Synthèse de code avec compromis entre performance et précision en arithmétique flottante IEEE 754

Laurent Thévenoux

ACADÉMIE DE MONTPELLIER
UNIVERSITÉ DE **P**ERPIGNAN **V**IA **D**OMITIA

THÈSE

présentée à l'Université de Perpignan Via Domitia
dans l'équipe-projet DALI-LIRMM pour
obtenir le diplôme de doctorat

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Énergie et Environnement**

Synthèse de code avec compromis entre performance et précision en arithmétique flottante IEEE 754

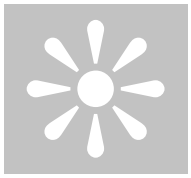
par

Laurent THÉVENOUX

Soutenue le 4 juillet 2014, devant le jury composé de :

Président :	M. Jean-Michel MULLER, Directeur de recherches	CNRS
Rapporteurs :	M. Jean-Marie CHESNEAUX, Professeur	Université Pierre et Marie CURIE
	Mme Sylvie PUTOT, Maître de conférence	CEA/Saclay, École Polytechnique
Directeurs :	M. Philippe LANGLOIS, Professeur	Université de Perpignan Via Domitia
	M. Matthieu MARTEL, Maître de conférence	Université de Perpignan Via Domitia

À Eva



REMERCIEMENTS

Tout d'abord, je tiens à remercier les personnes ayant participé à la validation scientifique de ces travaux. Mes directeurs, mon jury et mon comité de suivi ! Merci à mes directeurs, Philippe LANGLOIS et Matthieu MARTEL pour leur encadrement, ainsi que pour la confiance et la liberté qu'ils m'ont accordées. Merci à mes rapporteurs, pour le temps qu'ils ont consacré à la relecture de cette thèse. D'autant plus que les délais ont parfois été très courts. Merci Jean-Marie CHESNEAUX et Sylvie PUTOT pour votre réactivité et pour vos remarques qui m'ont permis d'améliorer la clarté et la justesse des travaux exposés dans ce document. Merci aussi à Jean-Michel MULLER d'avoir complété mon jury. Je souhaite de plus adresser un remerciement particulier aux membres de mon comité de suivi de thèse, qui ont suivi l'avancée de mes travaux d'année en année : Bernard GOOSSENS et Laurent IMBERT. Enfin, merci au membres de l'équipe DALI pour m'avoir accueilli et prodigué de nombreux conseils, sans oublier Sylvia pour son travail plus qu'efficace dans les affaires administratives. Et bien sûr, merci à tous les doctorants (et jeunes docteurs) qui ont croisé mon chemin. Particulièrement, ceux ayant partagé mon bureau pour nos échanges toujours enrichissant ! Je ne cite pas vos noms, mais vous vous reconnaîtrez à la lecture de ces phrases je l'espère !

D'un point de vue plus personnel, je souhaite remercier ma famille qui à toujours été présente. Merci Maman et Papa d'avoir rendu tout ça possible. Enfin, un très grand merci à Eva, pour son soutien et sa patience qui a été mise à rude épreuve ces derniers mois.

Pour finir, merci à vous et bonne lecture !

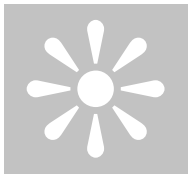
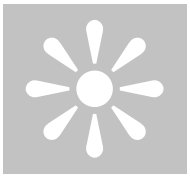


TABLE DES MATIÈRES



AVANT PROPOS

Motivations

Des préoccupations majeures en informatique

L'AVÈNEMENT de l'informatique a permis de s'affranchir d'un grand nombre de tâches répétitives et fastidieuses. De la bureautique au calcul scientifique, l'informatique permet le développement de plus en plus efficace de ces applications. La mise en page de ce document ou l'analyse des exemples d'applications scientifiques qui le composent, sont de bons exemples de ce que l'informatique permet : des procédures répétées de très nombreuses fois, de façon automatique et rapide. Le tout, libre d'erreurs humaines...

Cependant, depuis l'apparition des premiers ordinateurs, les faits nuancent cette dernière affirmation. Premièrement, la *notion de rapidité* est toute à fait relative. C'est pourquoi l'étude des performances (au sens large) et de leurs améliorations font parties depuis toujours des préoccupations majeures lors du développement d'applications informatiques [Hop69, BGS94, HP12b]. Considérons le temps de calcul requis à l'exécution d'une tâche comme critère de performance. Nous constatons que même dans les environnements où le temps n'est pas une ressource critique, de nombreuses études tentent de l'optimiser. Ne serait-ce par exemple, que le démarrage d'une machine, la compilation ou l'exécution d'une tache donnée, le temps fait l'objet de recherches toujours plus poussées [Bir04, ALSU06, DRV00]. Le temps est un *critère fondamental* en informatique, si bien que même la notion de temps elle même, et la manière de le mesurer, est sujette à de nombreuses études [LPGP12, Bai91, WD10].

Secondement, l'erreur humaine est toujours possible tant que l'homme produira lui-même du code informatique. Mis à part les erreurs d'« attention », nous ne sommes pas toujours conscients et responsables des erreurs qui peuvent survenir au sein de nos machines. Par exemple, les arithmétiques employées au sein des ordinateurs peuvent provoquer des *erreurs de précision numérique* [Gol91, Hig93, Hig02, MBdD+10]. De par le fait qu'il soit impossible de représenter l'ensemble infini des nombres réels sur une machine qui ne dispose que de ressources finies, l'ordinateur effectue des approximations plus ou moins fidèles des nombres réels qu'il manipule. Ces approximations sont la principale source d'erreurs affectant la précision numérique des calculs. À l'instar des performances, la précision numérique occupe une place de choix dans les préoccupations des chercheurs en informatique [BBDN10, HLB01, GLL09, Rum09].

Certaines catastrophes illustrent parfaitement cette problématique. Le 20 février 1992 un missile Patriot rate sa cible, un Scud irakien, lors de la première guerre du golfe [BOB92]. L'interception échouée du Scud par le missile Patriot est provoquée par une erreur de logiciel dans son système de coordination. Le logiciel utilisant une valeur impossible à représenter correctement en binaire. Les erreurs d'approximations successives ont finies par causer cette erreur qui coûta la vie à 28 des belligérants. Le 4 juin 1996, le vol inaugural de la fusée Ariane 5 se solde par un échec [LHHL96]. Le rapport conclu à un dépassement de capacité. C'est-à-dire, qu'à un moment, une valeur est devenue trop grande pour être représentée correctement. L'incident coûta des centaines de millions d'euros. Ces erreurs, auraient pu être évitée avec plus de contrôle lors du développement ou de l'utilisation de ces applications. Le constat est sévère, mais quelque soit la source de l'erreur, la faute revient presque toujours aux concepteurs (ou utilisateurs) de ces applications. L'ordinateur, lui, faisant strictement ce qu'on lui demande de faire. Ainsi, la recherche développe de nombreuses méthodes d'aide et de vérification. Ces méthodes assistent les concepteurs lors du développement de leurs applications, afin de les analyser, les améliorer et les certifier [The04, GLL09, MR11, DGP+09, IM12].

Approche : des compromis entre performances et précision

Nous introduisons ici trois notions importantes de notre approche. La première est la *précision numérique* des résultats finaux des algorithmes déployés au sein de codes informatiques. La deuxième est la *performance* de ces codes en terme de temps d'exécution. Concrètement, les critères qui nous permettent de caractériser ces performances sont : (i) le temps de calcul chronométré ou mesuré précisément à partir du nombre de cycles ; et (ii), le niveau de parallélisme d'instructions. Troisièmement, notre approche s'intéresse aux *compromis* entre ces deux premières notions.

En effet, comme nous l'avons vu à la section précédente, les chercheurs développent régulièrement des techniques pour améliorer toujours plus les performances et la précision. Cependant, l'amélioration de l'un est généralement dégradante pour l'autre : produire du code précis et sûr pénalise les performances et vice-versa. C'est dans ce contexte

que nous proposons une méthode, qui concilie ces deux aspects afin de fournir des programmes avec *compromis* entre performances et précision. L'idée est d'améliorer la précision en minimisant les effets sur les performances.

Pour des applications où la précision et les performances ne sont pas cruciales, nous profitons des ressources disponibles pour gagner en précision tout en contrôlant le surcoût en temps d'exécution. Le but est de fournir aux concepteurs de programmes utilisant l'arithmétique flottante, des techniques *automatiques* pour en améliorer la précision tout en préservant un certain degré de performances. Les compromis performances–précision représentent alors les attentes de ces utilisateurs.

D'un côté, afin d'améliorer la précision, nous proposons d'automatiser l'utilisation des transformations sans erreur [LMT10b, LMT10a, LMT12]. Celles-ci permettent de corriger les erreurs d'un calcul en arithmétique flottante avec de meilleures performances par rapport à d'autres méthodes automatiques, telles que les expansions. Nous verrons que la transformation complète d'un programme équivaut à sa compensation. D'un autre côté, contrôler le surcoût sur les performances se traduit par une application sélective des transformations sans erreur. Nous parlerons alors de compensation « partielle » ou « amoindrie ». Nous proposons pour se faire diverses stratégies d'optimisation, afin de fournir des codes partiellement transformés.

Objectif : l'automatisation par la synthèse de code

La *synthèse* de code [Gul10, SGF10] consiste à produire un code à partir des attentes d'un utilisateur exprimées sous la forme de spécifications. Dans le cadre de nos travaux, les spécifications des utilisateurs sont des compromis entre performances et précision. L'utilisateur spécifie l'importance de ces critères selon ses besoins. Par exemple, si le temps d'exécution est plus important que la précision, la synthèse doit produire un programme améliorant la précision pour un faible surcoût.

Les stratégies d'amélioration partielle de la précision permettent d'obtenir des codes transformés prenant en compte des compromis sur les critères de performances et de précision. Cependant, il n'est pas possible de connaître a priori le bon choix parmi l'ensemble des codes transformés. C'est-à-dire, la transformation qui produit la meilleure réponse selon un compromis. C'est pourquoi, le code optimisé est obtenu à l'aide d'une recherche parmi une multitude de choix possibles de codes transformés. Cet ensemble de transformations est généré automatiquement au sein d'un outil de *synthèse* afin de fournir le meilleur code correspondant aux critères imposés.

L'*objectif final* de nos travaux est de permettre une première approche dans l'optimisation multi-critères performances-précision de programmes utilisant l'arithmétique flottante IEEE 754. Cette optimisation nécessite donc l'automatisation de (i) l'amélioration de la précision, via l'utilisation de compensations ; et (ii), la recherche d'une stratégie de

compensation amoindrie permettant d'en réduire l'effet sur les performances. Cette automatisation est assurée par la synthèse de code, qui, à partir d'un programme source et des attentes d'un utilisateur (i.e. les compromis entre performances et précision acceptés), permet la production d'un code optimisé le plus approprié possible.

Contributions

Les recherches effectuées lors de nos travaux apportent les contributions suivantes dans le cadre de l'optimisation multi-critères entre performances et précision numérique.

Automatiser l'amélioration de la précision

Certains outils permettent déjà d'automatiser l'amélioration de la précision par compensation, notamment les méthodes par différenciation automatique [BL02, HP12a]. Néanmoins, notre approche suit un modèle différent répondant à nos contraintes. En effet, les méthodes par différenciation automatique s'appliquent généralement par surcharge. Le temps de calcul étant un de nos critères d'optimisation, nous proposons alors des transformations de programme *source à source* afin de fournir de meilleures performances [BBL+02, HP12a]. La transformation source à source est de plus motivée par le besoin d'opérer des transformations spéciales pour un meilleur contrôle sur (i), les opérateurs destinés à la gestion des erreurs de calculs et (ii), le temps de calcul via une gestion plus fine du surcoût induit.

Nous avons deux classes d'opérateurs à disposition. La première permet de compenser un programme, via l'application systématique de transformations sans erreur et de leurs gestion. La seconde classe d'opérateurs, dits « opérateurs hybrides », fournie uniquement la gestion des erreurs sans l'utilisation des transformations sans erreur. Ces opérateurs ne gèrent que les erreurs héritées des calculs précédents (transformés par les opérateurs de la première classe) sans toutefois prendre en compte les erreurs que les calculs en cours génèrent. Cette seconde classe d'opérateurs hybrides, permet une compensation partielle (ou amoindrie) des programmes.

Nous proposons donc des opérateurs compensés pour la somme et le produit ainsi que leurs variantes hybrides. La transformation automatique permet alors de corriger les erreurs d'approximations commises lors des calculs des opérations élémentaires, tout en propageant les différentes sources d'erreurs à travers le programme.

Proposer des optimisations multi-critères performances-précision

L'amélioration d'un programme informatique nécessite sa transformation pour répondre à des critères d'optimisation. Nous proposons d'améliorer la précision à l'aide des transformations sans erreur qui ont des conséquences regrettables sur les performances des temps

d'exécution. Afin de réduire ces effets, nous proposons de ne les appliquer que partiellement. Cette application « sélective » soulève l'interrogation suivante : comment trouver parmi l'ensemble des transformations possibles, celle qui répondra le mieux aux critères de performances et de précision désirés ?

