

4.1 Bien poser la question

Pour un code numérique, par exemple de simulation scientifique, il n'est pas toujours possible de définir la «vraie» valeur. Par exemple elle peut ne pas être définie de manière unique (simulation de phénomènes quantiques, chaotiques [141, 36] ou tout simplement statistiques). Plus prosaïquement, plus le code est gros, plus il est difficile de définir une «vraie» valeur pour chaque variable interne. On simule des équations décrivant des modèles qui sont des approximations de la réalité, et la simulation réalise toutes sortes de discrétisations et d'approximations, avant même de parler des arrondis réalisés lors du calcul. Pour de tels codes, la meilleure validation restera pour longtemps la confrontation expérimentale du résultat calculé avec la réalité.

Les fonctions mathématiques, même si leur code peut être complexe, ont au moins cette grande qualité : non seulement la «vraie» valeur existe, mais en plus elle est définie mathématiquement de manière très pure. Pour une entrée x, qui est un rationnel bien défini[1], il existe une valeur de $f(x)$ qui est un réel bien défini. Par conséquent les erreurs absolue $\widetilde{f(x)} - f(x)$ et relative $\dfrac{\widetilde{f(x)} - f(x)}{f(x)}$ sont également bien définies en termes mathématiques simples.

Si l'on veut mécaniser ces calculs d'erreur, il faut toutefois également mécaniser cette définition de la «vraie» valeur. La technique utilisée par Harrison [73, 74] est de définir la fonction par son développement de Taylor à un ordre arbitraire. D'une part, ce développement de Taylor peut lui-même être construit à partir des propriétés les plus fondamentales de la fonction, comme les règles de dérivations, et d'arithmétique rationnelle, donc représentable. D'autre part, il permettra d'évaluer la fonction à une précision arbitraire – encore que la majoration du reste de Taylor ne soit pas toujours évidente. C'est ainsi qu'est construit le lien entre le monde formel et le monde numérique.

Une fois qu'on a une définition formelle de l'entrée et de la sortie de l'opérateur, la construction d'une preuve de propriété du code requiert de définir formellement la «vraie» valeur également pour les variables intermédiaires du code. C'est là que les choses se compliquent, puisque ces valeurs portent typiquement plusieurs couches d'approximation et d'arrondi. La manière dont elles sont calculées est le fruit de toute l'expertise du programmeur, mais cette expertise n'est pas écrite dans le code. Nous allons illustrer cela sur un exemple.

En fait, un des enseignements principaux que je retire de quatre années à écrire des preuves pour CRLibm, et en particulier du travail sur Gappa, est le suivant : une fois que l'on a réussi à définir formellement ce que sont supposées calculer les différentes valeurs intermédiaires d'un programme ou d'un circuit, on a fait le plus gros du travail. Il devient alors facile de formaliser les subtilités du programme, et ensuite il ne reste plus qu'à combiner calculatoirement des intervalles, ce qui peut parfaitement être automatisé, comme le

[1] Nous allons systématiquement considérer qu'un nombre flottant ne représente que lui-même. On pourrait vouloir considérer par exemple qu'un nombre flottant représente un intervalle de valeurs, de taille 1 ulp, autour de lui-même, mais cela est inutile et sans doute dangereux. Certes, il peut être intéressant de considérer un intervalle arbitraire en entrée, et d'étudier sa propagation à travers l'opérateur. Cet intervalle pourra par exemple correspondre, dans la simulation d'un dispositif physique, à l'intervalle d'incertitude associé à la grandeur physique mesurée. Dans un code numérique étudié morceau par morceau, cet intervalle pourra correspondre à l'intervalle de confiance du calcul alimentant l'opérateur. Dans les deux cas, cet intervalle sera très grand devant l'ulp : peu d'instruments au monde peuvent faire une mesure précise à 10^{-15}.

montre l'outil Gappa.

4.2 Un exemple de trois lignes

Pour illustrer la difficulté de définir ce que calcule chaque variable du programme, considérons le morceau de code suivant, tiré du sinus de CRLibm, qui réalise une simple évaluation par Horner d'un polynôme impair proche de l'approximation de Taylor du sinus (son coefficient d'ordre 1 est égal à 1).

```
1  yh2 = yh*yh;
2  ts = yh2 * (s3 + yh2*(s5 + yh2*s7));
3  Fast2Sum(*psh,*psl,   yh, yl+ts*yh);
```

Listing 4.1 – Trois lignes de C.

Toutes les variables sont des flottants double-précision. Ce code prend en entrée un double-double (`yh`,`yl`) qui est une approximation de l'argument réduit (car la réduction d'argument trigonométrique, impliquant le nombre transcendant π, est nécessairement approchée). On suppose qu'on a déjà calculé une borne ϵ_{argred} sur l'erreur relative de cette réduction d'argument, et qu'on a également déjà prouvé que cet argument réduit était dans un certain intervalle, mettons $[-2^{-7}, 2^{-7}]$.

En principe, l'entrée étant un double-double, on devrait réaliser toute l'évaluation polynomiale en arithmétique double-double, et calculer donc

$$(\mathtt{yh} + \mathtt{yl}) + (\mathtt{yh} + \mathtt{yl})^3 \times (\mathtt{s3} + (\mathtt{yh} + \mathtt{yl})^2 \times (\mathtt{s5} + (\mathtt{yh} + \mathtt{yl})^2 \times (\mathtt{s7}))).$$

Ce code utilise une approximation de cette expression, qui est la suivante : On va se contenter d'arithmétique double-précision (et non pas double-double) pour toutes les itérations de Horner sauf la dernière. Ainsi, dans ces itérations, `yl` sera négligé, et les coefficients `s3` à `s7` seront stockés sur des flottants double-précision. L'expression devient :

$$(\mathtt{yh} + \mathtt{yl}) + \mathtt{yh}^3 \times (\mathtt{s3} + \mathtt{yh}^2 \times (\mathtt{s5} + \mathtt{yh}^2 \times \mathtt{s7})).$$

En fait on préfère reparenthèser cette expression comme suit, «pour mettre ensemble tous les petits termes » (nous y reviendrons ci-dessous) :

$$\mathtt{yh} + \left(\mathtt{yl} + \mathtt{yh} \times \mathtt{yh}^2 \times (\mathtt{s3} + \mathtt{yh}^2 \times (\mathtt{s5} + \mathtt{yh}^2 \times \mathtt{s7}))\right).$$

Dans cette écriture, seul le premier $+$ doit être précis : il sera exact, c'est le Fast2Sum (addition exacte de deux doubles renvoyant un double-double). Le reste sera de l'arithmétique flottante double-précision. On a ainsi évité toute utilisation d'arithmétique double-double.

Il s'agit à présent de garantir que le résultat sera proche du sinus. On va donc chercher à borner l'erreur totale due à ce code. En termes plus mathématiques, il s'agit de calculer une borne fine de la différence entre le vrai sinus et le double-double résultat.

Donnons d'abord une intuition non rigoureuse des ordres de grandeur. On a $\mathtt{yh} + \mathtt{yl} \in [-2^{-7}, 2^{-7}]$. Par conséquent `yh` est au maximum de l'ordre de 2^{-7}, donc `yh2` sera borné par 2^{-14}. Si l'on calcule en double le polynôme de Horner en `yh2`, on fera une erreur seulement sur les bits de poids faible de `ts`. Comme `ts` est de l'ordre de `yh2` (plus petit en fait puisque $\mathtt{s3} < 1$) on aura $\mathtt{ts} * \mathtt{yh} < 2^{-14}\mathtt{yh}$. De même, par l'hypothèse que (`yh`,`yl`) est un double-double, on a $\mathtt{yl} < 2^{-53}\mathtt{yh}$.

En résumé, toutes les erreurs d'arrondi du calcul en double `yl+ts*yh`, y compris celles du calcul de `ts`, sont sur les derniers bits d'un flottant qui est plus petit que 2^{-14}`yh`. Le Fast2Sum réalise la somme exacte de ces deux doubles (`yh` et `yl+ts*yh`), et renvoie le résultat sous forme d'un double-double. Ce sera bien le diable si toutes les erreurs réalisées par ce calcul ont fait perdre 4 bits de précision. Cette évaluation polynomiale sera donc précise au moins à 2^{-53-10}.

Nous avons à dessein utilisé du vocabulaire peu scientifique. C'est comme cela que l'expert construit, et parfois explique son code. Les premières preuves de CRLibm ressemblaient d'ailleurs à cela [39]. Mais dans cette explication nous avons encore escamoté plusieurs sources d'approximation. Par exemple, considérons juste la variable `yh2`. Il est clair qu'elle est supposée approcher y^2, mais c'est une approximation à plusieurs niveaux :

- y était approché par `yh` + `yl` à la précision relative ϵ_{argred}
- `yh` + `yl` est approché par `yh` en négligeant `yl`
- `yh`^{2} est approché par `yh2=yh*yh`, il y a une erreur d'arrondi flottante.

Autre exemple, il ne faut pas oublier que le polynôme n'est qu'une approximation de la fonction sinus, avec une borne d'erreur $\epsilon_{\text{approxts}}$.

La figure 4.1 reproduit les pages correspondant à la preuve dans CRLibm d'une borne de l'erreur commise par ces trois lignes. Cette preuve des fonctions trigonométriques fut la dernière que nous avons eu la prétention de réaliser à la main. Il faut d'ailleurs ajouter que ces pages de LaTeX sont assorties d'autant de pages de code Maple mécanisant autant que faire se peut ce calcul d'erreur. Le lecteur qui aura le courage de s'y plonger y trouvera certainement de nombreuses erreurs. Il y a aussi des faiblesses méthodologiques. Par exemple, le calcul d'erreur fait apparaître plusieurs normes infinies $||.||_\infty$, dont l'implémentation dans Maple n'a pas la prétention d'offrir une borne garantie, mais juste une approximation.

Voici posée la motivation de ce chapitre : comment rédiger la preuve d'une fonction de 100 lignes de code s'il faut deux pages pour trois lignes ? Qui la lira ? Qui la maintiendra si le code change ? Quelle confiance peut-on lui accorder ?

4.3 En quoi a-t-on confiance ?

En ce qui concerne la confiance, il faut tout de même disposer d'une base que l'on considère fiable, et qui est un support suffisant des normes IEEE-754 et C99 [5, 82]. Il faut aussi avoir confiance dans les calculs de pires cas réalisés par Lefèvre et Muller.