Pour y répondre, nous allions stratégies d'optimisations et synthèse de code. Deux types de stratégies émergent pour transformer et appliquer partiellement la compensation dans un programme. D'une part, les transformations opérant directement sur le code. Elles modifient les boucles qu'il contient à l'aide de techniques inspirées par le dépliage ou la fission des boucles. D'autre part, une technique moins statique appliquant les transformations sans erreur aux endroits où elles sont les plus grandes. Les stratégies d'optimisations permettent d'effectuer des transformations ayant des impacts différents sur les critères de performances et de précision. Par exemple, un programme ayant subit beaucoup de transformations présentera un temps de calcul augmenté en conséquence. Plus un programme aura subit de transformations, plus sa précision sera susceptible d'augmenter. C'est là qu'intervient la synthèse de code pour faire le choix du programme transformé présentant un bon compromis entre précision et temps de calcul.

Les outils CoHD et SyHD

Nos travaux sont implémentés par deux logiciels : *CoHD* et *SyHD*.

CoHD C'est un outil de transformation de code écrit en OCAML. Cet outil implémente les travaux présentés aux chapitres 5 et 6. Il est la base sur laquelle la synthèse de code opère mais peut être utilisé seul, dans un but de profilage ou d'optimisation manuelle de code. En effet, cet outil se charge de transformer le code source original, en un code cible transformé suivant les divers paramètres des stratégies d'optimisations. CoHD accepte en entrée des programmes en langage C. Il se base sur le *parser-lexer* FRONTC [Cas00] développé par le groupe de recherches TRACES à l'institut de recherche en informatique de Toulouse (IRIT). Notre outil est conçu à la manière d'un compilateur où les optimisations sont appliquées, en différentes passes, à la représentation intermédiaire du programme. Cette organisation facilite l'intégration de nouvelles passes. Il permet de plus l'exportation de l'arbre de représentation intermédiaire dans le format DOT de GRAPHVIZ.

SyHD C'est un synthétiseur de code écrit en OCAML. Cet outil implémente les travaux présentés au chapitre 7. Il est prévu pour être utilisé conjointement avec l'outil CoHD. SyHD recherche un code optimisé répondant le mieux aux compromis entre les critères de performances et précision qui lui sont spécifiés en paramètres. SyHD est entièrement automatique, mais un mode manuel permet à l'utilisateur d'effectuer ses propres optimisations. L'outil est conçu de façon modulaire afin de pouvoir être enrichi avec de nouvelles techniques de recherches pour améliorer la synthèse de

code. De plus, SyHD se base sur différents outils, tel que PAPI qui permet de mesurer les performances d'une application à l'aide des compteurs matériels du processeur [MBDH99]. Il utilise un compilateur C (GCC par défaut) afin de compiler les programmes qu'il génère dans le cadre de la synthèse de code. Il permet aussi l'exportation des résultats à l'aide de GNUPLOT afin d'en fournir une interprétation graphique.

Validation expérimentale

À l'aide des outils CoHD et SyHD, nous avons mis en pratique notre approche au travers de nombreuses expériences concernant l'optimisation multi-critères entre performances et précision. Nous avons étudié de nombreux cas pathologiques comme celui de la somme [ROO05] et ceux d'évaluations polynomiales [GLL09, JGH+13, JLCS10, JBL+11]. Ces cas disposent d'algorithmes compensés que nous avons pu utiliser comme référence (en plus des algorithmes *double-double*) pour contrôler le bon fonctionnement de notre approche. Nous avons ensuite étendu ces études afin d'y intégrer les optimisations avec des compromis performances-précision. Enfin, nous avons étudié un autre cas, celui de la résolution de système linéaire [DHK+06] par raffinement itératif. Ce cas n'a pas encore fait l'objet de recherches visant à en fournir une version compensée. Nous avons alors observé comment notre méthode permet d'en améliorer la précision et les performances. Nous avons consacré le chapitre 8 à la présentation des principaux résultats expérimentaux. Les résultats obtenus ont confirmé l'intérêt et le potentiel de notre approche.

Plan du document

Cette thèse s'articule autour de deux parties distinctes. La première partie présente l'état de l'art des différents aspects techniques sur lesquels ces travaux s'appuient. Les trois chapitres de cette partie présentent les bases des divers domaines informatiques rencontrés : la *transformation de code*, les *performances* et la *précision en arithmétique flottante* IEEE 754. L'arithmétique flottante IEEE 754 [75408, MBdD+10] est présentée au chapitre 1 grâce à la description des principales caractéristique de la norme et par l'ouverture sur un des aspects qui motive ce travail : la précision numérique des calculs effectués sur ordinateur. Le chapitre 2 présente un ensemble de méthodes permettant d'effectuer des calculs avec plus de précision. Nous y présentons les transformations sans erreur [Dek71, Knu97, PV93], les compensations [ROO05, GLL09] et les expansions [She97, Lau05, HLB01]. Enfin, le chapitre 3 présente la manipulation de programme à travers la compilation [ALSU06] et la synthèse [Gul10]. Ce chapitre aborde de plus l'une des difficultés majeures de l'évaluation des performances d'une application sur ordinateur, à savoir la fiabilité des mesures [Bai91, LPGP12, WD10].

La seconde partie, quant à elle, expose les travaux que nous avons réalisés pour l'étude des compromis entre performances et précision. Notre contribution dans ces recherches est détaillée au travers de cinq chapitres. Le chapitre 4 introduit en premier lieu une étude de cas préliminaire portant sur la somme de n termes en arithmétique flottante [LMT10b]. Cette étude vise à montrer que des compromis exploitables existent au sein des expressions arithmétiques, et donc, au sein des programmes qui les contiennent. L'étude permet de se rendre compte que, en règle générale, la précision et les performances ne s'accordent pas, et de par ce fait, une expression performante n'est pas des plus précise et inversement. Grâce à une étude exhaustive de réécritures de la somme, nous avons montré que des compromis performances-précision sont exploitables, et parfois même sans réelle conséquence sur les performances en pratique.

L'approche développée dans les chapitres suivants s'écarte de celle présentée dans le chapitre 4, par la méthode employée pour améliorer la précision. En effet, l'étude précédente à mis en exergue le fait que la recherche d'une expression *optimale* nécessite une étude exhaustive bien souvent impossible dû à l'explosion combinatoire qu'offre les expressions en arithmétique flottante. Bien que cet aspect ne soit pas retenu pour nos travaux, il n'en est pas moins étudié par d'autres chercheurs [Iou13, IM12]. Notre approche consiste à l'amélioration de la précision par ajout automatique de compensation. Le chapitre 5 présente comment l'application automatique des compensations s'opère au sein des codes informatiques [LMT10a, LMT12]. Nous montrons dans ce chapitre que l'application automatique de compensation permet de doubler la précision numérique à l'instar d'une autre méthode automatique à base d'expansions. Nous constatons de plus que l'utilisation des compensations automatiques, est une meilleure candidate par rapport aux expansions car elle permet un parallélisme d'instruction bien plus élevé. L'utilisation des compensations introduisant un coût non négligeable par rapport au coût du programme original, des compromis sont alors à faire sur ce point.

C'est donc via l'application partielle des compensations que nous cherchons à faire des compromis. Nous avons développé pour ce faire différentes stratégies d'optimisations afin de proposer des compromis entre performances et précision. Ces stratégies sont présentées dans le chapitre 6. Elles permettent de réaliser des compromis performances-précision en ne compensant que partiellement les calculs. Ces stratégies proposent une multitude de choix possibles sans toutefois permettre a priori l'émergence d'un code répondant au compromis demandé. C'est pourquoi, le chapitre 7 présente comment, grâce à la synthèse de code, nous proposons la recherche d'un programme disposant d'un bon compromis performances-précision suivant des contraintes établies. Ce chapitre présente de plus les outils informatiques CoHD et SyHD que nous avons été amenés à développer afin de réaliser cette synthèse de code. Enfin, le cinquième et dernier chapitre de cette seconde partie présente quelques résultats expérimentaux obtenus à l'aide des outils évoqués au chapitre précédent (voir chapitre 8).

Enfin, nous concluons cette thèse en résumant nos travaux et en y proposant de nombreuses perspectives. De plus, le chapitre 4, est une version améliorée de l'article [LMT10b], qui à servi de point de départ à cette thèse. Les travaux regroupés dans ce document ont été présentés lors de conférences, telles que [LMT10a] (voir chapitre 4) et [LMT12] (voir chapitre 5).

Notes sur la version électronique Pour une lecture optimale des figures de ce document, nous conseillons au lecteur de se reporter à la version électronique disponible sur internet.

Première partie

État de l’art

CHAPITRE 1

ARITHMÉTIQUE FLOTTANTE IEEE 754 ET PRÉCISION NUMÉRIQUE

Préambule

L'arithmétique flottante est une représentation informatique de l'arithmétique réelle. Dès la conception des premières machines capables d'effectuer des calculs faisant intervenir des nombres flottants, la question de la standardisation de leur représentation c'est naturellement posée. Si les débuts ont pu sembler chaotiques, très vite de nombreux industriels ont adopté la norme qui s'est imposée de nos jours sur l'ensemble des unités flottantes : la norme IEEE 754.

1.1 Introduction

L'IEEE *Standard for Binary Floating-point Arithmetic*, plus connu sous le nom de code IEEE 754 apparaît en 1985. Ce standard normalise la représentation des nombres flottants. Il est aujourd'hui le standard de représentation des nombres flottants le plus utilisé[1] et permet aux différents programmes informatiques utilisant cette arithmétique de devenir plus sûrs et portables. Une révision du standard en 2008 lui apportera quelques nouveautés [75408].

[1] Notons que l'IEEE n'est pas le seul organisme à proposer un standard. Citons à titre d'information, le travail de l'organisme international de normalisation (ISO) avec la norme ISO 10967 [10994].

Premièrement, ce chapitre abordera les principales caractéristiques du standard, telles que la représentation des nombres ou le comportement intrinsèque de l'arithmétique flottante. Dans une seconde partie, quelques notions sur l'analyse de la précision numérique seront abordées. Pour une description plus exhaustive de ces notions, nous encourageons la lecture de la norme elle-même [75408], ou les ouvrages et publications suivants [MBdD+10, Hig02, Ove01, Gol91].

1.2 Représentation des nombres

La représentation d'un nombre flottant x est la suivante :

$$x = (-1)^s \cdot m \cdot \beta^e \tag{1.1}$$

où :

$s \in \{0,1\}$	Le signe.
$m = d_0 \cdot d_1 d_2 \cdots d_{p-1}$	La mantisse où d_i est un chiffre entier tel que $0 \leq d_i < \beta$ et p est la précision (le nombre de chiffres significatifs).
β	La base.
e	L'exposant définit par : $e_{\min} \leq e \leq e_{\max}$ avec $e_{\min} = 2 - 2^{n-1} = 1 - e_{\max}$ où n est le nombre de bits utilisés pour coder l'exposant. Pouvant être positif ou négatif, l'exposant est décalé afin de faciliter la manipulation des nombres en machine. En effet, il est plus difficile de comparer des nombres signés que non signés. De ce fait, le décalage (ou biais) est fixé à $b = 2^{n-1} - 1 = e_{\max}$, ce qui permet la représentation non-signée de l'exposant.

Par convention, en base 2, le chiffre d_0 n'est pas représenté dans l'encodage du nombre, il est implicite et appelé partie entière. L'exposant est fixé de façon à ce que ce chiffre soit non-nul, ce qui permet une représentation normalisée des nombres et garantit qu'un nombre ne peut avoir qu'une unique représentation. Lorsque l'exposant est nul on obtient

des nombres dénormalisés. La partie de la mantisse comprenant les chiffres $d_1 d_2 \cdots d_{p-1}$ est aussi appelée partie fractionnaire. Les valeurs $e_{\min} - 1$ et $e_{\max} + 1$ existent mais sont réservées à la représentation de zéro, des nombres dénormalisés, des infinis et des `NaN` (de l'anglais *Not a Number* pour désigner un nombre non représentable : $x \div 0, \sqrt{-x}, \ldots$).

L'objectif des nombres dénormalisés est d'uniformiser la répartition des nombres représentables autour de 0. En effet, le 1 implicite dans la mantisse implique qu'il n'y a pas de nombre représentable entre 0 et $2^{e_{\min}+1}$. La figure 1.1 synthétise les caractéristiques de l'ensemble des nombres flottants.

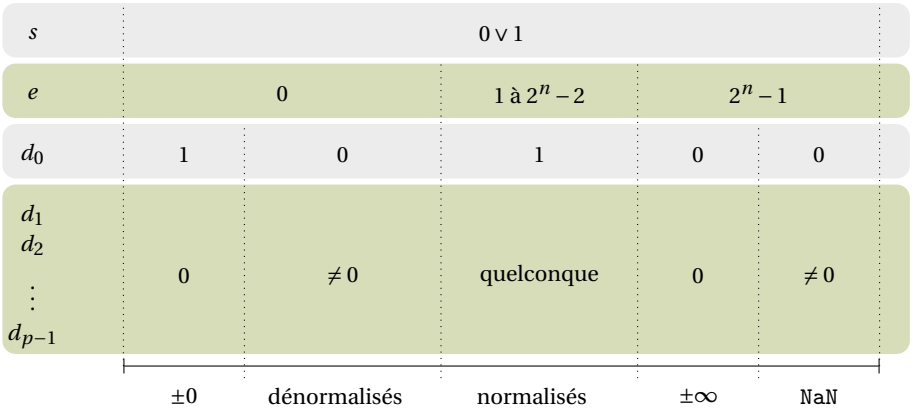


FIGURE 1.1 – Caractéristiques des nombres flottants dans l'arithmétique standardisée IEEE 754. Les nombres normalisés comportent l'ensemble des zéros signés et des nombres non-nuls. Les nombres dénormalisés comportent un petit ensemble de nombres non-nuls autour de 0, les infinis et les `NaN`.

1.2.1 Les formats

Les formats de représentation sont caractérisés par une base β, une précision p et un ensemble d'exposants $\{e_{\min}, \cdots, e_{\max}\}$. Il existe cinq formats principaux : trois formats binaires, encodés sur 32, 64 et 128 bits ainsi que deux formats décimaux encodés sur 64 et 128 bits. Seul les formats binaires seront abordés ici. La table 1.1 en donne les différentes caractéristiques et quelques exemples de nombres flottants du format *binary32* sont donnés dans la table 1.2.

format	p	$e_{\max}$	nombre de bits s	e	m
binary32	24	127		8	23
binary64	53	1023	1	11	52
binary128	113	16383		15	112

TABLE 1.1 – Caractéristiques principales des formats de représentation binaire de l'arithmétique IEEE 754. Les formats *binary32, binary64* et *binary128* sont plus communément appelés formats simple, double et quadruple précision.

s	e	d_0	$d_1 d_2 \cdots d_{23}$	Type	Valeur
$0 \vee 1$	0000 0000	0	000 0000 0000 0000 0000 0000	zéro	$\pm 0,0$
0	0111 1111	1	000 0000 0000 0000 0000 0000	normalisé	$+1,0$
$0 \vee 1$	1111 1110	1	111 1111 1111 1111 1111 1111	normalisé	$\pm 3,4 \times 10^{38}$
$0 \vee 1$	0000 0001	1	000 0000 0000 0000 0000 0000	normalisé	$\pm 1,18 \times 10^{-38}$
1	0000 0000	0	100 0000 0000 0000 0000 0000	dénormalisé	$-5,9 \times 10^{-39}$
$0 \vee 1$	1111 1111	0	000 0000 0000 0000 0000 0000	infini	$\pm\infty$
$0 \vee 1$	1111 1111	0	010 0000 0000 0000 0000 0000	NaN	`NaN`

TABLE 1.2 – Exemples de représentation de nombres codés en arithmétiques IEEE 754 dans le format *binary32* (format simple précision codé sur 32 bits).

1.3 Comportements de l'arithmétique flottante

Cette section aborde les comportements de l'arithmétique flottante, tant au niveau des nombres eux-mêmes, qu'au niveau des opérations arithmétiques élémentaires en décrivant les pièges que l'on peut rencontrer.

1.3.1 Les modes d'arrondi

L'arithmétique flottante étant une arithmétique approchée, tous les nombres réels ainsi que les résultats des opérations arithmétiques (flottantes) élémentaires passent par une phase d'arrondi. L'arrondi spécifie le moyen de passer d'une représentation « exacte » potentiellement encodée sur un nombre infini de bits, à la représentation finie des nombres flottants.

Il existe quatre modes d'arrondi définis par la norme : au plus près, noté $RN(\cdot)$; vers zéro, noté $RZ(\cdot)$; et les arrondis vers le bas ($-\infty$) noté $RD(\cdot)$ et vers le haut ($+\infty$) noté $RU(\cdot)$, où $\cdot$ représente un nombre ou une suite d'opérations arithmétiques. Les calculs entre les

parenthèses sont arrondis dans le mode désigné. La notation $fl(\cdot)$ sera utilisée pour représenter une suite d'opérations pouvant être effectuée indépendamment du mode d'arrondi (sous réserve qu'aucune autre indication ne soit fournie). La figure 1.2 présente des exemples d'arrondis pour trois nombres réels différents.

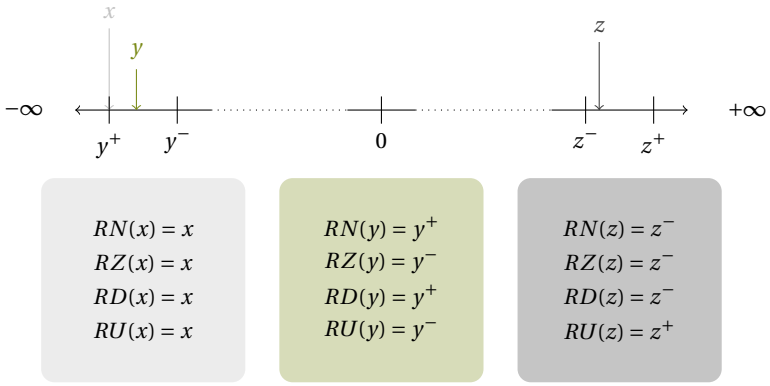


FIGURE 1.2 – Arrondis des nombres $x, y \in \mathbb{R}_-^*$ et $z \in \mathbb{R}_+^*$ selon les quatre modes défini par la norme IEEE 754.

1.3.2 Les sources d'erreurs

L'arithmétique flottante peut dans certains cas provoquer des résultats incorrects (voir par exemple le cas de l'algèbre linéaire [Lay05, p. 10]). Les erreurs d'arrondis sont la première source d'erreurs. De ces erreurs, découlent les dépassements de capacité ou encore des phénomènes d'absorption et d'élimination. Ces comportements engendrent parfois des bogues catastrophiques [BOB92, LHHL96].

Les erreurs d'arrondi

C'est la principale source d'erreurs de l'arithmétique flottante. Par exemple, la valeur 0,1 à une représentation infinie en flottant. L'arrondi de cette valeur, donc le nombre flottant correspondant ne sera pas exactement égal à 0,1 comme illustré à la figure 1.3.

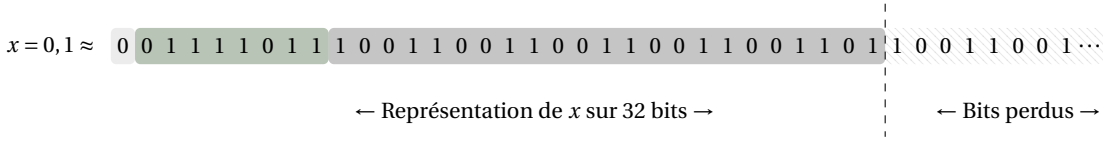


FIGURE 1.3 – Représentation dans le format *binary32* de l'arithmétique IEEE 754 du nombre réel $x = 0,1$ arrondi au plus près.

Les dépassements de capacité

Comme tous les formats de représentation finie de nombres, il est uniquement possible de coder des nombres entre une valeur minimale et maximale. Si un calcul doit produire une valeur supérieure à la valeur maximale représentable, alors le résultat sera la valeur spéciale $\pm\infty$. Il est donc impossible de calculer au-delà de la valeur maximale fixée par le format de représentation utilisé. Ce problème est le problème d'*overflow*. Il est à noter qu'une fois qu'un calcul à produit dans une de ses opérations la valeur spéciale $\pm\infty$ alors toutes les autres opérations utilisant cette valeur comme paramètre produiront la valeur $\pm\infty$ (ou `NaN`, par exemple dans le cas d'une multiplication par zéro). Inversement, il existe le problème d'*underflow* qui se présente lorsque l'on rencontre une valeur comprise entre zéro et la plus petite valeur représentable par le format de représentation utilisé. Dans ce cas, c'est le type d'arrondi choisi qui fixe le comportement à suivre. Un des risques posés par ce problème est la possibilité que la valeur soit arrondi à zéro, entraînant une annulation en cascade (à travers une suite de produits) ou au contraire produire la valeur spéciale ∞ dans le cadre d'une division.

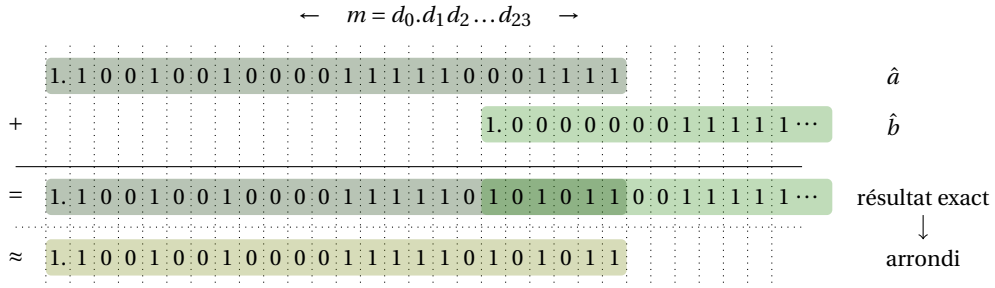
Les phénomènes d'absorption et d'élimination catastrophique

L'*absorption* est due à l'arrondi du résultat d'une addition ayant pour opérandes deux nombres flottants très éloignés. Soit $\hat{a}$ et $\hat{b}$ deux nombres flottants tels que $\hat{a}$ est bien supérieur à $\hat{b}$, une partie de $\hat{b}$ sera perdue dans le résultat de l'addition. On parle d'absorption catastrophique quand tous les chiffres de $\hat{b}$ sont perdus lors de l'addition. Dans ce cas, l'opération $fl(\hat{a}+\hat{b})$ sera égale à $\hat{a}$ alors que $\hat{b}$ aura été entièrement absorbé par $\hat{a}$. La figure 1.4(a) illustre ce phénomène. L'*élimination* survient lors de la soustraction de nombre très proches. À chaque opération il faut renormaliser le résultat en ajoutant des zéros à la mantisse car les chiffres de poids fort s'éliminent. Le résultat peut être très différent du résultat exact, et de plus représenter une valeur totalement erronée due aux imprécisions possibles des opérandes. On parle d'élimination catastrophique lorsqu'il ne reste que quelques chiffres significatifs dans le résultat. La figure 1.4(b) illustre ce phénomène.

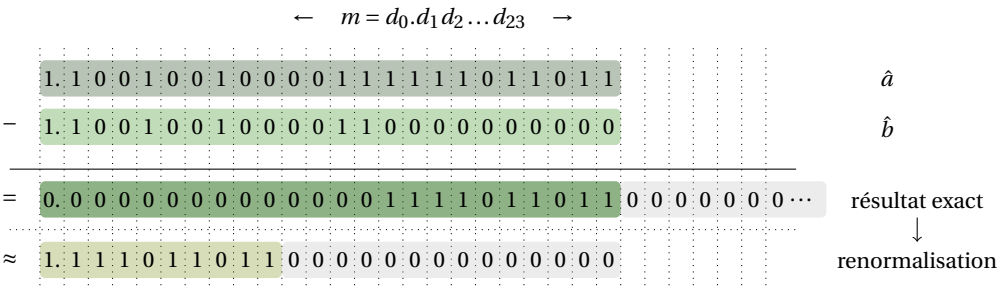
Combinées, ces erreurs peuvent par exemple, provoquer le résultat suivant : $fl((\hat{a}+\hat{b})-\hat{a}) = 0$ au lieu du résultat exact $\hat{b}$.

1.4 Analyse d'erreur de la précision numérique

Les sections précédentes ont mis en exergue le fait que l'arithmétique flottante présente de par ses caractéristiques des sources d'imprécisions. Les sections suivantes décrivent quelques outils et méthodes qui permettent d'analyser ces erreurs afin de caractériser la précision numérique d'un calcul.



(a) Absorption lors de l'addition de deux valeurs $\hat{a}$ et $\hat{b}$ très éloignées. On perd une partie de la mantisse de $\hat{b}$ (et tous les bits lors d'une absorption catastrophique).



(b) Élimination lors de la soustraction de deux nombres $\hat{a}$ et $\hat{b}$ très proches. Il faut renormaliser le résultat en ajoutant des zéros à droite de la mantisse car les bits de gauche s'éliminent. Le résultat d'une telle opération sur des opérandes exactes pourrait faire apparaître des bits non nuls à droite de la mantisse au lieu des zéros que l'on ajoute. C'est pourquoi la valeur calculée peut être très éloignée du résultat exact.

FIGURE 1.4 – Phénomènes d'absorption et d'élimination en arithmétique IEEE 754 (dans le format *binary32*).