Pour ce dernier point, il est trivial de vérifier que les valeurs trouvées sont effectivement très difficiles à arrondir. La question qui reste est : Lefèvre et Muller n'en ont-ils pas oublié ? Il y a eu un certain nombre de vérifications (notamment en lançant des tests sur la fonction, puis sur son inverse, et en utilisant différents algorithmes sur la même fonction) mais les algorithmes sont suffisamment complexes pour laisser la place à un *bug* bien caché, y compris dans les aspects les moins mathématiques du code, comme les entrées/sorties.

Ces tests seront peut-être validés un jour par d'autres. En attendant, la propriété d'arrondi correct que nous voulons prouver est lâchement exprimée dans [33] comme suit :

Théorème 2 *Sous les hypothèses suivantes :*

- *la précision au pire cas requise pour déterminer l'arrondi correct du logarithme en double-précision est* 2^{-118} *;*
- *le code a été compilé par un compilateur respectant la norme C99 ;*

However, DoSinZero is slightly different in that it doesn't add a constant value at the end of the computation, as do the three others. Its error computation has to consider more relative errors than absolute errors. We therefore need to compute its error separately, which is done in the following section.

This section 9.3.1 should also help understanding the method used to write the Gappa input file given in Section 9.3.2.

9.3.1 DoSinZero

Upon entering DoSinZero, we have in $y_h + y_l$ an approximation to the ideal reduced value $\hat{y} = x - k\frac{\pi}{256}$ with a relative accuracy $\varepsilon_{\text{argred}}$:

$$y_h + y_l = (x - k\frac{\pi}{256})(1 + \varepsilon_{\text{argred}}) = \hat{y}(1 + \varepsilon_{\text{argred}}) \tag{9.6}$$

with, depending on the quadrant, $\sin(\hat{y}) = \pm\sin(x)$ or $\sin(\hat{y}) = \pm\cos(x)$ and similarly for $\cos(\hat{y})$. This just means that $\hat{y}$ is the ideal, errorless reduced value.

In the following we will assume we are in the case $\sin(\hat{y}) = \sin(x)$, (the proof is identical in the other cases), therefore the relative error that we need to compute is

$$\varepsilon_{\text{sinkzero}} = \frac{(\texttt{*psh} + \texttt{*psl})}{\sin(\texttt{x})} - 1 = \frac{(\texttt{*psh} + \texttt{*psl})}{\sin(\hat{y})} - 1 \tag{9.7}$$

Listing 9.12: DoSinZero

```
1  yh2 = yh*yh;                                  \
2  ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));    \
3  Add12(*psh,*psl,   yh, yl+ts*yh);             \
```

One may remark that we almost have the same code as we have for computing the sine of a small argument (without range reduction) in Section 9.4.5. The difference is that we have as input a double-double $\texttt{yh} + \texttt{yl}$, which is itself an inexact term.

At Line 4, the error of neglecting y_l and the rounding error in the multiplication each amount to half an ulp: $\texttt{yh2} = \texttt{yh}^2(1 + \varepsilon_{-53})$, with $\texttt{yh} = (\texttt{yh} + \texttt{yl})(1 + \varepsilon_{-53}) = \hat{y}(1 + \varepsilon_{\text{argred}})(1 + \varepsilon_{-53})$

Therefore

$$\texttt{yh2} = \hat{y}^2(1 + \varepsilon_{\texttt{yh2}}) \tag{9.8}$$

with

$$\bar{\varepsilon}_{\texttt{yh2}} = (1 + \bar{\varepsilon}_{\text{argred}})^2(1 + \bar{\varepsilon}_{-53})^3 - 1 \tag{9.9}$$

Line 5 is a standard Horner evaluation. Its approximation error is defined by:

$$P_{\texttt{ts}}(\hat{y}) = \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\text{approxts}})$$

This error is computed in Maple as in 9.4.5, only the interval changes:

$$\bar{\varepsilon}_{\text{approxts}} = \left\| \frac{xP_{\texttt{ts}}(x)}{\sin(x) - x} - 1 \right\|_\infty$$

We also compute $\bar{\varepsilon}_{\text{hornerts}}$, the bound on the relative error due to rounding in the Horner evaluation thanks to the `compute_horner_rounding_error` procedure. This time, this procedure takes into account the relative error carried by `yh2`, which is $\bar{\varepsilon}_{\texttt{yh2}}$ computed above. We thus get the total relative error on `ts`:

$$\texttt{ts} = P_{\texttt{ts}}(\hat{y})(1 + \varepsilon_{\text{hornerts}}) = \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\text{approxts}})(1 + \varepsilon_{\text{hornerts}}) \tag{9.10}$$

The final `Add12` is exact. Therefore the overall relative error is:

$$\begin{aligned}
\varepsilon_{\text{sinkzero}} &= \frac{((\texttt{yh} \otimes \texttt{ts}) \oplus \texttt{yl}) + \texttt{yh}}{\sin(\hat{y})} - 1 \\
&= \frac{(\texttt{yh} \otimes \texttt{ts} + \texttt{yl})(1 + \varepsilon_{-53}) + \texttt{yh}}{\sin(\hat{y})} - 1 \\
&= \frac{\texttt{yh} \otimes \texttt{ts} + \texttt{yl} + \texttt{yh} + (\texttt{yh} \otimes \texttt{ts} + \texttt{yl}).\varepsilon_{-53}}{\sin(\hat{y})} - 1
\end{aligned}$$

Let us define for now

$$\delta_{\text{addsin}} = (\texttt{yh} \otimes \texttt{ts} + \texttt{yl}).\varepsilon_{-53} \tag{9.11}$$

Then we have

$$\varepsilon_{\text{sinkzero}} = \frac{(\texttt{yh} + \texttt{yl})\texttt{ts}(1 + \varepsilon_{-53})^2 + \texttt{yl} + \texttt{yh} + \delta_{\text{addsin}}}{\sin(\hat{y})} - 1$$

Using (9.6) and (9.10) we get:

$$\varepsilon_{\text{sinkzero}} = \frac{\hat{y}(1 + \varepsilon_{\text{argred}}) \times \frac{\sin(\hat{y}) - \hat{y}}{\hat{y}}(1 + \varepsilon_{\text{approxts}})(1 + \varepsilon_{\text{hornerts}})(1 + \varepsilon_{-53})^2 + \texttt{yl} + \texttt{yh} + \delta_{\text{addsin}}}{\sin(\hat{y})} - 1$$

To lighten notations, let us define

$$\varepsilon_{\sin 1} = (1 + \varepsilon_{\text{approxts}})(1 + \varepsilon_{\text{hornerts}})(1 + \varepsilon_{-53})^2 - 1 \tag{9.12}$$

We get

$$\begin{aligned}
\varepsilon_{\text{sinkzero}} &= \frac{(\sin(\hat{y}) - \hat{y})(1 + \varepsilon_{\sin 1}) + \hat{y}(1 + \varepsilon_{\text{argred}}) + \delta_{\text{addsin}} - \sin(\hat{y})}{\sin(\hat{y})} \\
&= \frac{(\sin(\hat{y}) - \hat{y}).\varepsilon_{\sin 1} + \hat{y}.\varepsilon_{\text{argred}} + \delta_{\text{addsin}}}{\sin(\hat{y})}
\end{aligned}$$

Using the following bound:

$$|\delta_{\text{addsin}}| = |(\texttt{yh} \otimes \texttt{ts} + \texttt{yl}).\varepsilon_{-53}| < 2^{-53} \times |y|^3/3 \tag{9.13}$$

we may compute the value of $\bar{\varepsilon}_{\text{sinkzero}}$ as an infinite norm under Maple. We get an error smaller than 2^{-67}.

9.3.2 DoSinNotZero and DoCosNotZero

The proof would here be much longer than the previous one, in the same spirit. It would therefore be much more error prone. We probably would be even less confident in such a proof than if it was generated automatically using experimental software. Therefore, let us do just that: We will use `Gappa` (see `http://lipforge.ens-lyon.fr/projects/gappa/`), another development of the Arenaire project to automate the proof of numerical properties, including an interface to the automatic theorem prover Coq. Gappa will assist us in computing bounds (in the form of intervals) for all the errors entailed by our code.

Here we need to compute the error bound for the following straight line of code.

Listing 9.13: DoSinNotZero

```
1  yh2 = yh*yh ;
2  ts = yh2 * (s3.d + yh2*(s5.d + yh2*s7.d));
3  tc = yh2 * (c2.d + yh2*(c4.d + yh2*c6.d ));
4  Mul12(&cahyh,h,&cahyh,l, cah, yh);
5  Add12(thi, tlo, sah,cahyh,h);
6  tlo = tc*sah+(ts*cahyh.h+(sal+(tlo+(cahyh.l+(cal+yh * cah+yl))))) ;
7  Add12(*psh,*psl,  thi, tlo);
```


FIG. 4.1 – `crlibm.pdf` version 1.0β1, pages 103 et 104.

– *par défaut, les calculs dans le système sont conformes à la norme IEEE-754 en double précision (resp. en précision double-étendue) ;*

alors pour tout flottant double-précision en entrée `x`*, l'appel du* `log(x)` *de* CRLibm *retourne le logarithme de* `x` *correctement arrondi au plus près.*

On a les mêmes théorèmes pour les arrondis dirigés.

Voila pour la confiance en le code et les outils qui le manipulent. En ce qui concerne la confiance dans sa preuve, nous suivons l'auteur de Gappa : nous n'allons pas faire confiance à Gappa, qui calcule des intervalles d'une manière pleine d'heuristiques et peut-être (cela s'est vu) de bugs. Par contre, nous ferons confiance à l'assistant de preuve Coq [80] qui sera chargé de la vérification de ce calcul. Le monde de la preuve formelle semble faire confiance à Coq du fait de son code ouvert, tout petit, et très stable.

Pour le compilateur comme pour Coq, il ne s'agit pas de reporter toute la responsabilité finale de la confiance en la preuve sur un autre outil, mais de bien séparer les responsabilités. Nous avons été très heureux de reporter des bugs pour certains compilateurs, mais nous refusons par principe de rendre notre code plus complexe pour les contourner.

Enfin, il reste à avoir confiance que la preuve correspond bien au code. Idéalement on aimerait un outil qui prend notre code, avec les propriétés de ses entrées, et calcule automatiquement l'erreur qu'il fait. Cela est clairement impossible. Dans notre exemple d'évaluation polynomiale, il y a trop d'approximations dans le code, et surtout elles sont trop implicites pour qu'un outil puisse seul les retrouver. L'outil Gappa nous demandera de les exprimer explicitement, et d'une manière formelle. Pour ce type de code, c'est l'approche qui offre le meilleur compromis entre automatisation et puissance.