1.4.1 Notions utilisées

Soit $\hat{x}$ une approximation d'un nombre réel x. Les mesures les plus utiles pour mesurer la précision de $\hat{x}$ sont l'*erreur absolue* :

$$E_{\text{abs}}(\hat{x}) = |x - \hat{x}|,$$

et l'*erreur relative* :

$$E_{\text{rel}}(\hat{x}) = \frac{|x - \hat{x}|}{|x|} \qquad \forall x \neq 0.$$

L'erreur relative de $\hat{x}$ permet aussi de quantifier le nombre de bits significatifs que partagent $\hat{x}$ et x [Ove01] :

$$N_{\text{Overton}}(\hat{x}) = -\log_2 E_{\text{rel}}(\hat{x}). \tag{1.2}$$

Ce nombre est au plus égal à $p-1$, ou p dans le cas de l'arrondi au plus près.

On peut distinguer deux types de précision. La précision sur le résultat, qui est l'erreur relative entachant une quantité calculée. C'est l'approximation d'un résultat exact provenant d'un algorithme de calcul. Et la précision de calcul, équivalente à la précision de travail et qui désigne l'erreur relative commise lors de chaque opération arithmétique élémentaire $\circ \in \{+,-,\times,\div\}$.

Cette dernière est majorée par l'unité d'arrondi

$$u = \begin{cases} 2^{-p} & \text{si arrondi au plus près,} \\ 2^{1-p} & \text{sinon.} \end{cases} \tag{1.3}$$

Soient deux nombres flottants $\hat{a}$ et $\hat{b}$, $x = \hat{a} \circ \hat{b}$ le résultat exact de l'opération et $\hat{x} = fl(\hat{a} \circ \hat{b})$ le résultat calculé en arithmétique flottante alors on a

$$u \geq E_{\text{rel}}(\hat{x}).$$

L'évaluation d'une opération $\circ \in \{+,-,\times,\div\}$ satisfait, pour tous les modes d'arrondi, les propriétés suivantes (si pas de dépassement de capacité) :

$$fl(\hat{a} \circ \hat{b}) = (\hat{a} \circ \hat{b})(1+\delta_1) = (\hat{a} \circ \hat{b}) \div (1+\delta_2) \tag{1.4}$$

où $|\delta_1|, |\delta_2| \leq u$. Cette relation est aussi désignée comme étant le *modèle standard* de l'arithmétique flottante pour analyser les algorithmes numériques.

1.4.2 Les fonctions *ulp* et *ufp*

La fonction ulp La notion d'`ulp` (de l'anglais *Unit in the Last Place*) peut varier d'une source à l'autre (voir le travail récapitulatif de [Mul05]). Nous considérons l'`ulp` d'un nombre flottant selon la définition (1.5) donnée ci-dessous.

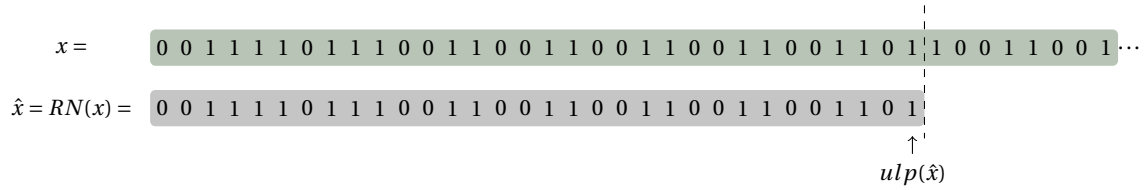
Soit $\hat{x}$ un nombre flottant, l'$ulp(\hat{x})$ est défini par

$$ulp(\hat{x}) = \begin{cases} 2^{e-p} & \text{si arrondi au plus près,} \\ 2^{e+1-p} & \text{sinon.} \end{cases} \tag{1.5}$$

L'`ulp` est le *poids* du dernier bit de la mantisse. On pourra ainsi mesurer une borne d'erreur absolue commise lors d'un arrondi et caractériser la précision d'un nombre en arithmétique flottante. Un exemple est donné à la figure 1.5.

La fonction ufp À l'image de l'`ulp`, la notion d'`ufp` [ROO08] (de l'anglais *Unit in the First Place*) désigne le *poids* du premier bit de la mantisse d'un nombre flottant. L'`ufp` est alors indépendant du nombre de bits de la mantisse.

Soit $\hat{x}$ un nombre flottant, l'$ufp(\hat{x})$ est défini par



$$|x - \hat{x}| \leq ulp(\hat{x}) = 2^{e+1-p} = 2^{-4+1-24} \approx 7.4505805969238e^{-09}$$

FIGURE 1.5 – Arrondi et `ulp` de $x = 0,1$ en arrondi au plus près en arithmétique flottante IEEE 754 (dans le format *binary32*).

$$ufp(\hat{x}) = \begin{cases} 2^{\lfloor log_2 |\hat{x}| \rfloor} & \forall \hat{x} \neq 0, \\ 0 & \text{si } \hat{x} = 0. \end{cases} \tag{1.6}$$

On peut alors faire la relation suivante entre `ulp` et `ufp` :

$$ulp(\hat{x}) = u \cdot ufp(\hat{x}) \text{ si arrondi au plus près, } 2u \cdot ufp(\hat{x}) \text{ sinon.}$$

1.4.3 Analyser la propagation des erreurs d'arrondi

L'analyse de la propagation des erreurs d'arrondi présentée ici, est une méthode qui cherche à mesurer l'écart entre le résultat calculé et la solution exacte. Cette méthode est appelée analyse directe ou encore analyse a priori. Nous présentons ici la *Running Error Bound* [Hig02], qui permet de calculer une borne d'erreur relative de façon dynamique au sein d'algorithmes numériques.

Soit $\hat{a} = a + \alpha$ et $\hat{b} = b + \beta$ où α et β sont les erreurs absolues entachant $\hat{a}$ et $\hat{b}$. À partir du modèle standard (1.4) on dérive des bornes d'erreurs notées $\sigma_\circ$ pour chaque opération élémentaire $\circ \in \{+, -, \times, \div, \sqrt{}\}$ décrit à la table 1.3. La borne d'erreur est calculée en même temps que l'évaluation des opérations l'algorithme. Les erreurs se propagent alors d'opération élémentaire en opération élémentaire jusqu'à fournir une borne pour le résultat du calcul.

Remarquez que dans la table 1.3, chaque borne d'erreur est composée d'un terme d'erreur $u|\cdot|$ lié à l'opération arithmétique $\circ$, tandis que le reste de l'expression propage simplement les erreurs de l'entrée vers la sortie de l'opérateur. Cependant, l'utilisation du terme d'erreur $u|\cdot|$ présente les inconvénients suivants qui peuvent conduire à une surapproximation de la borne d'erreur : la borne est toujours calculée même si l'opération arithmétique ne provoque pas d'erreur, et peut de plus, être deux fois plus grande que l'erreur réelle. Il est possible de contourner ce problème en utilisant des méthodes plus précises pour évaluer l'erreur commise lors de l'opération arithmétique. Dans [ZL10], les

Opération	Borne d'erreur
$+, -$	$\lvert\sigma_{\pm}\rvert \leq u\lvert\hat{a} \pm \hat{b}\rvert + \alpha + \beta$
$\times$	$\lvert\sigma_{\times}\rvert \leq u\lvert\hat{a}\hat{b}\rvert + \alpha\lvert\hat{b}\rvert + \beta\lvert\hat{a}\rvert$
$\div$	$\lvert\sigma_{\div}\rvert \leq u\left\lvert\frac{\hat{a}}{\hat{b}}\right\rvert + \frac{\alpha\lvert\hat{b}\rvert + \beta\lvert\hat{a}\rvert}{\hat{b}^2}$
$\sqrt{}$	$\lvert\sigma_{\sqrt{}}\rvert \leq u\lvert\sqrt{\hat{a}}\rvert + \frac{\alpha}{2\lvert\sqrt{\hat{a}}\rvert}$

TABLE 1.3 – Bornes d'erreurs des opérations élémentaires en analyse d'erreur directe *Running Error Bound*.

auteurs utilisent, par exemple, les registres flottants 80 bits de la FPU (*Floating-Point Unit*) des processeurs modernes afin de calculer plus finement la borne.

Néanmoins, l'analyse directe n'est pas la seule analyse disponible. J. WILKINSON a popularisé, dans les années 60, l'analyse indirecte aussi appelée analyse a posteriori [Wil63]. Enfin, pour une introduction générale à l'analyse numérique voir [Hil87].

1.5 Conclusion

Nous avons abordé dans ce chapitre l'essentiel de l'arithmétique flottante IEEE 754 et nous avons constaté que cette arithmétique pouvait engendrer des pertes de précision.

Nous avons de plus présenté quelques notions et outils que nous utiliserons par la suite pour analyser et mesurer cette précision. Particulièrement, nous avons présenté une méthode d'analyse directe (Running Error Bound) permettant de borner l'erreur d'un calcul. Enfin, nous avons montré qu'il est possible d'affiner cette borne selon le moyen employé pour la calculer. Ceci nous est utile pour remarquer qu'allier avec les transformations sans erreur (détaillées dans le chapitre 2), il est possible de calculer non plus une borne, mais l'erreur du calcul elle-même.

CHAPITRE 2

TRANSFORMATIONS SANS ERREUR, COMPENSATION ET EXPANSION

Préambule

Le constat est simple. En arithmétique flottante, chaque opération élémentaire induit une erreur d'arrondi qui se propage au fil des calculs. En conséquence, l'erreur entachant le résultat final peut dans certains cas le rendre non significatif. C'est là qu'entrent en jeu les transformations sans erreur, les compensations ou encore les expansions afin de limiter les pertes de précision numérique au sein de ces calculs.

2.1 Introduction

LES transformations sans erreur, les compensations ou encore les expansions qui sont présentées dans ce chapitre font partie de la famille des nombreuses solutions permettant de prévenir ou de corriger les comportements indésirables de l'arithmétique flottante. Chaque méthode présente ses avantages et ses inconvénients, mais pour autant, l'une n'est pas forcément meilleure que l'autre. Elles seront privilégiées ou non, suivant le contexte de leur utilisation. Par exemple, les systèmes temps réel emploierons des méthodes qui présentent l'avantage d'être rapide.

On peut distinguer deux types de méthodes, les unes à base de *réécritures*, et les autres à base d'*extension* de la précision des calculs. La réécriture des calculs exploite la non asso-

ciativité des opérations de l'arithmétique flottante. De ce fait, l'expression $(a+b)-a$ sera nulle si $a \gg b$. Ceci est dû au phénomène d'absorption illustré à la figure 1.4(a). Le résultat devient correct en réécrivant l'expression en $(a-a)+b$. Ce phénomène de réécriture sera abordé plus en détails (dans le cas de la somme [LMT10b]) au chapitre 4. Néanmoins, même avec une grande connaissance en arithmétique flottante, ces méthodes nécessitent des outils d'aide à la réécriture permettant de maintenir l'explosion combinatoire, comme par exemple, l'interprétation abstraite [Mar09, DGP+09, IM12]. Ces méthodes ne seront pas abordées ici. L'objet de ce chapitre porte sur le second aspect : étendre la précision des calculs[1].

Il existe plusieurs façons d'étendre la précision des calculs effectués en arithmétique flottante. Une première solution est d'utiliser des algorithmes développés spécialement pour des applications spécifiques. Par exemple, si l'on ne prend que le cas des algorithmes de sommation de n nombres flottants, on constate que la recherche dans ce domaine est très riche [RS88, Hig93, McN04]. De nouveaux algorithmes sont même publiés tous les ans depuis la fin des années 90 [ROO08, ZH10], améliorant encore la précision de l'évaluation de ces sommes. L'aspect qui nous motive dans ce chapitre est l'augmentation « physique » du nombre de bits consacrés à la précision. Ces méthodes permettent une plus grande représentation des nombres flottants, ce qui a pour effet de réduire ou de corriger les erreurs commises lors de calculs. Les méthodes abordées dans les sections suivantes exploitent les transformations sans erreur présentées à la section 2.2. La section 2.3 présente les algorithmes compensés et la section 2.4 traite quant à elle des expansions. Plus particulièrement, elle traite d'une expansion de taille fixe, les *double-word*. Ces expansions sont néanmoins plus communément appelées *double-double*. L'appellation *double-word* permet d'être le plus générique possible et de rester ainsi indépendant du format de représentation des nombres flottants. De ce fait, en les nommant *double-double*, nous faisons référence à des *double-word* où le mot (ou *word*) est un nombre flottant binaire double précision[2], e.g. *binary64*.

Enfin, citons à titre d'information d'autres moyens d'amélioration de la précision. MPFR, qui est un outil simulant des nombres flottants entièrement personnalisables [FHL+07]. Les approches stochastiques, telle que celle implémentée dans CADNA qui permet d'estimer la propagation de erreurs d'arrondis pour les programmes écrit en FORTRAN [JC08]. Ou encore les méthodes différentielles qui permettent d'améliorer la précision des calculs pour certaines classes d'algorithmes, telle que celle implémentée dans CENA [BL02].

[1] Notez toutefois que ces deux méthodes ne sont pas ambivalentes et peuvent être utilisées de concert.

[2] Ainsi nous faisons toujours référence dans les pages de cette thèse à des nombres flottants binaires double précision. Les algorithmes *double-double* présentés dans ce chapitre peuvent donc s'appliquer aux autres formats de représentation binaire de l'arithmétique flottante (par exemple, si on considère le format *binary32*, on nommera l'expansion *double-word* en *double-float*).

2.2 Les transformations sans erreur

Les transformations sans erreur (de *l'anglais Error Free Transformation*, EFT) sont des algorithmes permettant de calculer les erreurs commises par les opérations élémentaires en arithmétique flottante. Nous avons vu dans le chapitre précédent que chaque opération élémentaire passe par une phase de renormalisation. Considérons les mantisses de deux nombres flottants codés sur n chiffres. Pour effectuer une somme ou un produit, il faut en fait $2n$ chiffres de mantisse pour représenter exactement le résultat. Cependant, la phase de renormalisation impose un résultat sur n chiffres, laissant les n chiffres supplémentaires potentiellement significatif à l'abandon. Ce phénomène est illustré dans la figure 1.4 du chapitre 1 pour l'exemple de la somme. Ces transformations sans erreur permettent donc d'effectuer des opérations élémentaires avec deux fois plus de précision que celle fournie par le format utilisé.

Soit $\circ \in \{+,-,\times\}$ des opérations élémentaires effectuées dans le mode d'arrondi au plus près, et $\hat{a}$ et $\hat{b}$ deux nombres flottants encodés dans un des formats binaire ($\beta = 2$) de l'IEEE 754. Alors il existe deux nombres flottants $\hat{x}$ et $\hat{y}$ (dans le même format et sauf certains dépassements de capacité) qui vérifient :

$$RN(\hat{a} \circ \hat{b}) = \hat{x} + \hat{y}.$$

Les EFT produisent alors, pour une opération $RN(\hat{a} \circ \hat{b})$, un résultat exactement représentable par la somme de $\hat{x}$ et $\hat{y}$ où $\hat{x} = RN(\hat{a} \circ \hat{b})$ le plus proche nombre flottant de l'opération $(\hat{a} \circ \hat{b})$ et où $|\hat{y}| \leq \frac{1}{2} ulp(\hat{x})$. L'erreur commise lors de l'opération est alors *exactement* représentée par $\hat{y}$. La relation précédente nous permet de constater que $\hat{x}$ est, pour ainsi dire, la partie haute du résultat, $\hat{y}$ la partie basse et qu'aucun bit de ces deux parties ne se chevauchent. Les algorithmes 2.1, 2.2, 2.3, 2.5 et 2.6 proposent des transformations sans erreur pour les opérations élémentaires $\circ \in \{+,-,\times,\div\}$.

2.2.1 L'addition

L'algorithme 2.1, dû à DEKKER en 1971 [Dek71] calcule la somme exacte de deux nombres flottants a et b tel que $|a| \geq |b|$.

1. **fonction** FASTTWOSUM(a, b)
2. $\quad x \leftarrow RN(a+b)$ $\quad \triangleright\ |a| \geq |b|$
3. $\quad y \leftarrow RN((a-x)+b)$
4. $\quad$ **retourner** $[x,\ y]$
5. **fin fonction**

Algorithme 2.1 – FASTTWOSUM [DEKKER, 1971].

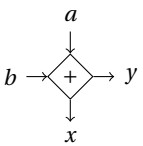


FIGURE 2.1 – Symbole de l'algorithme 2.1.

L'algorithme FASTTWOSUM requiert 3 opérations flottantes et un test préliminaire pour déterminer si $|a| \geq |b|$ lorsque ceux ci sont inconnus. L'algorithme 2.2, évoqué par O. MØLLER en 1965 [Møl65], puis proposé par D. KNUTH en 1969 [Knu97] calcule la somme exacte de deux nombres en 6 opérations flottantes. Malgré le surcout par rapport à FASTTWOSUM, cet algorithme est préféré car il n'utilise aucun branchement conditionnel. Il se révèle même en pratique finalement plus performant. Remarquez que ce dernier algorithme est optimal en nombre d'opérations flottantes et de profondeur de l'arbre de dépendances (i.e. la longueur du chemin critique du graphe de flot de données : la latence) [KLLM12]. Les algorithmes FASTTWOSUM et TWOSUM produisent bien entendu le même résultat.

1. **fonction** TWOSUM($a,\ b$)
2. $\quad x \leftarrow RN(a+b)$
3. $\quad z \leftarrow RN(x-a)$
4. $\quad y \leftarrow RN((a-(x-z))+(b-z))$
5. $\quad$ **retourner** $[x,\ y]$
6. **fin fonction**

Algorithme 2.2 – TWOSUM [KNUTH, 1969].

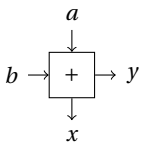


FIGURE 2.2 – Symbole de l'algorithme 2.2.

2.2.2 La multiplication

L'algorithme 2.3 dû à T. DEKKER en 1971, calcul le produit de deux nombres flottants.

1. **fonction** TWOPRODUCT($a,\ b$)
2. $\quad x \leftarrow RN(a \times b)$
3. $\quad [a_H,\ a_L] = \text{SPLIT}(a)$
4. $\quad [b_H,\ b_L] = \text{SPLIT}(b)$
5. $\quad y \leftarrow RN(a_L \times b_L - (((x - a_H \times b_H) - a_L \times b_H) - a_H \times b_L))$
6. $\quad$ **retourner** $[x,\ y]$
7. **fin fonction**

Algorithme 2.3 – TWOPRODUCT [DEKKER, 1971].

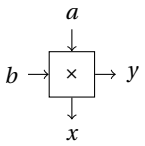


FIGURE 2.3 – Symbole de l'algorithme 2.3.

1. **fonction** SPLIT(a)
2. $\quad c \leftarrow RN(f \times a)$ $\quad \triangleright\ f = 2^{\lceil p/2 \rceil} + 1$
3. $\quad a_H \leftarrow RN(c-(c-a))$
4. $\quad a_L \leftarrow RN(a-a_H)$
5. $\quad$ **retourner** $[a_H,\ a_L]$
6. **fin fonction**

Algorithme 2.4 – SPLIT [VELTKAMP, 1968].

Cet algorithme fait appel à SPLIT (voir algorithme 2.4), une procédure permettant de couper un nombre flottant en deux [Vel68, Vel69]. Soit $\hat{a} = \hat{a}_H + \hat{a}_L$ où $\hat{a}_H$ et $\hat{a}_L$ ne se chevauchent pas et où $|\hat{a}_L| < |\hat{a}_H|$ ($\hat{a}_H$ contient les $p/2$ premiers bits de la mantisse de $\hat{a}$ et $\hat{a}_L$ les $p/2$ derniers). SPLIT nécessitant 4 opérations flottantes, TWOPRODUCT à un coût de 17 opérations (10 produits et 7 soustractions).

Si l'on dispose d'un FMA, l'algorithme 2.5 permet d'effectuer le produit de deux nombres flottants en seulement deux opérations (1 produit et 1 FMA). Le FMA (*Fused Multiply and Add*), est une instruction machine permettant d'effectuer une multiplication suivie d'une addition, le tout arrondi une seule fois (ajoutée à la norme IEEE en 2008).

1. **fonction** TWOPRODUCTFMA(a, b)
2. $\quad x \leftarrow RN(a \times b)$
3. $\quad y \leftarrow RN(\text{FMA}(a, b, -x))$
4. $\quad$ **retourner** $[x, y]$
5. **fin fonction**

Algorithme 2.5 – TWOPRODUCTFMA.

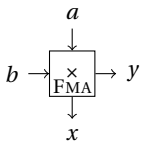


FIGURE 2.4 – Symbole de l'algorithme 2.5.

2.2.3 La division et la racine carrée

La division et la racine carrée disposent elles aussi de transformations sans erreur. M. PICHAT et J. VIGNES proposent en 1993 une transformation pour la division [PV93]. Nous ne détaillerons pas la transformation pour la racine carrée, notez que P. MARKSTEIN propose une solution dans [Mar90].

Soit $\hat{a}$ et $\hat{b}$ deux nombres flottants, M. PICHAT et J. VIGNES proposent un algorithme de transformation sans erreur qui fournit en résultat deux nombres flottants, un quotient $\hat{q}$ et un reste $\hat{r}$:

$$\hat{q} = RN(\hat{a} \div \hat{b}) \text{ et } \hat{r} \text{ tel que } \hat{a} = \hat{b}\hat{q} + \hat{r}.$$

L'algorithme 2.6 calcule cette transformation avec un coût de 20 opérations (1 division, 9 soustractions et 10 produits). Si l'on dispose d'un FMA, l'appel à TWOPRODUCTFMA réduit le coût à 5 opérations (une division, un produit, un FMA et 2 soustractions).

1. **fonction** DIVREM(a, b)
2. $q \leftarrow RN(a \div b)$
3. $[x, y] =$ TWOPRODUCT(q, b)
4. $r \leftarrow RN((a - x) - y)$
5. **retourner** $[q, r]$
6. **fin fonction**

Algorithme 2.6 – DIVREM [PICHAT et VIGNES, 1993].

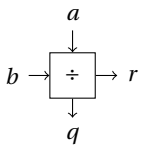


FIGURE 2.5 – Symbole de l'algorithme 2.6.

2.2.4 Utilisation

Les transformations sans erreur présentées dans cette section peuvent être utilisées de deux façons différentes :

1. Au niveau de la seule opération concernée, ces transformations permettent le calcul (en logiciel), sur deux fois plus de chiffres [Dek71].
2. Avec plusieurs opérations combinées, les erreurs d'arrondis s'accumulent. Les transformations sans erreur vont être utilisées pour *compenser* ces erreurs et obtenir un résultat global de meilleure qualité. Voir les sections 2.3 et 2.4.

La table 2.1 récapitule les principales caractéristiques de ces transformations. Pour chaque sortie, la table donne le nombre d'opérations flottantes, ainsi que la latence en nombre d'opérations flottantes sur machine idéale (voir définition 3.2.1) nécessaires à son calcul.

Algorithme	*flop*		Latence	
	x	y	x	y
FASTTWOSUM	1	3	1	3
TWOSUM		6		5
TWOPRODUCT		17		8
TWOPRODUCTFMA		2		2

Algorithme	*flop*		Latence	
	q	r	q	r
DIVREM	1	20	1	10
DIVREMFMA		5		4

TABLE 2.1 – Latence idéale et nombre d'opérations flottantes (*flop*) des transformations sans erreur.

2.3 Les compensations

Les algorithmes compensés permettent de faire des calculs avec K fois plus de précision que disponible. Nous nous intéressons ici aux algorithmes compensés qui calculent avec

deux fois la précision disponible. Ces algorithmes nécessitent l'intervention de spécialistes pour voir le jour. En effet, les compensations ne s'appliquent pas de façon automatique, il faut développer les algorithmes « à la main » et fournir la preuve que le résultat calculé, l'a été avec deux fois plus de précision avant d'être arrondi dans la précision courante. Le principe de tels algorithmes est d'accumuler les erreurs commises lors de chaque opération flottante tout le long du déroulement des calculs. Ce terme d'erreur est enfin appliqué en correction du résultat final car il représente l'accumulation de toutes les erreurs effectuées lors des calculs avant d'être appliqué au résultat final. Le résultat ainsi *compensé* n'est plus affecté par ces erreurs.

Par exemple, prenons l'algorithme 2.7, effectuant la somme de n nombres flottants. Cet algorithme a un coût de $(n-1)$ opérations flottantes.

1. **fonction** SUM(a_1, a_2, …, a_n)
2. **pour** $i = 2 : n$ **faire**
3. $s \leftarrow RN(s + a_i)$
4. **fin pour**
5. **retourner** s
6. **fin fonction**

Algorithme 2.7 – SUM.

S. RUMP, T. OGITA et S. OISHI ont proposé en 2005 une version compensée de cet algorithme utilisant les transformations sans erreur [ROO05] (voir algorithme 2.8). L'algorithme a un coût de $7(n-1)+1$ opérations flottantes.

1. **fonction** SUM2(a_1, a_2, …, a_n)
2. $s \leftarrow a_1$
3. $e \leftarrow 0$
4. **pour** $i = 2 : n$ **faire**
5. $[s, \epsilon] =$ TWOSUM(s, a_i)
6. $e \leftarrow RN(e + \epsilon)$
7. **fin pour**
8. **retourner** $RN(s + e)$
9. **fin fonction**

Algorithme 2.8 – SUM2 [OGITA, RUMP et OISHI, 2005].