Mais même Gappa est un outil totalement distinct du compilateur, et dont l'entrée est totalement distincte du code C. La confiance que la preuve Gappa prouve bien quelque chose sur le code C ne va pas de soi. Nous avions le même problème avec la preuve LaTeX. S'il est possible d'augmenter cette confiance par de la discipline autour de conventions bien établies, la technique la plus convaincante est d'*automatiser* : soit la production du code depuis la preuve (ce que fait Coq, mais pas encore pour du code flottant), soit la production de la preuve depuis le code (mais alors il faut annoter le code, ce que proposent des outils comme Fluctuat), soit enfin la production conjointe du code et de sa preuve à partir d'une spécification de plus haut niveau. C'est vers cette dernière approche que tend CRLibm. Remarquons la convergence entre cette approche de génération de code/preuve et les générateurs de circuits du chapitre 2.

4.4 Des progrès de l'automatisation

Depuis les premières preuves de CRLibm, par exemple celle de l'exponentielle dans la thèse de Defour [39], nous avons progressé vers toujours plus de mécanisation de la production du code et de sa preuve.

Au départ, il s'agissait juste d'écrire pour chaque fonction un script Maple qui calculait les différentes constantes du code (des valeurs de tables et coefficients polynomiaux aux bornes d'erreur utilisées dans les tests d'arrondi) et les écrivait dans un fichier `.h` qui faisait partie du code de la fonction. L'idée était de limiter les erreurs de calcul humaines, et de pouvoir explorer plus facilement les compromis dans le code. La preuve de la borne d'erreur était rédigée en LaTeX et a constitué le document `CRLibm.pdf`, distribué avec la bibliothèque depuis sa première version bêta.

La seconde étape fut l'arrivée de Gappa. Nous avons alors cessé de rédiger des preuves papier. À la place, nous écrivons une preuve au format d'entrée de Gappa, qu'il vérifie ensuite. Nous continuons d'étendre `crlibm.pdf` avec des descriptions fines des algorithmes, et aussi avec certains aspects des preuves que Gappa ne sait pas gérer. Nous continuons d'écrire un script Maple qui produit les constantes, et les injecte dans le code C, et désormais aussi dans le code Gappa.

CRLibm a d'ailleurs beaucoup orienté le développement de Gappa. Nous avons réclamé des améliorations syntaxiques : pour augmenter la confiance dans les preuves, nous avons par exemple demandé que Gappa implémente les règles d'évaluation et de parenthésage du C. On peut ainsi écrire

```
ts <ieee64ne>= yh2 * (s3 + yh2*(s5 + yh2*s7));
```

et Gappa insèrera des fonctions d'arrondi partout où l'évaluation C99 le fait. Pour augmenter le champ d'application des preuves, nous avons discuté d'une syntaxe qui permet de définir une erreur relative arbitraire (comme celle de nos opérateurs double-double et triple-double). Nous avons aussi participé à l'amélioration du moteur de Gappa. Chaque fois que, en rédigeant une preuve, nous avions l'impression de faire un travail répétitif, nous avons discuté de la possibilité d'automatiser ce genre de travail. Cela se traduit par des preuves beaucoup plus courtes à présent que dans les premiers temps.

La troisième étape est le fait de Sylvain Chevillard et Christoph Lauter. Il s'agissait au départ d'implanter une norme infinie validée [20] ad-hoc pour CRLibm, puisque celle de Maple, comme déjà dit, n'a pas d'autre prétention que d'être une approximation de la norme infinie aussi bonne que possible. On a besoin d'une norme infinie pour borner l'erreur d'approximation de la fonction par un polynôme sur un intervalle donné.

L'algorithme de Chevillard et Lauter est basé sur des idées simples : il s'agit d'étudier les variations de la différence entre la fonction et son approximation. Cette différence est maximale soit sur les bords de l'intervalle d'entrée, soit lorsque sa dérivée s'annule. Les fonctions que nous étudions sont assez régulières pour qu'il suffise de chercher par dichotomie des intervalles, les plus petits possibles, où la dérivée change de signe. On calcule ensuite, par arithmétique d'intervalle, l'image de ces intervalles, et on en déduit un encadrement aussi fin que nécessaire de la norme infinie. La recherche des intervalles est faite par des algorithmes en C, mais on arrive à produire une preuve formelle de la norme infinie. Elle ne peut pas encore passer complètement dans Gappa, car Gappa n'a pas de notion de fonction élémentaire. Pour contourner ce problème, il faudra sans doute passer par des développements de Taylor. En attendant, on peut laisser, dans la preuve, des hypothèses sur l'image d'un intervalle par une fonction qui sont suffisamment simples pour ne pas entamer la confiance dans la preuve.

En parallèle, la thèse en cours de Chevillard porte sur la détermination des meilleurs polynômes d'approximation d'une fonction donnée, lorsque l'on contraint les coefficients à être des nombres machine [17]. Les outils développés à ce sujet se sont interfacés avec les programmes de Lauter, évoqués en 3.3, qui produisent un schéma d'évaluation en arithmétique mixte double, double-double et triple-double. En y ajoutant la norme infinie validée, on obtient une chaîne complète qui, à partir de la spécification de la fonction et de la précision requise, produit du code C et des preuves Gappa (pas encore tout à fait complètes, mais déjà parfaitement convaincantes) d'un schéma d'évaluation optimisé de la fonction par un polynôme en arithmétique mixte.

Cet outil spectaculaire est encore au niveau de prototype, mais a été utilisé pour l'arcsinus de CRLibm. Il montre bien le progrès dans la direction que nous suivons : produire, de

plus en plus automatiquement, une preuve en parallèle du code, preuve qui au final pourra être validée par Coq en passant par l'outil Gappa.

4.5 Une preuve Gappa

Ce chapitre ne serait pas complet sans une preuve en Gappa. Nous allons décrire celle des trois lignes prises ci-dessus en exemple. Toutefois, le fichier Gappa complet fait plus de 150 lignes, et nous ne le détaillerons pas d'un bout à l'autre. Nous allons nous contenter d'en extraire les parties qui correspondent à la spécification, et un exemple de la démarche à conduire pour mener la preuve à bien. Nous voulons montrer que, bien que la preuve obtenue soit encore plus longue que la preuve papier de la figure 4.1, il est possible de la construire de manière incrémentale.

4.5.1 L'outil Gappa dans les grandes lignes

Gappa utilise l'arithmétique d'intervalle pour prouver des propriétés sur des réels. Une preuve, au sens de Gappa, est la preuve de l'encadrement d'une valeur par un intervalle.

De telles preuves seront construites à partir de toute une bibliothèque de théorèmes de base décrivant par exemple comment les opérations de base propagent de tels encadrements en arithmétique d'intervalle. Un exemple simplifié d'un tel théorème est :

$$a \in [a_g,\ a_d] \wedge b \in [b_g,\ b_d] \Longrightarrow a - b \in [a_g - b_d,\ a_d - b_g].$$

La bibliothèque contient aussi de nombreux théorèmes utiles sur les flottants, par exemple encadrant les erreurs absolues et relatives des différentes opérations. Ces théorèmes tiennent compte des cas spéciaux comme les nombres sous-normaux de la norme IEEE-754. Tous ces théorèmes sont prouvés en Coq.

Lorsque l'on demande à Gappa de prouver un encadrement, l'outil cherche à le dériver des théorèmes de sa bibliothèque. Pour s'y ramener, il dispose de différentes heuristiques qui tentent de construire une telle dérivation. À un instant donné, Gappa peut disposer de plusieurs manières de calculer un encadrement. Il sélectionnera au final celle qui donne l'intervalle le plus fin, ou la dérivation la plus courte d'un encadrement satisfaisant. Nous illustrerons cette part automatique de l'exploration sur notre exemple.

Toutefois, la combinatoire de cette exploration est bien trop grande pour que Gappa puisse la couvrir exhaustivement. Il faut donc aider l'outil en lui donnant des indices. Il s'agit de suggestions de réécritures des expressions mathématiques qu'il ne sait pas encadrer. C'est là que l'on va exprimer et formaliser les intuitions que l'on a utilisées pour construire le code. En termes d'arithmétique d'intervalle, il s'agit d'exprimer les corrélations entre les différentes variables pour réduire le phénomène, habituel en arithmétique d'intervalle, de *décorrélation*. L'archétype de la décorrélation est le calcul de $x - x$ en utilisant le théorème de soustraction d'intervalle ci-dessus : si $x \in [-1, 1]$, on calculera par arithmétique d'intervalle $x - x \in [-2, 2]$ et non pas $[0, 0]$. Nous aurons des variations plus subtiles sur ce thème, avec la différence de deux nombres très proches, typiquement deux expressions qui diffèrent par un arrondi. Il s'agira d'aider Gappa à encadrer une telle différence au moyen d'un théorème sur les arrondis, et non pas uniquement au moyen des théorèmes généraux de l'arithmétique d'intervalles. Nous allons également donner un exemple.

Lorsque Gappa a pu prouver l'encadrement demandé, il sait en produire une preuve en Coq qui s'appuie sur les preuves des théorèmes de sa bibliothèque.

Concrétisons à présent cet exposé sur la preuve d'un encadrement du résultat des trois lignes de notre exemple de la page 63.

4.5.2 L'énoncé du problème dans Gappa

Pour commencer, voici la partie du fichier Gappa qui décrit le code sur lequel on veut encadrer des valeurs. Pour l'obtenir, nous avons essentiellement recopié notre code C et inséré des `IEEEdouble`, opération que nous considérons sans risque.

```
1   #--------------- Définition d'un mode d'arrondi  ----------------
2   @IEEEdouble = float<ieee_64,ne>;
3
4   #--------------- Spécification de l'entrée  ---------------------
5   # yh+yl est un double-double (que l'on appelera Yhl)
6   yh = IEEEdouble(Yhl);
7   yl = Yhl - yh;
8
9   #--------------- Transcription du code C ------------------------
10
11  s3 = IEEEdouble(-1.6666666666666665741480812812369549646974e-01);
12  s5 = IEEEdouble(8.3333333333333321768510160154619370587 17e-03);
13  s7 = IEEEdouble(-1.9841269841269841252631711547849135968136e-04);
14
15  yh2 IEEEdouble=  yh * yh;
16  ts  IEEEdouble=  yh2 * (s3 + yh2*(s5 + yh2*s7));
17  r   IEEEdouble=  yl + yh*ts;
18  s             =  yh + r;   #  Ici pas d'arrondi, c'est le Fast2Sum
```

Voici quelques commentaires. En Gappa, toutes les variables représentent des réels. Ici, `Yhl` est un réel arbitraire. La définition de `yh`, par contre, le contraint à être un double. On remarque que la définition de `yl` ne spécifie pas que `yl` est un double aussi. A-t-on vraiment besoin de cette information ? Laissons flotter le suspense pour le moment.

Les coefficients `s3` à `s7` sont donnés en décimal dans le code C, et le compilateur C est tenu de les arrondir au double le plus proche. Les décimaux ici exprimés sont très proches de tels doubles, donc faciles à arrondir. Il ne faut toutefois pas oublier d'insérer ces arrondis dans le script Gappa.

Comme déjà mentionné, la syntaxe «`IEEEdouble=`» insère des arrondis partout où le code C réalisera un arrondi.

Enfin, on escamote ici le fait que `s` soit représenté comme un double-double : cette information est parfaitement inutile pour calculer une erreur sur `s`.

Ensuite vient la description des différents objets mathématiques approchés par ce code.

```
20  My2 = My*My; Mts = My2 * (s3 + My2*(s5 + My2*s7)); PolySinY = My +
21  My*Mts;
```

Ici les seuls doubles sont `s3`, `s5` et `s7`, les autres variables sont des réels arbitraires. On a ainsi défini notre polynôme d'approximation `PolySinY` comme une fonction de la variable `My`, qui est l'argument réduit idéal dont notre double-double d'entrée n'est qu'une approximation.

Enfin, il faut énoncer le théorème que l'on demande à Gappa de prouver. Le listing suivant commence par définir un certain nombre de variables d'erreur, essentiellement pour augmenter la lisibilité.

```
26 epstotal = (s - SinY)/SinY;
27 epsapprox = (PolySinY - SinY)/SinY;
28 epsround = (s - PolySinY)/PolySinY;
29
30 epsargred = (Yhl - My)/My;
31
32 #------ Énoncé du théorème à prouver --------------------------
33 {
34     # Hypothèses
35     My in [ -6.14e-03, 6.14e-03]         # [- Pi/512, Pi/512]
36  /\ epsargred in [-2.53e-23, 2.53e-23]   # prouvé (encore) en papier
37  /\ epsapprox in [-2.26e-24, 2.26e-24]   # prouvé en dehors de ce fichier
38
39 ->
40    # but
41    epstotal in ?
42 }
```

La variable `epstotal` est l'erreur que l'on veut encadrer : l'erreur relative entre la valeur calculée `s` et le sinus de l'argument réduit idéal `SinY`.

C'est en fait la définition de `epsapprox` qui définit le sinus. Elle peut se lire comme «tout ce qu'on sait du sinus de `My`, c'est que c'est une valeur réelle qui sera à une distance relative plus petite que $2,26 \cdot 10^{-24}$ de `PolySinY`». C'est un peu contre-intuitif de définir le sinus mathématique à partir de son approximation polynomiale, mais cela permet d'introduire une valeur idéale du sinus alors que Gappa ne connait pas la fonction sinus. L'encadrement sur `epsapprox` est calculé (hors de ce script Gappa) par la norme infinie de la section précédente.

Un théorème Gappa est composé d'hypothèses et de buts qui sont des appartenances à des intervalles, avec les quantificateurs logiques usuels. Ici on a trois hypothèses et un seul but, dont l'intervalle n'est pas spécifié : on demande à Gappa de calculer le plus fin qu'il peut.

4.5.3 Aide Gappa, et Gappa t'aidera

Lorsque Gappa est invoqué sur le fichier ainsi construit, il répond qu'il ne sait pas construire une preuve du théorème.

Voyons comment Gappa procède. D'abord, il possède une définition de `epstotal` qui suggère le calcul

$$\texttt{epstotal} = \frac{\texttt{s} - \texttt{SinY}}{\texttt{SinY}}.$$

C'est une première manière de calculer un encadrement, qui du point de vue de l'arithmétique d'intervalle sera probablement catastrophique, puisqu'elle divise un numérateur par un dénominateur dont on sait qu'il va pouvoir s'approcher près de zéro[2].

[2]Un algorithme à base de fractions continues, dû à Kahan et Douglas et repris dans [112], permet de montrer que l'argument réduit idéal, et donc son sinus, ne peuvent pas s'approcher arbitrairement près de zéro. Il faudra injecter cette hypothèse à un certain point de la preuve, sans quoi notre `epstotal` ne sera même pas défini partout, et Gappa ne pourra en donner un encadrement. Nous escamotons cette difficulté dans cette présentation.

Toutefois, Gappa a une heuristique qui lui permet de trouver un chemin de calcul plus prometteur. Lorsque nous avons défini la variable `epsapprox` à la ligne 27, Gappa a pris note que `PolySinY` semblait être une approximation de `SinY`. Or on lui demande une erreur relative entre `s` et, justement, `SinY`. Gappa va donc essayer de passer par l'approximation intermédiaire `PolySinY`. Plus précisément, pour calculer `epstotal`, Gappa va essayer d'utiliser la formule

$$\texttt{epstotal} = \texttt{epsapprox} + \texttt{epsround} + \texttt{epsapprox} \cdot \texttt{epsround}.$$

Cette formule est une identité mathématique, comme le lecteur pourra le vérifier aisément à partir des définitions de `epstotal`, `epsapprox` et `epsround`. Elle présente l'avantage d'ajouter et de multiplier des nombres qui ressemblent à des erreurs relatives, donc (espère Gappa) qui seront petits. En termes d'intervalles, cette seconde formulation est donc bien plus prometteuse. Nous insistons sur le côté heuristique[3] de la chose : Gappa explore ainsi, pour chaque valeur qu'il doit encadrer, différentes manières de calculer l'encadrement.

Toutefois, ce n'est visiblement pas suffisant. On peut interroger Gappa en ajoutant le but `epsround in ?` pour vérifier qu'il ne sait pas encadrer `epsround`. Les raisons en sont celles qui ont été énumérées en 4.2, essayons à présent de les formaliser.

Nous allons commencer par définir pas à pas les différentes approximations qui séparent `s` de `PolySinY`. Cela donne les définitions Gappa suivantes, avec les définitions des erreurs relatives incrémentales :

```
28 S1 = yh + (yl + IEEEdouble(yh*ts));    # s sans son dernier arrondi
29 S2 = yh + (yl + yh*ts);                # encore un arrondi
30 S3 = (yh+yl) + (yh+yl)*ts;             # on a négligé yl
31
32 eps1 = (s-S1)/S1;
33 eps2 = (S1-S2)/S2;
34 eps3 = (S2-S3)/S3;
35 eps4 = (S3-PolySinY)/PolySinY;
```

Au passage, il faut prendre garde à toujours exprimer les erreurs comme des différences du moins précis au plus précis. Ajoutons à présent `eps1` à `eps4` aux buts à prouver. Encore raté ! Gappa n'arrive à en prouver aucun.

Il faut donc l'aider plus. Pour cela, Gappa offre un mécanisme d'*indices*. Un indice est une identité mathématique entre deux expressions `exp1 -> exp2` qui se lit «Pour obtenir un bon encadrement de `exp1`, essaye donc l'expression équivalente `exp2`». Gappa vérifie d'ailleurs que les deux expressions sont mathématiquement équivalentes.

Considérons par exemple `eps4`, qui est la différence entre `S3` et `PolySinY`. Ce sont deux expressions de polynômes, dans lesquelles il n'y a plus d'arrondi, et les coefficients sont identiques. Sur le fond, cette différence doit donc se ramener à la différence entre `Yhl=yh+yl`, utilisé dans `S3`, et `My`, utilisée dans `PolySinY`. Or nous avons une mesure de cette différence : c'est très précisément `epsargred`. L'exercice consiste donc à ramener `eps4` à une expression faisant intervenir `epsargred` et dont le calcul par arithmétique d'intervalle donne un encadrement fin. C'est ce que fait le listing suivant. On part de `eps4 -> (S3-PolySinY)/PolySinY`, qui est juste la définition de `eps4`, et on la réécrit progressivement jusqu'à une expression équivalente, mais plus prometteuse du point de vue de son évaluation en arithmétique d'intervalles.

[3] Mentionnons aussi que si nous avions écrit `epsapprox = PolySinY/SinY -1;` qui est une expression équivalente, Gappa n'aurait pas essayé d'utiliser une variable comme une approximation de l'autre...

```
42 eps4 ->
43 #  (S3-PolySinY)/PolySinY;
44 #   S3/PolySinY - 1;
45 #   ((yh+yl) + (yh+yl)*ts) / (My + My*Mts)  - 1;
46 #   ((yh+yl)/My) * (1+ts)/(1+Mts)  - 1;
47 #   (epsargred+1) * (1+ts)/(1+Mts)  - 1;
48 #    epsargred * (1+ts)/(1+Mts)   +  1 * (1+ts)/(1+Mts)  - 1;
49 #    epsargred * (1+ts)/(1+Mts)   +  (ts-Mts)/(1+Mts);
50    epsargred * (1+ts)/(1+Mts)   +  Mts*((ts-Mts)/Mts) / (1+Mts);
```

D'après nos ordres de grandeur du 4.2, cette expression fera l'affaire. En effet, `Mts` comme `ts` vont être petits devant 1. Le premier terme sera donc proche de `epsargred`, quant au second, nous l'avons exprimé comme l'erreur relative de `ts` par rapport à `Mts`, qui sera de l'ordre de 2^{-52}, multipliée par `Mts` qui est de l'ordre de 2^{-14}. On aura donc une somme de deux termes qui restent très petits, et la somme, même calculée en arithmétique d'intervalle, sera petite.

Toutefois, Gappa échoue encore. Qu'est-ce qui coince ? Pour le savoir, ajoutons à ses buts des encadrements pour `ts`, `Mts` et pour `(ts-Mts)/Mts`. On obtient enfin un succès :
`Mts in [-2^(-17.2801), 0]`

Toutefois, Gappa ne trouve pas d'encadrement pour `ts`. Pour l'aider, nous allons utiliser la même approche : écrire les différentes couches approximation intermédiaires entre `ts` et `Mts`, définir les erreurs relatives correspondantes, puis donner si nécessaire à Gappa des indices ramenant ces erreurs à ce qu'il sait déjà encadrer. Nous ne détaillerons pas la suite de ce processus, car il faudra au passage apprendre à Gappa à passer de `My` à `yh2` à travers les trois approximations énumérées page 64. Le lecteur intéressé trouvera le script Gappa complet dans la distribution de CRLibm.