De nouveaux algorithmes compensés sont publiés régulièrement. Les auteurs de SUM2 proposent par exemple, dans le même article, un algorithme compensé pour le produit scalaire. Récemment un effort à été porté sur les algorithmes d'évaluation polynomiale :

- en 2007, pour l'algorithme de HORNER [GLL09] ;
- en 2010, pour l'algorithme de DECASTELJAU (polynômes de Bernstein) et de sa dérivée DECASTELJAUDER [JLCS10] ;
- en 2011, pour l'algorithme de CLENSHAW (polynômes de Tchebychev) [JBL+11] ;
- et en 2013 pour l'algorithme de la dérivée de Horner, HORNERDER [JGH+13].

2.4 Les expansions

Les expansions permettent de faire du calcul multi-précision en représentant les nombres par des n-uplets de nombres flottants. Initialement proposées par T. DEKKER en 1971, elles ont ensuite été étudiées par D. PRIEST en 1991. J. SHEWCHUK en 1997 [She97] propose une version épurée du travail de son prédécesseur avec des algorithmes qui utilisent cette fois les propriétés de la norme IEEE 754. Il définit les opérations entre expansions pour la somme et le produit. Plus récemment, M. DAUMAS propose en 1999 une meilleure solution pour la multiplication de deux expansions [Dau99]. C. FINOT-MOREAU introduit le concept des pseudo-expansions en 2001 [FM01]. Son travail complète les travaux précédents avec notamment la division de deux expansions.

Définition 2.4.1 (Expansion)**.** *Une expansion est une représentation d'un grand nombre X par un n-uplet $(x_1, x_2, \ldots, x_n)$ où :*

- *X est égal à la somme exacte, non arrondie, des x_i, $1 \leq i \leq n$;*
- *les x_i non nuls sont triés par ordre décroissant de valeur absolue ;*
- *et les x_i non nuls ne se recouvrent pas, x_i et x_{i+1} ne doivent pas avoir de bits significatifs de même amplitude (i.e. $|x_{i+1}| \leq \frac{1}{2} ulp(x_i)$).*

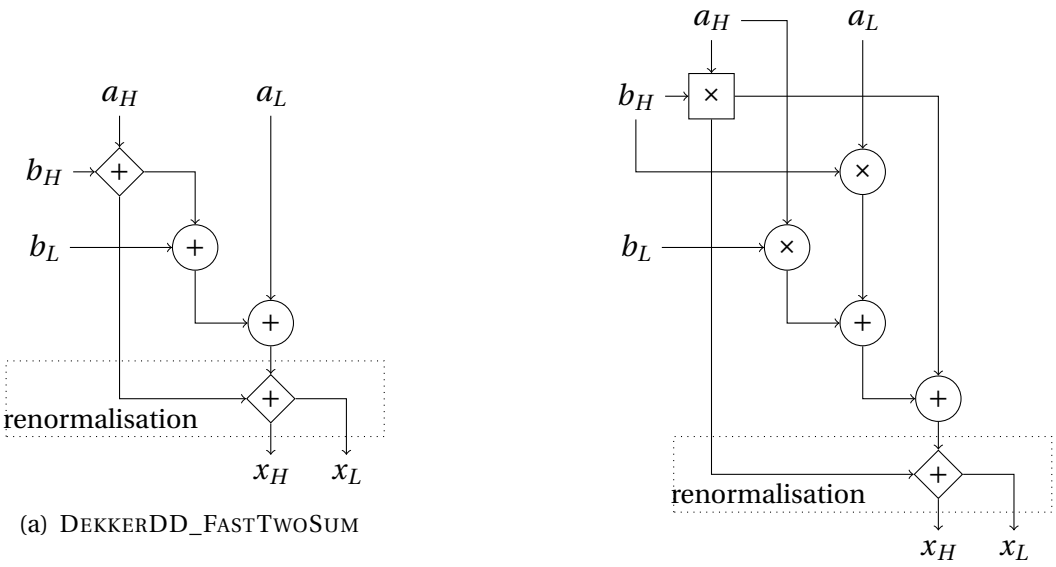
Les x_i sont appelés les composantes de l'expansion. La longueur d'une expansion est définie par le nombre de ses composantes.

Le nombre de composantes d'une expansion croît de façon à représenter exactement le résultat des opérations. Si aucune contrainte n'est appliquée sur la taille des expansions, les résultats finaux des calculs sont exacts. Cependant, des expansions longues sont coûteuses en calcul. Fixer la taille de ces expansions à quelques unités permet en pratique d'obtenir des bons résultats. Dans cette section, nous nous intéressons à un type très particulier d'expansions, les *double-double*.

2.4.1 Le double-double

Le *double-double* est une expansion de taille fixée à deux composantes. Soit X une expansion représentant le 2-uplet (x_H, x_L) vérifiant la définition 2.4.1. Les algorithmes *double-double* effectuent les opérations avec une précision doublée (Des preuves sont clairement

détaillées dans [Lau05]). Les algorithmes *double-double* nécessitent une phase de renormalisation pour assurer que $|x_L| \leq \frac{1}{2} ulp(x_H)$. Cette phase est assurée par un appel à l'algorithme FASTTWOSUM et est représentée par une boîte en pointillés dans les figures 2.6 et 2.7. Les algorithmes 2.9 et 2.10 dus à T. DEKKER [Dek71] calculent respectivement la somme et le produit de deux nombres *double-double*.



(a) DEKKERDD_FASTTWOSUM

(b) DEKKERDD_TWOPRODUCT

FIGURE 2.6 – Représentation des algorithmes *double-double* de [DEKKER, 1971].

Les algorithmes 2.11 et 2.12 attribués à K. BRIGGS et W. KAHAN par D. BAILEY et al. dans la librairie QDLIB [HLB01] calculent la somme de deux nombres *double-double* plus précisément que ceux de T. DEKKER.

L'algorithme 2.13 permet d'effectuer la division de deux nombres *double-double*. La division est basée sur la méthode de NEWTON et est implémentée dans la boîte à outil *Scilab* QUPAT [SIH10].

La table 2.2 récapitule les principales caractéristiques des algorithmes *double-double*. Pour chaque sortie, la table donne le nombre d'opérations flottantes, ainsi que la latence idéale (toujours en nombre d'opérations flottantes) nécessaires à son calcul.

Les algorithmes utilisant les *double-double*, représentent tous les résultats intermédiaires d'un calcul avec deux nombres flottants. De ce fait, les calculs se font sans qu'aucun terme d'erreur soit utilisé pour les accumuler : les résultats sont constamment représentés avec deux fois plus de précision.

fonction DEKKERDD_FASTTWOSUM(a_H, a_L, b_H, b_L)
  si $|a_H| \geq |b_H|$ **alors**
    $[r_H, r_L] =$ FASTTWOSUM(a_H, b_H)
    $r_L \leftarrow RN((r_L + b_L) + a_L)$
  sinon
    $[r_H, r_L] =$ FASTTWOSUM(b_H, a_H)
    $r_L \leftarrow RN((r_L + a_L) + b_L)$
  fin si
  retourner $[x_H, x_L] =$ FASTTWOSUM(r_H, r_L)
fin fonction

Algorithme 2.9 – DEKKERDD_FASTTWOSUM [DEKKER, 1971].

fonction DEKKERDD_TWOPRODUCT(a_H, a_L, b_H, b_L)
  $[r_H, r_L] =$ TWOPRODUCT(a_H, b_H)
  $r_L \leftarrow RN(r_L + ((x_H \times y_L) + (x_L \times y_H)))$
  retourner $[x_H, x_L] =$ FASTTWOSUM(r_H, r_L)
fin fonction

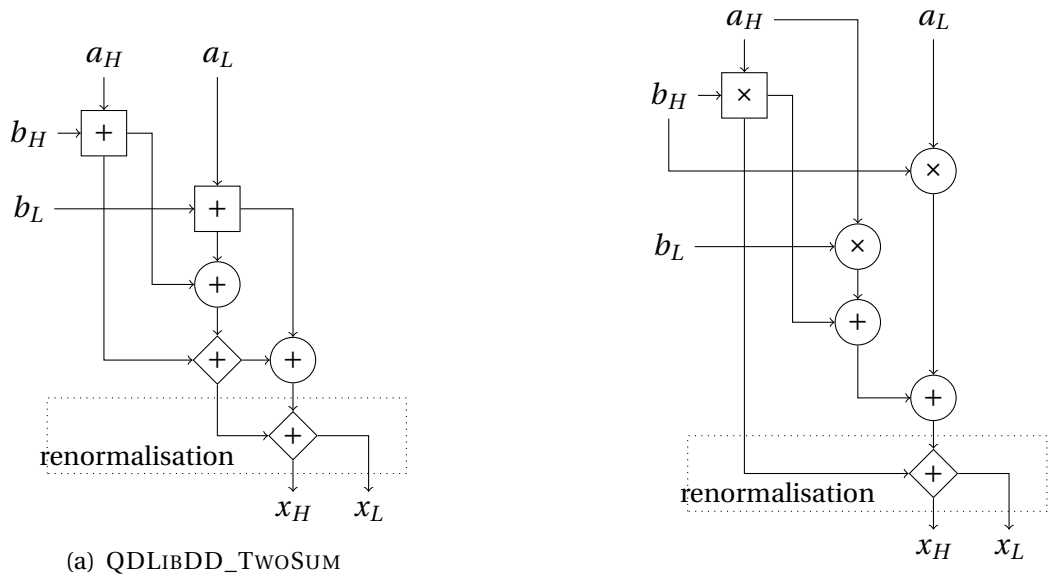
Algorithme 2.10 – DEKKERDD_TWOPRODUCT [DEKKER, 1971].

fonction QDLIBDD_TWOSUM(a_H, a_L, b_H, b_L)
  $[r_H, r_L] =$ TWOSUM(a_H, b_H)
  $[s_H, s_L] =$ TWOSUM(a_L, b_L)
  $c \leftarrow RN(r_L + s_H)$
  $[u_H, u_L] =$ FASTTWOSUM(r_H, c)
  $w \leftarrow RN(s_L + u_L)$
  retourner $[x_H, x_L] =$ FASTTWOSUM(u_H, w)
fin fonction

Algorithme 2.11 – QDLIBDD_TWOSUM [QD LIBRARY, 2000].

fonction QDLIBDD_TWOPRODUCT(a_H, a_L, b_H, b_L)
  $[r_H, r_L] =$ TWOPRODUCT(a_H, b_H)
  $r_L \leftarrow RN(r_L + (a_H \times b_L))$
  $r_L \leftarrow RN(r_L + (a_L \times b_H))$
  retourner $[x_H, x_L] =$ FASTTWOSUM(r_H, r_L)
fin fonction

Algorithme 2.12 – QDLIBDD_TWOPRODUCT [QD LIBRARY, 2000].



(a) QDLIBDD_TWOSUM

(b) QDLIBDD_TWOPRODUCT

FIGURE 2.7 – Représentation des algorithmes *double-double* de [QD LIBRARY, 2000].

1. **fonction** QUPAT_DIVISION(a_H, a_L, b_H, b_L)
2. $c \leftarrow RN(a_H \div b_H)$
3. $[p,\ e]$ = TWOPRODUCT(c, b_H)
4. $cc \leftarrow RN((a_H - p - e + a_L - c \times b_L)/b_H)$
5. **retourner** $[x_H,\ x_L]$ = FASTTWOSUM(c, cc)
6. **fin fonction**

Algorithme 2.13 – QUPAT_DIVISION [QUPAT TOOLBOX, 2010].

Algorithme	*flop*		Latence	
	x_H	x_L	x_H	x_L
DEKKERDD_FASTTWOSUM	6 (+1 test)	8 (+1 test)	6	8
DEKKERDD_TWOPRODUCT	22	24	10	12
QDLIBDD_TWOSUM	18	20	11	13
QDLIBDD_TWOPRODUCT	22	24	11	13
QUPAT_DIVISION	25	27	15	17

TABLE 2.2 – Latence idéale et nombre d'opérations flottantes (*flop*) des algorithmes *double-double*.

Les algorithmes *double-double* peuvent être appliqués de façon automatique, par simple surcharge des opérations élémentaires. Reprenons l'exemple de l'algorithme de

sommation 2.7. L'algorithme 2.14 présente une version de l'algorithme SUM en *double-double*. Puisque a_i n'est pas un *double-double*, SUMDD a un coût de $10(n-1)$ opérations flottantes (en effet on peut faire l'économie d'un TWOSUM, FASTTWOSUM et d'une addition à chaque appel à QDLIBDD_TWOSUM).

1. **fonction** SUMDD(a_1, a_2, …, a_n)
2. $s_H \leftarrow a_1$
3. $s_L \leftarrow 0$
4. **pour** $i = 2 : n$ **faire**
5. $[s_H, s_L] =$ QDLIBDD_TWOSUM(s_H, s_L, a_i, $\emptyset$)
6. **fin pour**
7. **retourner** s_H
8. **fin fonction**

Algorithme 2.14 – SUMDD.

2.5 Compensation versus double-double

Dans les sections précédentes nous avons abordé deux méthodes qui permettent d'effectuer des calculs avec le double de précision disponible. Il y a toutefois deux axes pour lesquels ces méthodes sont fondamentalement différentes :

1. Le niveau de connaissances requis en analyse numérique pour développer des algorithmes compensés est beaucoup plus important que celui nécessaire à l'utilisation des *double-double*. En effet, l'application de ces derniers peut se faire par une simple surcharge des opérations élémentaires dans un programme effectuant des calculs flottants.
2. Cependant, les algorithmes compensés ont l'avantage (en pratique), d'être de meilleurs candidats au parallélisme d'instruction [Lou07, LL07] et offrent donc de meilleures performances. Par exemple, les auteurs constatent en pratique que pour l'algorithme d'évaluation polynomiale d'HORNER que COMPHORNER (la version compensée) s'exécute au moins deux fois plus rapidement que DDHORNER (la version en *double-double*). Ceci est dû à la phase de renormalisation présente dans les algorithmes *double-double* que les algorithmes compensés ne font pas. La renormalisation s'avère être pénalisante pour les performances car elle linéarise les calculs et réduit alors le parallélisme d'instruction exploitable par le processeur.

En résumé, on dispose de deux techniques qui offrent au final la même précision. Néanmoins, l'une, « automatique », facile à mettre en œuvre et demandant peut de

connaissances dans le domaine. L'autre est bien plus performante mais doit être appliquée « manuellement » pour chaque problème distinct par des experts en la matière. Enfin, nous conclurons ce chapitre par une comparaison entre ces deux techniques dans le cas de l'algorithme de sommation 2.7.

2.5.1 Comparaison entre SUM2 et SUMDD

Les codes 2.1 et 2.2 implémentent respectivement les algorithmes de somme SUM2 et SUMDD en langage C. Chaque itération de la boucle à un coût de 7 opérations flottantes pour SUM2 et de 10 opérations pour SUMDD.

Code 2.1 : SUM2. ◁

```
Ligne 1  s = a[0];
       -  e = 0;
       -  for(i=1; i<n; i++)
       -    {
       5      /* [s,y] = TwoSum(s,a[i]) */
       -      t = s;
       -      s += a[i];
       -      z = s - t;
       -      y = (t - (s - z)) + (a[i] - z);
      10
       -      e += y;
       -    }
       -  s += e;
```

Code 2.2 : SUMDD. ◁

```
sh = a[0];                                  Ligne 1
sl = 0;                                          -
for(i=1; i<n; i++)                               -
  {                                              -
    /* [x,y] = TwoSum(sh,a[i]) */                5
    x = sh + a[i];                               -
    z = x - sh;                                  -
    y = (sh - (x - z)) + (a[i] - z);             -
                                                 -
    t = sl + y;                                 10
                                                 -
    /* [sh,sl] = FastTwoSum(x,t) */              -
    sh = x + t;                                  -
    sl = (x - sh) + t;                           -
  }                                             15
```

Les deux codes donnent des résultats similaires, mais leurs comportements sont fondamentalement différents. L'algorithme SUM2 calcule pour chaque addition, l'erreur qui est engendrée (lignes [8-9]) puis l'accumule au fur et à mesure du déroulement de la boucle (ligne 11) pour enfin compenser le résultat final avec ce terme d'erreur ainsi accumulé. La compensation se fait à ligne 13. L'algorithme SUMDD, quant à lui, calcule l'erreur engendrée lors de chaque addition (lignes [7-9]), mais n'accumule pas ensuite cette erreur. L'erreur est directement réinjectée dans les résultats des additions à chaque tour de boucle à l'aide notamment du FASTTWOSUM (lignes [13-14]). En effet, les résultats sont représentés sur deux nombres flottants pour chaque calculs intermédiaires du déroulement de l'algorithme.

Il est facile de voir que les algorithmes SUM2 et SUMDD introduisent 7 et 10 fois plus d'opérations flottantes que l'algorithme original SUM. On pourrait donc s'attendre à un

temps d'exécution relativement proche pour les algorithmes SUM2 et SUMDD, avec néanmoins de meilleures performances pour l'algorithme compensé (3 opérations flottantes de moins par itération de boucle). Cependant, comme soulevé par LANGLOIS et LOUVET [LL07], on constate en pratique une nette différence entre les deux algorithmes, la version compensée étant entre trois et quatre fois plus rapide que l'algorithme *double-double*. La table 2.3 donne les temps d'exécution[3] de ces algorithmes pour des sommes de n termes. Ces temps ont été obtenus à l'aide de la bibliothèque Unix `time` (moyenne sur 100 000 exécutions).

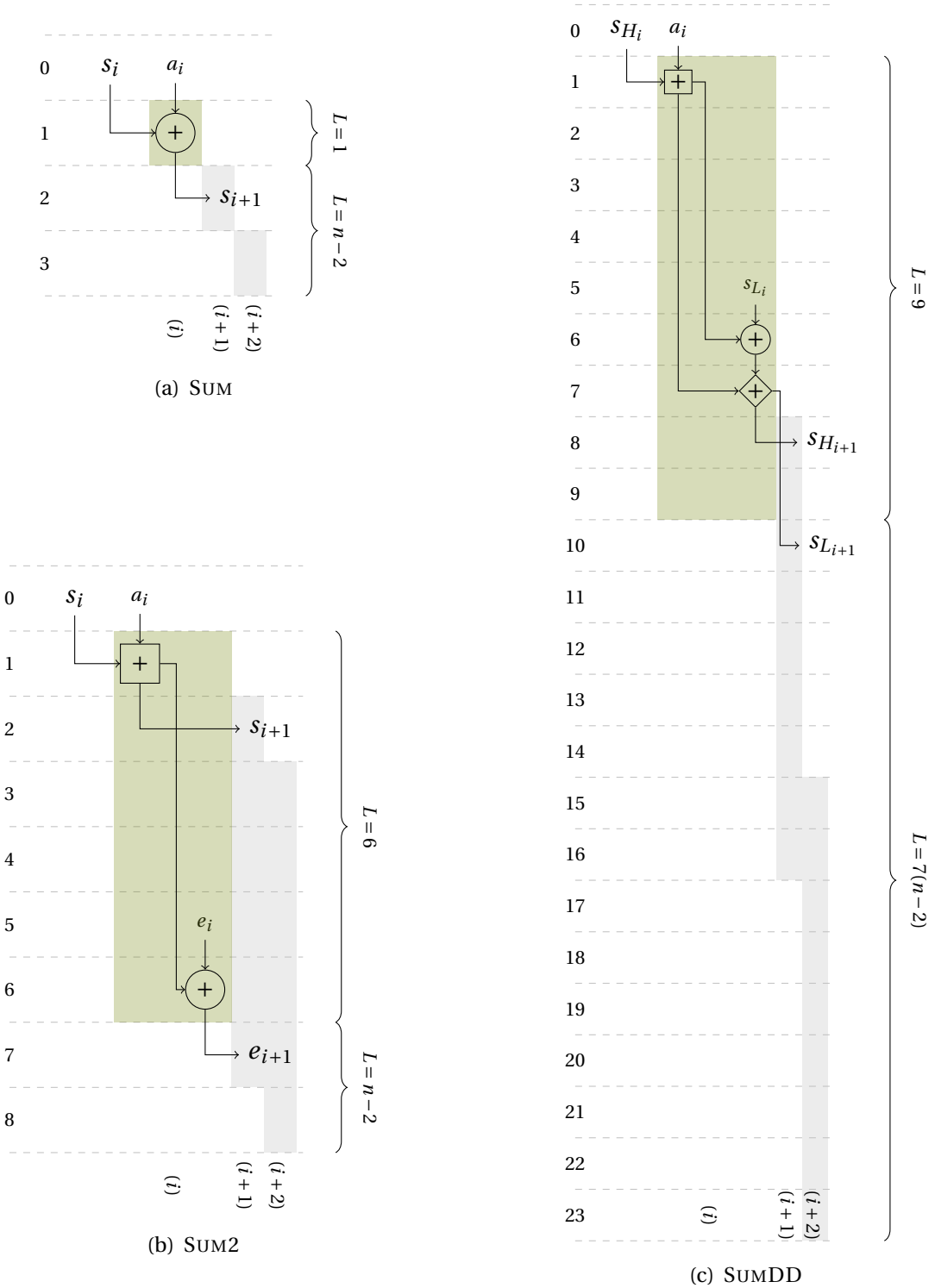
n	10^2	10^3	10^4	10^5	10^6	*flop*
SUM	0,1	0,9	9,8	98	1251	$n-1$
SUM2	0,2	2,3	23	243	2579	$7(n-1)+1$
SUMDD	0,8	7,9	78	789	8174	$10(n-1)$
SUMDD/SUM2	4	3,43	3,39	3,25	3,17	$\approx 1,43$

TABLE 2.3 – Temps d'exécution approximatifs en μs et nombres d'instruction flottantes théorique des algorithmes SUM2 et SUMDD.

Les différences de performances entre ces deux algorithmes s'expliquent par le fait que la version compensée est beaucoup plus parallélisable que la version *double-double*. La figure 2.8 illustre ce phénomène en comparant les latences idéales (c'est-à-dire des latences obtenues sur une machine idéale vérifiant la définition 3.2.1) des deux algorithmes. On constate que l'algorithme SUM2 est plus parallélisable car il est possible d'exécuter l'itération $(i+1)$ un cycle après le début de l'itération i. Alors que pour l'algorithme SUMDD, l'itération $(i+1)$ ne peut être exécutée qu'après le $7^{\text{ème}}$ cycle de l'itération (i). Sur une machine idéale il est alors possible d'exécuter l'algorithme SUM2 en $7+(n-2)$ cycles. Comme montré à la figure 2.8(b), la première itération coûte six cycles et les $(n-2)$ suivantes coûtent seulement un cycle puisque les cinq premiers peuvent se faire en parallèle à la première itération (sans oublier la dernière opération en sortant de la boucle). La figure 2.8(c) montre que le coût de la première itération de SUMDD est de neuf cycles. Les itérations suivantes coûtent chacune sept cycles car seulement deux cycles peuvent s'exécuter en parallèle entre deux itérations de la boucle. Ce qui donne une latence de $9+7(n-2)$ cycles pour l'algorithme SUMDD. Soit une latence approximativement sept fois supérieure à SUM2 sur machine idéale.

Bien entendu, les mesures *in situ*, sont loin d'atteindre les performances idéales, et ce pour de nombreuses raisons. La principale étant qu'en matériel, le parallélisme d'instruction est limité. Le processeur ne peut donc qu'exécuter un petit nombre d'instruction au

[3] Exécutables obtenus dans l'environnement A.1 décrit dans l'annexe A.



(a) SUM

(b) SUM2

(c) SUMDD

FIGURE 2.8 – Latence idéale (notée L) des algorithmes de sommation SUM, SUM2 et SUMDD.

même instant. Cependant, le processeur réussit à exploiter ce parallélisme latent au mieux de ces capacités, d'où les performances accrues des algorithmes compensés par rapport aux algorithmes *double-double*.

Les mesures obtenues ici, l'on été avec une simple bibliothèque disponible par défaut sous UNIX. Nous verrons qu'il se pose alors un nouveau problème, qui est celui de la mesure des performances. Nous verrons dans le chapitre 3 que ce problème s'avère être difficile et qu'aucune réponse entièrement satisfaisante n'existe encore.

CHAPITRE 3

MANIPULATION DE PROGRAMMES ET MESURE DES PERFORMANCES

Préambule

La connaissance se fonde sur l'accumulation d'observations et de faits mesurables. Or, on constate la non fiabilité des mesures lorsque l'on évalue les performances d'un algorithme sur un ordinateur. Ce chapitre présente un aperçu des facteurs inhérents à la non fiabilité des mesures ainsi que des outils PAPI *et* PERPI *permettant de s'en affranchir. Une courte introduction à la manipulation de programme permettra de mieux en appréhender le fonctionnement.*

3.1 Introduction à la transformation de code

TOUT algorithme implémenté au sein d'un programme passe par au moins une phase de compilation avant de prendre la forme d'un fichier exécutable compréhensible par une machine. Cette phase détermine nombre de caractéristiques du code exécutable (vitesse d'exécution, occupation mémoire, etc.). C'est pourquoi la compilation est un domaine de recherche très riche et les compilateurs tendent à générer des exécutables toujours plus performants. Cette introduction présente brièvement ce qu'est un compilateur et aborde un autre domaine de transformation de code : la synthèse.

3.1.1 Compilation

Un compilateur [Hop69, App98, ALSU06] est un outil informatique qui transforme un code, *le langage source* en un autre langage : *le langage cible*. Le langage source est écrit dans un langage de programmation de haut niveau d'abstraction facilement compréhensible pour l'humain (tel le C), pour ensuite être généralement transformé en un langage cible de plus bas niveau directement exploitable par la machine (le code objet). Il existe de plus des outils de transformation *source à source* qui transforment du code écrit dans un langage de haut niveau, dans le même (ou un autre) langage de haut niveau. L'intérêt de ces outils est de transformer automatiquement des programmes écrits dans des langages de haut niveau afin d'y appliquer des optimisations qu'il serait fastidieux d'effectuer manuellement. Ils sont de plus indépendants de l'architecture et moins lourds à écrire qu'un compilateur.