Levons toutefois le suspense sur une question posée il y a trois pages de cela : l'information selon laquelle `yl` est un double dans notre code est elle nécessaire à la preuve ? La réponse est oui : l'un de nos indices consistera à se ramener à l'erreur relative de l'addition flottante `yl+ts*yh` de la troisième ligne de notre code C. Or Gappa a dans sa bibliothèque des théorèmes sur l'addition de deux flottants, mais n'en a pas sur l'arrondi de l'addition d'un réel arbitraire avec un flottant ! Pour que Gappa puisse exploiter notre indice, il devra savoir que `yl` aussi est un flottant.

4.5.4 Une page de publicité

Cet exemple illustre qu'une grande force de Gappa, et des outils de preuve formelle en général, est de ne rien laisser au hasard. Lorsqu'on a fini la preuve, on sait qu'on n'a rien laissé de côté. C'est particulièrement précieux pour les valeurs sous-normales.

L'autre grande force de Gappa est de maintenir la confiance dans la preuve au long de son développement. Si un indice est inutile, il est tout simplement ignoré. Si un indice est faux (si ce n'est pas une identité mathématique) un warning est produit. Il faut porter un soin particulier à la transcription du code et à l'expression du théorème à prouver, mais une fois ceci fait correctement il est très difficile d'obtenir une preuve fausse.

Enfin, la troisième grande force de l'outil est son interactivité : on peut à tout instant ajouter des buts au théorème à prouver, et observer ainsi ce que Gappa sait encadrer et ce qu'il ne sait pas.

Avec l'aide de Gappa, on trouve au final une borne de l'erreur relative totale d'approximation (`epstotal`) de $2^{-67,24}$. La preuve Coq compte plus de 7000 lignes.

4.6 Limites et perspectives

Ce chapitre n'a pas beaucoup parlé de matériel. La raison principale est que, les FPGA étant reconfigurables et les précisions recherchées petites, il a jusque là toujours été possible de vérifier nos opérateurs par test exhaustif, à pleine vitesse sur un vrai circuit FPGA. Cela deviendra impossible si la double-précision se généralise. Nous avons déjà des générateurs d'opérateurs, il faudra alors qu'ils génèrent également une preuve de l'opérateur. Il ne semble pas y avoir d'obstacle à utiliser Gappa pour cela aussi.

Cela dit, Gappa ne sait pas tout faire. On aimerait bien mécaniser la preuve du code utilisé pour tester si l'arrondi est possible dans CRLibm [1], ou les 60 pages de preuves de l'arithmétique triple-double [88], ou la preuve de l'arrondi correct des pires cas de Lauter [29]. Ce sont des exemples de preuves qui n'entrent pas dans la catégorie «prouver qu'une valeur appartient à un intervalle». Les travaux de Sylvie Boldo [9] ont montré qu'on peut réaliser ce type de preuves directement en Coq, mais nos algorithmes sont beaucoup plus gros que ceux traités par Boldo. Pour de telles preuves, il faudrait trouver un langage de plus haut niveau que Coq, mais capable de produire du Coq. Les difficultés récurrentes sont ici la combinatoire des situations : dans une preuve impliquant de nombreux résultats flottants, on aura par exemple pour chacun de ces flottants à vérifier ce qui se passe en cas d'arrondi pair, ou distinguer le cas des puissances de deux, parce que les règles d'arrondi sont légèrement différentes dans ces cas. Ces cas forment une combinatoire très lourde, l'assistant de preuve vérifiera qu'aucun cas n'a été oublié, mais pour le moment l'énumération des cas est à notre charge. Elle gagnerait à être automatisée.

4.7 Pour conclure

L'arrivée de Gappa dans CRLibm est un exemple unique à ma connaissance de l'utilisation massive d'outils de preuve formelle par des gens n'appartenant pas à la communauté de la preuve formelle. Cela est rendu possible par un outil qui fait l'interface entre les deux communautés, celle du logiciel flottant et celle de la preuve formelle.

On peut faire la comparaison avec la «démocratisation» de la programmation, dans les années 50 : l'arrivée des langages de haut niveau et des compilateurs a permis l'utilisation des ordinateurs par des gens qui n'étaient pas spécialistes de l'architecture des ordinateurs. Avant cette révolution, on programmait en assembleur, et le physicien qui voulait programmer devait devenir un spécialiste d'un autre domaine. L'assembleur n'a pas disparu, bien au contraire, mais il s'est caché : De nos jours, l'assembleur écrit à la main ne représente plus qu'une part infime des programmes.

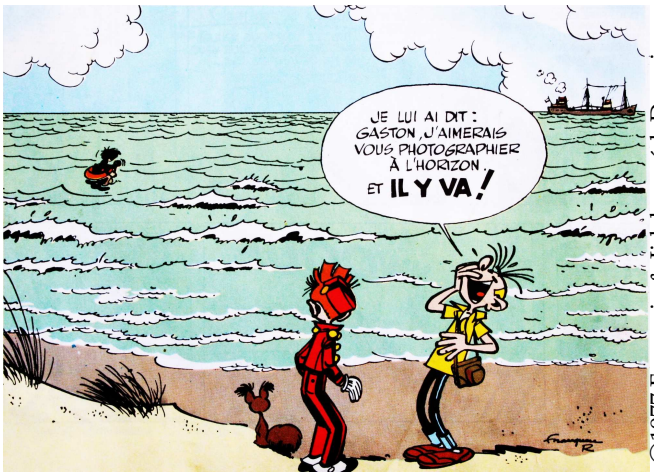
Un assistant de preuve tel que Coq est à mon avis équivalent à un assembleur, et Gappa à un langage de plus haut niveau, qui au final produit cet assembleur après un complexe processus d'optimisation. J'espère que cette approche aura le succès qu'elle a eu en programmation.

Je dois ajouter que, pour un utilisateur qui n'a pas vocation à devenir un spécialiste de preuve formelle, se frotter à ce monde est très enrichissant, notamment parce que cela enseigne à formaliser ses intuitions. Mais, de même que les concepts qui sous-tendent les

langages de haut niveau actuels (orientation objet, programmation fonctionnelle, etc.) n'ont pas d'équivalent direct dans les assembleurs et les architectures de processeur, il est encore plus enrichissant de travailler à développer des concepts de preuve de haut niveau qui sont compatibles avec la preuve formelle par «compilation». C'est la réussite de Gappa.

CHAPITRE 5

Conclusions et perspectives



©1977 Franquin & Jidehem, éd. Dupuis

Dans ces travaux, nous avons tourné quelques pages, comme celle des méthodes à base de table au chapitre 2. Des pages sont en train de se tourner, comme celle de la normalisation des fonctions élémentaires au chapitre 3. Nous avons découvert quelques pages encore presque vierges, comme celles de l'utilisation des FPGA comme accélérateurs flottants au chapitre 2, ou l'utilisation au chapitre 4 d'assistants de preuve enfin accessibles aux masses. À quoi ressemblera le livre qui s'écrit ?

Je crois que la tendance qui se dégage, c'est qu'il est temps que la notion de bibliothèque s'efface, lorsque c'est possible, devant celle de générateur de code ou de circuit. À petite échelle, on sait déjà construire de tels générateurs. L'étape suivante sera que cette évolution percole dans les langages de programmation. Actuellement, on écrit

```
pow(cos(omega*t+pi), 2)
```

et on peut prouver a postériori que l'erreur sera plus petite qu'une certaine borne, parce que les opérateurs de bibliothèque pour `pow`, `cos`, `*`, `+` et `pi` offrent une certaine qualité. Mais à terme, ne préférerait-on pas écrire

```
eval (cos(omega*t+pi))^2 +/- 0.001 for t in[0..10000]
```

et avoir un compilateur[1] qui nous produirait un circuit ou un programme taillé au plus juste

[1] Le mot de compilateur est ici utilisé à contrecœur, puisque son étymologie renvoie aux temps anciens où le rôle principal d'un compilateur était justement d'assembler des morceaux de code pris sur une étagère, sans plus de traitement qu'il n'en faut pour faire une compilation des tubes de Mozart. Depuis, les compilateurs sont devenus des compilateurs optimiseurs, c'est même dans l'optimisation que se trouve l'essentiel de la recherche contemporaine dans ce domaine [108]. C'est bien ce processus d'optimisation de plus en plus poussée que j'entends par le terme de compilation.

pour évaluer cette expression avec la précision demandée ? C'est un changement plus profond qu'il n'y paraît puisque le `cos`, opérateur de bibliothèque dans la première expression, approximation mensongère du cosinus, a été remplacé dans la seconde expression par la vraie fonction cosinus. Ce serait un grand progrès de sincérité de la part des langages de programmation.

Certes, la communauté du calcul formel offre des langages sincères : le `cos` de Maple représente un vrai cosinus, avec lequel on peut faire du calcul formel, contrairement à celui du C. Mais elle n'offre pas encore le compilateur qui va avec quand il s'agit de l'évaluer numériquement. Pire, la documentation fournie ne mentionne rien au sujet de la précision de cette évaluation.

De notre côté, nous avons appris à manipuler de plus en plus automatiquement les fonctions et les erreurs, et nous progressons pour savoir synthéthiser du code ou du circuit pour des classes de plus en plus vastes de calculs. La plupart de mes contributions peuvent ainsi être considérées comme de petits progrès vers un compilateur pour un langage sincère. Pour des expressions comme celle que j'ai prise en exemple, nous croyons que ce but est accessible, et c'est l'objet du projet ANR EVAFlo auquel je participe. Pour des codes plus complexes, heureusement, l'ancienne et la future manière de spécifier un calcul peuvent cohabiter.

Considérant mon bagage de chercheur très applicatif, cette direction ne m'intéresse pas uniquement parce qu'elle fera progresser le niveau d'abstraction d'un programme. La seconde expression est aussi celle qui permettra l'optimisation la plus fine. Qui a besoin d'une réduction d'argument trigonométrique qui marche jusqu'à 2^{1024} [113] si `t` reste plus petit que 1000 ? Pourquoi utiliser un cosinus de bibliothèque dont la réduction d'argument doit multiplier par l'irrationnel $1/\pi$, alors que `omega` est sans doute lui-même défini comme égal à $2\pi F$ quelques lignes plus haut dans le code ? Pourquoi utiliser 53 bits de mantisse alors qu'on veut un résultat précis à 10-11 bits ? Etc. Cette problématique de l'optimisation «au plus juste» n'est pas l'aspect le moins riche de cette nouvelle approche. Elle viendra compléter des optimisations d'ordre plus syntaxique qu'on commence juste à explorer, comme les fonctions optimisées pour les boucles [22] ou le partage de réduction d'argument pour les paires sinus/cosinus [103].

Nos travaux n'ont porté que sur des fonctions d'une variable. Il faudra aussi s'intéresser aux fonctions de plusieurs variables, pour lesquelles tout reste à faire.

Pour du code plus complexe, même spécifier le résultat que l'on veut obtenir restera longtemps un problème ouvert, et un langage sincère universel se trouve, au mieux, à l'horizon.

Comme Gaston, je suis heureux de croire que mes petits coups de rame me rapprochent de cet horizon.

Bibliographie

[1] CR-Libm, a library of correctly rounded elementary functions in double-precision. `http://lipforge.ens-lyon.fr/www/crlibm/`.

[2] Interval arithmetic in high performance technical computing. Rapport de recherche, Sun Microsystems, Sept. 2002.

[3] M. Allie and R. Lyons. A root of less evil. *Signal Processing Magazine*, 22(2) :93–96, Mar. 2005.

[4] C. S. Anderson, S. Story, and N. Astafiev. Accurate math functions on the Intel IA-32 architecture : A performance-driven design. In *7th Conference on Real Numbers and Computers*, pages 93–105, 2006.

[5] ANSI/IEEE. *Standard 754-1985 for Binary Floating-Point Arithmetic (also IEC 60559)*. 1985.

[6] M. Arnold, T. Bailey, J. Cowles, and M. Winkel. Applying features of IEEE 754 to sign/logarithm arithmetic. *IEEE Transactions on Computers*, 41(8) :1040–1050, Août 1992.

[7] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 657–666. Springer, 2002.

[8] J.-L. Beuchat and A. Tisserand. Small multiplier-based multiplication and division operators for Virtex-II devices. In *Field-Programmable Logic and Applications*, volume 2438 of *LNCS*. Springer, Sept. 2002.

[9] S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. Thése de doctorat, École Normale Supérieure de Lyon, Nov. 2004.

[10] S. Boldo and M. Daumas. A mechanically validated technique for extending the available precision. In *35th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, 2001. IEEE Computer Society Press.

[11] S. Boldo and M. Daumas. A simple test qualifying the accuracy of horner's rule for polynomials. *Numerical Algorithms*, 37(1-4) :45–60, 2004.

[12] G. Brebner. Field-programmable logic : Catalyst for new computing paradigms. In *International Workshop on Field Programmable Logic and Applications*, Tallin, Estonia, 1998.

[13] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1) :57–70, 1978.

[14] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞- approximations. In *18th Symposium on Computer Arithmetic*, pages 169–176. IEEE Computer Society Press, 2007.

[15] N. Brisebarre and G. Hanrot. Floating-point L^2- approximations to functions. In *18th Symposium on Computer Arithmetic*, pages 177–184. IEEE Computer Society Press, 2007.

[16] N. Brisebarre, F. Hennecart, J.-M. Muller, A. Tisserand, and S. Torres. Meplib : Machine efficient polynomial library. LGPL software, 2004.

[17] N. Brisebarre, J.-M. Muller, and A. Tisserand. Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2) :236–256, Juin 2006.

[18] N. Brisebarre, J.-M. Muller, A. Tisserand, and S. Torres. Hardware operators for function evaluation using sparse-coefficient polynomials. *Electronic Letters*, 42 :1441–1442, 2006.

[19] K. Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, Mai 1994.

[20] S. Chevillard and C. Q. Lauter. Certified infinite norm using interval arithmetic. In *SCAN 2006 - 12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Duisburg, Germany, 2006.

[21] J. N. Coleman and E. I. Chester. Arithmetic on the European logarithmic microprocessor. *IEEE Transactions on Computers*, 49(7) :702–715, Juil. 2000.

[22] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[23] M. Cosnard, A. Guyot, B. Hochet, J. M. Muller, H. Ouaouicha, P. Paul, and E. Zysmann. The FELIN arithmetic coprocessor chip. In *8th Symposium on Computer Arithmetic*, pages 107–112, 1987.

[24] D. Das Sarma and D. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. McAllister, editors, *12th Symposium on Computer Arithmetic*, pages 17–28, Bath, UK, 1995. IEEE.

[25] M. Daumas. Expansions : lightweight multiple precison arithmetic. In *Architecture and Arithmetic Support for Multimedia*, Dagstuhl, Germany, 1998.

[26] M. Daumas and C. Finot. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science*, 5(6) :323–338, 1999.

[27] F. de Dinechin. The price of routing in FPGAs. *Journal of Universal Computer Science*, 6(2) :227–239, Fév. 2000.

[28] F. de Dinechin and D. Defour. Software carry-save : A case study for instruction-level parallelism. In *Seventh International Conference on Parallel Computing Technologies*, Sept. 2003.

[29] F. de Dinechin, D. Defour, and C. Q. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Rapport de recherche 2004-10, LIP, École Normale Supérieure de Lyon, Mar. 2004.

[30] F. de Dinechin and J. Detrey. Multipartite tables in JBits for the evaluation of functions on FPGAs. In *IEEE Reconfigurable Architecture Workshop, International Parallel and Distributed Symposium*, Avr. 2002.

[31] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th Symposium on Computer Arithmetic*, pages 288–295. IEEE Computer Society Press, Juin 2005.

[32] F. de Dinechin, C. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, 2006.

[33] F. de Dinechin, C. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 41 :85–102, 2007.

[34] F. de Dinechin and V. Lefèvre. Constant multipliers for FPGAs. In *Parallel and Distributed Processing Techniques and Applications*, pages 167–173, 2000.

[35] F. de Dinechin and S. Maidanov. Software techniques for perfect elementary functions in floating-point interval arithmetic. In *Real Numbers and Computers*, Juil. 2006.

[36] F. de Dinechin, E. McIntosh, and F. Schmidt. Massive tracking on heterogeneous platforms. In *9th International Computational Accelerator Physics Conference (ICAP)*, Oct. 2006.

[37] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3) :319–330, 2005.

[38] D. Defour. Cache-optimised methods for the evaluation of elementary functions. Rapport de recherche 2002-38, LIP, École Normale Supérieure de Lyon, 2002.

[39] D. Defour. *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. Thése de doctorat, École Normale Supérieure de Lyon, Lyon, France, Sept. 2003.

[40] D. Defour. Collapsing dependent floating point operations. Rapport de recherche, DALI Research Team, LP2A, University of Perpignan, France, Déc. 2004.

[41] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, 2002.

[42] D. Defour, F. de Dinechin, and J.-M. Muller. A new scheme for table-based evaluation of functions. In *36th Asilomar Conference on Signals, Systems, and Computers*, Nov. 2002.

[43] D. Defour, G. Hanrot, V. Lefèvre, J.-M. Muller, N. Revol, and P. Zimmermann. Proposal for a standardization of mathematical function implementations in floating-point arithmetic. *Numerical algorithms*, 37(1-4) :367–375, Jan. 2004.

[44] A. DeHon. DPGA-coupled microprocessors : Commodity ICs for the early 21st century. In *FPGAs for Custom Computing Machines*. IEEE, 1994.

[45] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3) :224–242, 1971.

[46] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Field-Programmable Gate Arrays*, pages 75–85. ACM, 2005.

[47] J. Detrey. *Arithmétiques réelles sur FPGA ; Virgule fixe, virgule flottante et système logarithmique*. Thése de doctorat, École Normale Supérieure de Lyon, Lyon, France, Sept. 2007.

[48] J. Detrey and F. de Dinechin. Second order function approximation using a single multiplication on FPGAs. In *14th Intl Conference on Field-Programmable Logic and Applications (LNCS 3203)*, pages 221–230. Springer, Août 2004.

[49] J. Detrey and F. de Dinechin. Outils pour une comparaison sans a priori entre arithmétique logarithmique et arithmétique flottante. *Technique et science informatiques*, 24(6) :625–643, 2005.

[50] J. Detrey and F. de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In *39th Asilomar Conference on Signals, Systems & Computers*. IEEE, 2005.

[51] J. Detrey and F. de Dinechin. A parameterized floating-point exponential function for FPGAs. In *Field-Programmable Technology*. IEEE, Déc. 2005.

[52] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *Application-specific Systems, Architectures and Processors*, pages 328–333. IEEE, 2005.

[53] J. Detrey and F. de Dinechin. Opérateurs trigonométriques en virgule flottante sur FPGA. In *RenPar'17, SympA'2006, CFSE'5 et JC'2006*, pages 96–105, Perpignan, France, Oct. 2006.

[54] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. In *Microprocessors and Microsystems*. Elsevier, 2007. À paraître.

[55] J. Detrey and F. de Dinechin. A tool for unbiased comparison between logarithmic and floating-point arithmetic. *Journal of VLSI Signal Processing*, 2007. À paraître.

[56] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, Juin 2007.

[57] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. In *Field-Programmable Gate Arrays*, pages 50–55. ACM, 2002.

[58] C. Doss and R. L. Riley, Jr. FPGA-based implementation of a robust IEEE-754 exponential unit. In *Field-Programmable Custom Computing Machines*, pages 229–238. IEEE, 2004.

[59] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Field-Programmable Gate Arrays*, pages 86–95. ACM, 2005.

[60] J. Duprat and J.-M. Muller. Ecrire les nombres autrement pour calculer plus vite. *Technique et Science Informatique*, 10(3), 1991.

[61] M. Ercegovac. Radix-16 evaluation of certain elementary functions. *IEEE Transactions on Computers*, C-22(6) :561–566, Juin 1973.

[62] M. D. Ercegovac and T. Lang. *Division and Square Root : Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994.

[63] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[64] A. G. Ershov and T. P. Kashevarova. Interval mathematical library based on Chebyshev and Taylor series expansion. *Reliable Computing*, 11(5) :359–367, 2005.

[65] P. Farmwald. High-bandwidth evaluation of elementary functions. In *5th Symposium on Computer Arithmetic*, pages 139–142, 1981.

[66] M. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8) :154–161, Fév. 1970.

[67] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Interscience, 2001.

[68] S. Gal. Computing elementary functions : A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.

[69] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1) :26–45, Mar. 1991.

[70] GMP, the GNU multi-precision library. http ://gmplib.org/.

[71] R. E. Goldschmidt. Applications of division by convergence. Thése de Master, Electrical Engineering, Massachusetts Institute of Technology, Juin 1964.

[72] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Reconfigurable Architecture Workshop, Intl. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.

[73] J. Harrison. Floating point verification in HOL light : The exponential function. In *Algebraic Methodology and Software Technology*, pages 246–260, 1997.

[74] J. Harrison. Formal verification of floating point trigonometric functions. In *Formal Methods in Computer-Aided Design : Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[75] J. Harrison, T. Kubaska, S. Story, and P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.

[76] H. Hassler and N. Takagi. Function evaluation by table look-up and addition. In *12th Symposium on Computer Arithmetic*, pages 10–16, Bath, UK, 1995. IEEE.

[77] K. S. Hemmert and K. D. Underwood. An analysis of the double-precision floating-point FFT on FPGAs. In *13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, Avr. 2005.

[78] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In *15th Symposium on Computer Arithmetic*, pages 155–162. IEEE, Juin 2001.

[79] W. Hofschuster and W. Krämer. FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format. Rapport de recherche Nr. 98/7, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1998.

[80] G. Huet, G. Kahn, and C. Paulin-Mohring. *The Coq proof assistant : a tutorial : version 8.0*, 2004.

[81] C. Iordache and D. W. Matula. Analysis of reciprocal and square root reciprocal instructions in the AMD K6-2 implementation of 3DNow ! *Electronic Notes in Theoretical Computer Science*, 24, 1999.

[82] ISO/IEC. *International Standard ISO/IEC 9899 :1999(E). Programming languages – C*. 1999.

[83] ISO/IEC. *International Standard ISO/IEC1539-1 :2004. Programming languages – Fortran – Part 1 : Base language*. 2004.

[84] D. Knuth. *The Art of Computer Programming, vol.2 : Seminumerical Algorithms*. Addison Wesley, 3rd edition, 1997.

[85] I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, 1993.

[86] W. Krämer. Inverse standard functions for real and complex point and interval arguments with dynamic accuracy. In *Scientific Computation with automatic result verification*, pages 185–211. Springer-Verlag, 1988.

[87] C. Q. Lauter. A correctly rounded implementation of the exponential function on the Intel Itanium architecture. Rapport de recherche 2003-54, LIP, École Normale Supérieure de Lyon, Nov. 2003. Available at ftp ://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2003/RR2003-54.ps.gz.

[88] C. Q. Lauter. Basic building blocks for a triple-double intermediate format. Rapport de recherche 2005-38, LIP, École Normale Supérieure de Lyon, Sept. 2005.

[89] C. Q. Lauter. Exact and mid-point rounding cases of power(x,y). Rapport de recherche 2006-46, Laboratoire de l'Informatique du Parallélisme, 2006.

[90] B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.

[91] B. Lee and N. Burgess. A dual-path logarithmic number system addition/subtraction scheme for FPGA. In *Field-Programmable Logic and Applications*, Lisbon, Sept. 2003.

[92] D. Lee, A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12) :1520–1531, Déc. 2005.

[93] D.-U. Lee, W. Luk, J. Villasenor, and P. Cheung. Hierarchical segmentation schemes for function evaluation. In *IEEE Conference on Field-Programmable Technology*, Tokyo, dec 2003.

[94] D.-U. Lee, J. Villasenor, W. Luk, and P. Leong. A hardware gaussian noise generator using the box-muller method and its error analysis. *IEEE Transactions on Computers*, 55(6), Juin 2006.

[95] V. Lefèvre. Multiplication by an integer constant. Rapport de recherche RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[96] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. Thése de doctorat, École Normale Supérieure de Lyon, Lyon, France, Jan. 2000.

[97] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. http ://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm, 2004.

[98] D. Lewis. Interleaved memory function interpolators with application to an accurate LNS arithmetic unit. *IEEE Transactions on Computers*, 43(8) :974–982, Août 1994.

[99] D. M. Lewis. An architecture for addition and subtraction of long word length numbers in the logarithmic number system. *IEEE Transactions on Computers*, 39(11), Nov. 1990.

[100] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Rapport de recherche, Hewlett-Packard company, april 2001.

[101] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *FPGAs for Custom Computing Machines*. IEEE, 2002.

[102] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[103] P. Markstein. Accelerating sine and cosine evaluation with compiler assistance. In J.-C. Bajard and M. Schulte, editors, *16th Symposium on Computer Arithmetic*, pages 137–140. IEEE Computer Society, Juin 2003.

[104] D. Matula. Improved table lookup algorithms for postscaled division. In *15th Symposium on Computer Arithmetic*, pages 101–108, Vail, Colorado, Juin 2001. IEEE.

[105] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. Thése de doctorat, École Normale Supérieure de Lyon, Lyon, France, 2006.

[106] R. E. Moore. *Interval analysis*. Prentice Hall, 1966.

[107] MPFR. `http://www.mpfr.org/`.

[108] S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[109] J.-M. Muller. Algorithmes de division pour microprocesseurs : illustration à l'aide du "bug" du pentium. *Technique et Science Informatiques*, 14(8), Oct. 1995.

[110] J.-M. Muller. A few results on table-based methods. *Reliable Computing*, 5(3) :279–288, 1999.

[111] J.-M. Muller. On the definition of ulp(x). Research Report RR2005-09, Laboratoire de l'Informatique du Parallélisme (LIP), February 2005.

[112] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, 2 edition, 2006.

[113] K. C. Ng. Argument reduction for huge arguments : good to the last bit. Technical report, SunPro, Mountain View, CA, USA, Juil. 1992.

[114] S. F. Oberman. Floating point division and square root algorithms and implementation in the amd-k7(tm) microprocessor. In *14th Symposium on Computer Arithmetic*. IEEE, Avr. 1999.

[115] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2) :154–161, 1997.

[116] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product.

[117] F. Ortiz, J. Humphrey, J. Durbano, and D. Prather. A study on the design of floating-point functions in FPGAs. In *Field Programmable Logic and Applications*, volume 2778 of *LNCS*, pages 1131–1135. Springer, Sept. 2003.

[118] G. Paul and M. W. Wilson. Should the elementary functions be incorporated into computer instruction sets ? *ACM Transactions on Mathematical Software*, 2(2) :132–142, Juin 1976.

[119] J. A. Piñeiro, J. D. Bruguera, and J.-M. Muller. Faithful powering computation using table look-up and a fused accumulation tree. In *15th Symposium on Computer Arithmetic*, pages 40–47, Vail, Colorado, Juin 2001. IEEE.

[120] D. Priest. Fast table-driven algorithms for interval elementary functions. In *13th Symposium on Computer Arithmetic*, pages 168–174. IEEE, 1997.

[121] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. In *Workshop on Validated Computing*, pages 155–161. SIAM, 2002.

[122] G. Revy. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Rapport de recherche, Laboratoire de l'Informatique du Parallélisme - ENS Lyon, 2006.

[123] S. M. Rump. Rigorous and portable standard functions. *BIT Numerical Mathematics*, 41(3), 2001.

[124] M. Schulte and J. Stine. Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8) :842–847, Août 1999.

[125] N. A. Sherwani. *Algorithms for VLSI physical design automation*. Kluwer Academic, 1993.

[126] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.

[127] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machine. In *FPGAs for Custom Computing Machines*, pages 155–162. IEEE, 1995.

[128] G. Snider, P. Kuekes, W. B. Culbertson, R. J. Carter, A. S. Berger, and R. Amerson. The Teramac configurable computer engine. In *International Workshop on Field Programmable Logic and Applications*, pages 44–53. LNCS 975, 1995.

[129] D. Stehlé, V. Lefèvre, and P. Zimmermann. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers*, 54(3) :340–346, 2005.

[130] D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst cases and lattice reduction. In *16th Symposium on Computer Arithmetic*, pages 142–147. IEEE Computer Society Press, 2003.

[131] J. Stine and M. Schulte. The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, 21(2) :167–177, 1999.

[132] S. Story and P. Tang. New algorithms for improved transcendental functions on IA-64. In *14th Symposium on Computer Arithmetic*, pages 4—11. IEEE, Avr. 1999.

[133] Sun Microsystems. *VIS^{TM}Instruction Set User Manual*, Mai 2001. Part Number 805-1394-03.

[134] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. R. Role. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid-State Circuits*, 19(4) :497–506, Août 1984.

[135] I. Sutherland and R. Sproull. Logical Effort : Designing for speed on the back of an envelope. In *Advanced Research in VLSI*, pages 1–16, Santa Cruz, 1991.

[136] I. E. Sutherland, R. F. Sproull, and D. Harris. *Logical Effort : Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, 1st edition (1999). ISBN : 1558605576.

[137] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2) :144–157, Juin 1989.

[138] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4) :378 – 400, Déc. 1990.

[139] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In *10th Symposium on Computer Arithmetic*. IEEE, Juin 1991.

[140] P. T. P. Tang. Table-driven implementation of the expm1 function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 18(2) :211–222, Juin 1992.

[141] M. Taufer, D. Anderson, P. Cicotti, and C. Brooks. Homogeneous redundancy : a technique to ensure untegrity of molecular simulation results using public computing. In *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[142] F. J. Taylor, R. Gill, J. Joseph, and J. Radke. A 20 bit logarithmic number system processor. *IEEE Transactions on Computers*, 37(2), Fév. 1988.

[143] J. W. Thomas, J. P. Okada, P. Markstein, and R.-C. Li. The *libm* library and floating-point arithmetic in HP-UX for Itanium-based systems. Technical report, Hewlett-Packard Company, Palo Alto, CA, USA, Déc. 2004.

[144] K. Underwood. FPGAs vs. CPUs : Trends in peak floating-point performance. In *ACM/SIGDA Field-Programmable Gate Arrays*. ACM Press, 2004.

[145] J. Vuillemin. On computing power. In *Programming Languages and System Architectures, LNCS 782*, pages 69–86, Zürich, Switzerland, 1994.

[146] A. Walters and P. Athanas. A scalable FIR filter using 32-bit floating-point complex arithmetic on a configurable computing machine. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 1998.

[147] M. Wirthlin. Constant coefficient multiplication using look-up tables. *Journal of VLSI Signal Processing*, 36(1) :7–15, 2004.

[148] W. F. Wong and E. Goto. Fast evaluation of the elementary functions in double precision. In *Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 349–358, 1994.

[149] W. F. Wong and E. Goto. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Transactions on Computers*, 43(3) :278–294, Mar. 1994.

[150] W. F. Wong and E. Goto. Fast evaluation of the elementary functions in single precision. *IEEE Transactions on Computers*, 44(3) :453–457, Mar. 1995.

[151] C. Wrathall and T. C. Chen. Convergence guarantee and improvements for a hardware exponential and logarithm evaluation scheme. In *4th Symposium on Computer Arithmetic*, pages 175–182, 1978.

[152] Xilinx Corporation. *Virtex-II Platform FPGA Handbook*, Déc. 2000.

[153] Xilinx Corporation. *Sine/Cosine Lookup Table V4.2*, Nov. 2002.

[154] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*. Thése de doctorat, Swiss Federal Institute of Technology, Zurich, 1997.

[155] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3) :410–423, Sept. 1991.

Index

additionneur, 18
    rapide, 18
algorithme de Ziv, 41
approximation polynomiale, 4
approximation rationnelle, 5
arithmétique d'intervalle, 39, 57
arithmétique logarithmique, 14
arrondi
    au plus près, 9
    au plus près pair, 39
    correct, 39
    fidèle, 9
ASIC, 15
ATA, 29

bipartite, 20
    symétrique, 21

DDE, 48, 51
digital signal processing, 12
dilemme du fabricant de tables, 40
division
    itérations quadratiques, 13
    récurrence sur les chiffres, 13
    SRT, 13
double précision, 39
double étendue, 45
double-double, 48
double-double-étendue, 48, 51
DSP, 12
dyadique, 6

erreur, 62
    absolue, 3
    d'approximation, 7
    d'arrondi, 7
    relative, 3
Estrin, 6

Fast2Sum, 50
field programmable gate array, 15
FMA, 6
FPGA, 15
frcpa, 13
fréquence, 17
fused multiply and add, 6

Gappa, 61, 67
générateurs d'opérateurs, 11

Hassler et Takagi, 27
Horner, 6

IA64, 13, 40
IEEE-754, 39
    révision, 56
Itanium, 5, 6, 13, 40

Knuth et Eve, 7

latence, 17
Lauter, 51
LNS, 14
logarithme
    en virgule flottante
        en matériel, 31
look-up table, 18
LUT, 18

multipartite, 22, 23
mémoires, 19

Paterson et Stockmeyer, 7
pipeline, 6
pires cas pour les fonctions élémentaires, 42
porte logique, 17
propriété d'inclusion, 57
précision double étendue, 45

réduction d'argument, 5
    par intervalle, 5
    par table, 5

schémas d'évaluation, 5
simple précision, 39
SRT, 13
SSE2, 45
STAM, 21
STBM, 21
surface, 15

tables, 19
traitement du signal, 12
triple-double, 48

ulp, 8

VHDL, 17
virgule flottante
    IEEE-754, 39
    sur FPGA, 35

Ziv, 42

Publications annexées à ce document

[37] Florent de Dinechin, Arnaud Tisserand. **Multipartite table methods**. *IEEE Transactions on Computers*, 54(3) :319-330, 2005.
Cet article fait le tour des méthodes d'approximation matérielles à base de tables et d'additions.

[52] Jérémie Detrey, Florent de Dinechin. **Table-based polynomials for fast hardware function evaluation**. In *16th Intl Conference on Application-specific Systems, Architectures and Processors*. IEEE Computer Society Press, July 2005.
Cet article introduit les méthodes à base de tables d'ordre supérieur pour l'évaluation de fonction en matériel. Le chapitre 2 de la thèse de J. Detrey en est une version étendue.

[55] Jérémie Detrey, Florent de Dinechin. **A tool for unbiased comparison between logarithmic and floating-point arithmetic**. *Journal of VLSI Signal Processing*, 2007. À paraître.
Cet article présente deux bibliothèques concurrentes pour l'arithmétique virgule flottante et le système logarithmique. Il montre que c'est la meilleure manière de décider quelle arithmétique est le mieux adaptée à une application donnée, avec ses contraintes de performance et de précision.

[54] Jérémie Detrey, Florent de Dinechin. **Parameterized floating-point logarithm and exponential functions for FPGAs**. *J. Microprocessors and Microsystems*. Elsevier, 2007. À paraître.
Cet article présente des algorithme matériels pour l'exponentielle et le logarithme en virgule flottante. Une approche originale à base de table offre une latence très petite pour une surface acceptable jusqu'à la simple précision.

[53] Jérémie Detrey, Florent de Dinechin. **Opérateurs trigonométriques en virgule flottante sur FPGA**. In *RenPar'17, SympA'2006, CFSE'5 et JC'2006*, pages 96-105, Perpignan, France, October 2006.
Cet article est une continuation du précédent. Il démontre l'implémentation d'un opérateur trigonométrique dual sinus/cosinus. Il discute également différentes variantes de la spécification qui permettent d'obtenir des opérateurs plus compacts.

[56] Jérémie Detrey, Florent de Dinechin, Xavier Pujol. **Return of the hardware floating-point elementary function**. In *18th Symposium on Computer Arithmetic*. IEEE Computer Society Press, June 2007. A paraître.
Cet article montre comment une réduction d'argument itérative reciblée pour la structure fine des FPGA permet d'obtenir des fonctions élémentaires flottantes jusqu'à la double précision avec un coût matériel acceptable.

[27] Florent de Dinechin. **The price of routing in FPGAs**. *Journal of Universal Computer Science*, 6(2) :227-239, February 2000.
Cet article étudie l'évolution des architectures de FPGA.

[33] Florent de Dinechin, Christoph Quirin Lauter, Jean-Michel Muller. **Fast and correctly rounded logarithms in double-precision**. *Theoretical Informatics and Applications*, 41 :85-102, 2007.
Cet article décrit en détail une fonction de CRLibm avec ses différentes implémentations. Il présente les techniques mises en œuvre pour obtenir l'arrondi correct d'une fonction élémentaire en un temps moyen comparable à celui d'une bibliothèque standard, avec un temps au pire cas acceptable, et avec une garantie de la propriété d'arrondi correct.

[31] Florent de Dinechin, Alexey Ershov, and Nicolas Gast. **Towards the post-ultimate libm**. In 17th Symposium on Computer Arithmetic, pages 288-295. IEEE Computer Society Press, June 2005.
Cet article mesure le coût intrinsèque de l'arrondi correct, en terme de temps moyen et de temps au pire cas, en particulier dans le cas où l'on s'affranchit de l'exigence de portabilité.

[28] Florent de Dinechin, David Defour. **Software carry-save : A case study for instruction-level parallelism**. In *Seventh International Conference on Parallel Computing Technologies*, September 2003.
Cet article étudie la performance de l'arithmétique au format SCS sur les architectures de processeur modernes.

[32] Florent de Dinechin, Christoph Lauter, Guillaume Melquiond. **Assisted verification of elementary functions using Gappa**. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318-1322, 2006.
Cet article présente l'outil Gappa et sa mise en œuvre pour obtenir une preuve formelle de la borne d'erreur d'un calcul. Nous attachons en fait le rapport de recherche dont il est issu, qui est plus complet.

[35] Florent de Dinechin, Sergey Maidanov. **Software techniques for perfect elementary functions in floating-point interval arithmetic**. In *Real Numbers and Computers*, July 2006.
Cet article discute l'implementation de fonctions élémentaires pour l'arithmétique d'intervalle «parfaites», c'est-à-dire prouvées, retournant le plus petit intervalle possible, et moins de deux fois plus lente que les fonctions scalaires.

Publications appended to this document

[37] Florent de Dinechin, Arnaud Tisserand. **Multipartite table methods**. *IEEE Transactions on Computers*, 54(3) :319-330, 2005.
This article covers the subject of hardware approximation methods using only tables and additions.

[52] Jérémie Detrey, Florent de Dinechin. **Table-based polynomials for fast hardware function evaluation**. In *16th Intl Conference on Application-specific Systems, Architectures and Processors*. IEEE Computer Society Press, July 2005.
This article introduces higher-order approximation methods for hardware function evaluation. More details can be found in Chapter 2 of Jérémie Detrey's thesis (in French).

[55] Jérémie Detrey, Florent de Dinechin. **A tool for unbiased comparison between logarithmic and floating-point arithmetic**. *Journal of VLSI Signal Processing*, 2007. To appear.
This article introduces two libraries, one for for floating-point and the other one for logarithmic arithmetic. It claims that this choice allows one to choose the arithmetic best suited to a given application, considering performance and accuracy constraints.

[54] Jérémie Detrey, Florent de Dinechin. **Parameterized floating-point logarithm and exponential functions for FPGAs**. *J. Microprocessors and Microsystems*. Elsevier, 2007. To appear.
This article presents hardware-oriented algorithms for floating-point exponential and logarithm operators. Thanks to an original table-based approach, the operators have very small latency, and their area is acceptable up to single precision.

[53] Jérémie Detrey, Florent de Dinechin. **Opérateurs trigonométriques en virgule flottante sur FPGA**. In *RenPar'17, SympA'2006, CFSE'5 et JC'2006*, pages 96-105, Perpignan, France, October 2006.
This article is a follow-up to the previous, for a dual sine/cosine operator. It also discusses several variants of the specification. Just for you I also attach an english version submitted (and just accepted) to the FPL conference.

[56] Jérémie Detrey, Florent de Dinechin, Xavier Pujol. **Return of the hardware floating-point elementary function**. In *18th Symposium on Computer Arithmetic*. IEEE Computer Society Press, June 2007. To appear.
This article shows how an iterative argument reduction of exponential and logarithm can be retargeted to the fine-grained structure of FPGAs. This leads to floating-point operators that scale well to double-precision.

[27] Florent de Dinechin. **The price of routing in FPGAs**. *Journal of Universal Computer Science*, 6(2) :227-239, February 2000.
This article was a study of the trends in FPGA architectures. It has a very nice conclusion which fails to offer any useful solution to Los Angeles traffic-jams.

[33] Florent de Dinechin, Christoph Quirin Lauter, Jean-Michel Muller. **Fast and correctly rounded logarithms in double-precision**. *Theoretical Informatics and Applications*, 41 :85-102, 2007.
This article details the implementations in CRLibm of the logarithm function.

[31] Florent de Dinechin, Alexey Ershov, and Nicolas Gast. **Towards the post-ultimate libm**. In 17th Symposium on Computer Arithmetic, pages 288-295. IEEE Computer Society Press, June 2005.
This article presents measures of the intrinsic cost of correct rounding, including experiments with processor-optimized, non-portable code.

[28] Florent de Dinechin, David Defour. **Software carry-save : A case study for instruction-level parallelism**. In *Seventh International Conference on Parallel Computing Technologies*, September 2003.
This article studies the performance of the SCS multiple precision format on modern processor architectures.

[32] Florent de Dinechin, Christoph Lauter, Guillaume Melquiond. **Assisted verification of elementary functions using Gappa**. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318-1322, 2006.
This article introduces the Gappa tool and its use for the validation of elementary function implementations. Attached is actually the extended research report.

[35] Florent de Dinechin, Sergey Maidanov. **Software techniques for perfect elementary functions in floating-point interval arithmetic**. In *Real Numbers and Computers*, July 2006.
This article discusses the implementation of interval elementary functions which are "perfect" : they return the smallest possible interval, they are fully proven, and their performance is within a factor two of the corresponding scalar function.