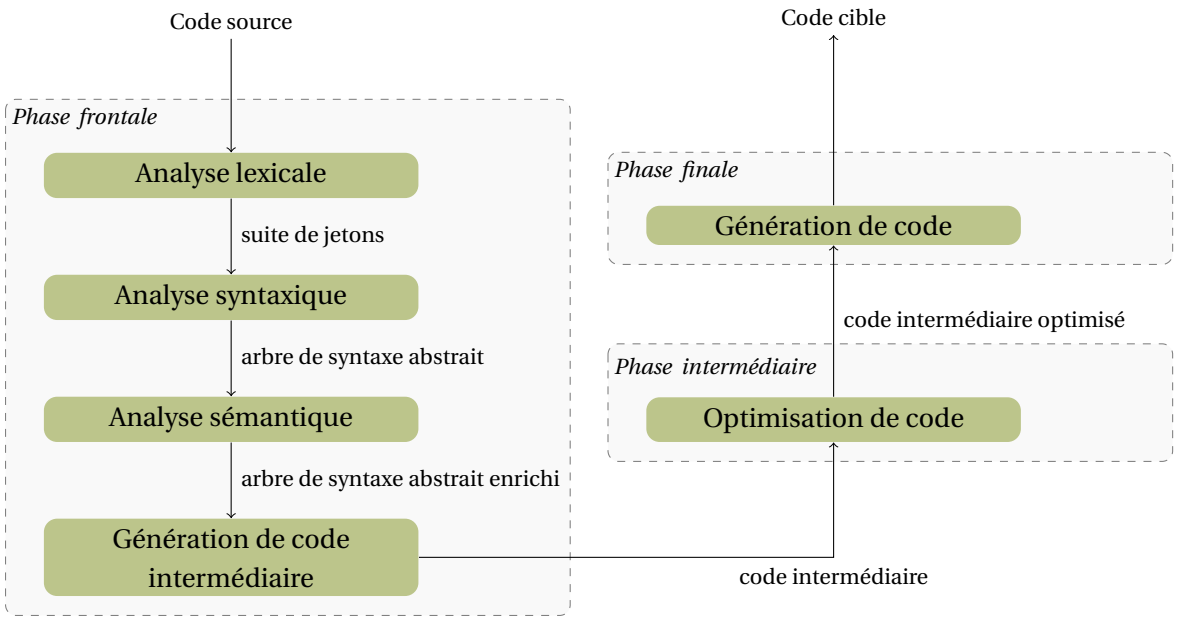


FIGURE 3.1 – Structure d'un compilateur (chaîne de compilation classique).

La figure 3.1 décrit les principales étapes d'une chaîne de compilation. La figure exhibe trois phases indépendantes et interchangeables.

- La *phase frontale*, transforme le programme source en une représentation intermédiaire. Elle peut par exemple le transformer en du code au format trois-adresses ou sous la forme *Static Single Assignment (*SSA*)*.
- La *phase finale*, transforme une représentation intermédiaire en un langage cible de haut ou bas niveau.

- La *phase intermédiaire*, applique une suite d'optimisations à partir de la représentation intermédiaire.

Un compilateur peut posséder plusieurs phases frontales et finales. En effet, il dispose d'une phase frontale ou finale distincte pour chaque langage source ou cible supportés. On peut ainsi écrire des compilateurs pour toute une gamme de langages et d'architectures en partageant la partie intermédiaire, à laquelle on attache une partie frontale par source et une partie finale par cible. Les étapes de la compilation incluent :

- L'*analyse lexicale*, découpe le code source en petits morceaux appelés *jetons*. Chaque jeton est une unité atomique unique du langage, par exemple un mot-clé, un identifiant ou un symbole.
- L'*analyse syntaxique*, implique l'analyse de la séquence de jetons pour identifier la structure syntaxique du programme [GJ08]. Cette phase s'appuie généralement sur la construction d'un arbre d'analyse. On remplace la séquence linéaire des jetons par une structure en arbre construite selon la grammaire formelle qui définit la syntaxe du langage (voir figure 3.2).

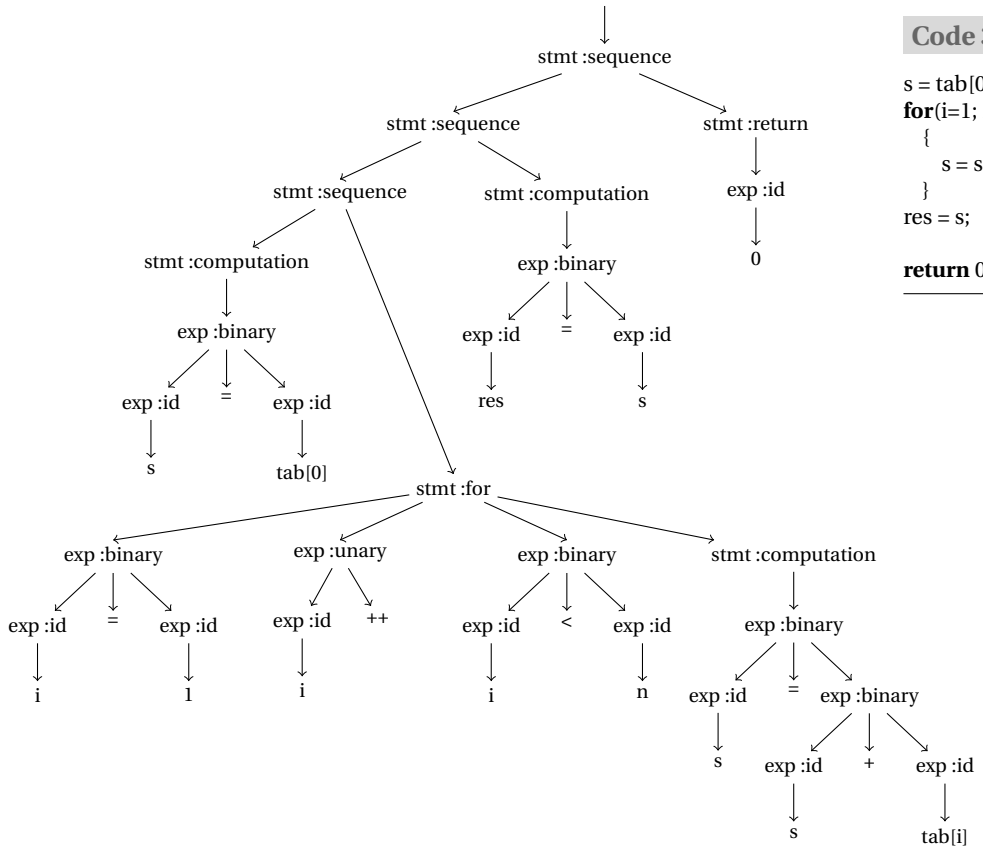
```
statement := sequence of statement * statement
           | for of expression * expression * expression * statement
           | return of expression
           | computation of expression
expression := binary of expression * operator * expression
            | unary of operator * expression
            | id
operator := '+' | '=' | '<' | '++'
```

FIGURE 3.2 – Exemple de grammaire formelle (suffisante pour définir la syntaxe du code 3.1).

- L'*analyse sémantique*, est la phase durant la laquelle le compilateur ajoute des informations sémantiques à l'arbre d'analyse syntaxique (type, portée, ...). Cette phase peut émettre des avertissements et rejeter des programmes incorrects.
- La *transformation du code source en code intermédiaire* [ALSU06]. La représentation intermédiaire se traduit la plupart du temps sous la forme d'arbres dérivés de l'arbre en sortie de l'analyse sémantique. Un exemple de représentation intermédiaire est proposé à la figure 3.3. La figure illustre l'arbre de représentation intermédiaire du code 3.1.
- L'application des techniques d'*optimisation* sur le code intermédiaire. L'optimisation consiste a rendre un programme informatique plus performant selon un ou des critères définis selon l'usage que l'on veut faire du programme. Un critère de performance communément rencontré est le temps d'exécution. Il est au cœur des recherches en optimisation [RG81, BGS94, Sar98, DRV00] depuis de nombreuses an-

nées via, par exemple, l'optimisation des boucles, l'ordonnancement des instructions, l'utilisation des registres et des caches, etc. Néanmoins, d'autres critères d'optimisation existent : la consommation énergétique, le temps de compilation, l'occupation mémoire ou encore la précision numérique.

- La *génération de code*, transforme la représentation intermédiaire dans le langage cible.



Code 3.1 : SUM. ◁

```
s = tab[0];
for(i=1; i<n; i++)
  {
    s = s + tab[i];
  }
res = s;

return 0;
```

FIGURE 3.3 – Exemple de représentation intermédiaire du code 3.1 construite à partir de l'arbre de syntaxe abstrait enrichi en sortie de la phase frontale (`stmt` et `exp` sont les abréviations respectives de *statement* et *expression*).

3.1.2 Synthèse de code

La synthèse de code [Gul10, SGF10] consiste à produire un code à partir des attentes d'un utilisateur. Elles sont exprimées sous la forme de spécifications. À l'inverse des compilateurs qui prennent en entrée des codes de haut niveau et réalisent des traductions vers

du code bas niveau, les synthétiseurs acceptent une variété de spécifications et réalisent des recherches sur un ensemble de codes pour déterminer celui qui répond au mieux aux contraintes imposées par l'utilisateur.

Un synthétiseur est caractérisé par trois aspects : (i) le type de spécifications acceptées exprimant les attentes de l'utilisateur, (ii) l'espace de recherche sur lequel opérer, et (iii) les techniques de recherches employées.

(i) Les spécifications de l'utilisateur peuvent être exprimées sous la forme de relations logiques entre les entrées et les sorties, des exemples d'entrées-sorties, des langages naturels, des codes partiels ou insuffisants et encore des formules (C. MOUILLERON et G. REVY utilisent par exemple des spécifications sous forme de formules dans CGPE [MR11]).

(ii) L'espace de recherche peut porter sur des codes impératifs ou fonctionnels (avec la possibilité de restreindre le champ d'action sur certaines structures de contrôle ou d'un ensemble d'opérations). Des modèles restreints de calculs, tels que des traducteurs d'expressions régulières, des grammaires, ou bien encore des représentations logiques.

(iii) Les techniques de recherche peuvent être basées sur la recherche exhaustive, l'apprentissage par espace de version, l'apprentissage automatique (intelligence artificielle, algorithmes génétiques [Gol89]), l'utilisation d'heuristiques, ou bien par le biais de techniques de raisonnement logique.

3.2 Évaluer les performances d'un algorithme

Évaluer les performances d'un algorithme est une chose ardue. Au chapitre 2, nous avons utilisé deux méthodes différentes afin de caractériser les performances des algorithmes. En premier lieu, il est possible de compter le nombre d'instructions caractéristiques, c'est-à-dire, dans notre cas, le nombre d'instructions flottantes (*flop*) utilisées par l'algorithme. Il est aussi possible de caractériser les performances d'un algorithme en mesurant le temps nécessaire à la résolution de ses opérations.

Comme le confirment les résultats des mesures du tableau 2.3, il existe des nuances importantes entre ces deux moyens de mesures. En effet, on constate que la mesure du nombre d'opérations flottantes indique un ratio de 1,43 entre SUMDD et SUM2, c'est-à-dire, que selon ce critère, SUM2 est 1,43 fois plus rapide que SUMDD. Cependant, les mesures de temps d'exécution exhibent que SUM2 est entre 3 et 4 fois meilleurs que SUMDD. Ces différences s'expliquent par des niveaux de parallélisme différents entre ces deux algorithmes. On constate même que la machine ne parvient pas à tirer entièrement partie du potentiel de parallélisation de SUM2. De plus, nous verrons à la section 3.3 que ces mesures ne sont pas reproductibles. Certains auteurs affirment même qu'une part de hasard

accompagne ces mesures de performances [LPGP12]. Dans son article [Rum09] S. RUMP écrit :

> Measuring the computing time of summation algorithms in a high-level language on today's architectures is more of a hazard than scientific research.

dans son article intitulé : « Ultimately Fast Accurate Summation ». D. BAILEY écrit de plus un article sur les mesures de performances intitulé : « Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers » [BLW11, chapitre 1].

Un moyen fiable d'évaluer les performances consiste à étudier le niveau de parallélisme d'instruction (ILP, voir définition 3.2.2) d'un algorithme sur une machine idéale (voir définition 3.2.1).

Définition 3.2.1 (Machine idéale)**.** *La machine idéale de* HENNESSY *et* PATTERSON *[HP12b] est une machine qui dispose d'une infinité de ressources. La seule contrainte entre les instructions sur une telle machine est la contrainte de lecture après écriture* LAE. *De ce fait, une instruction est lue, un cycle après l'écriture des instructions dont elle dépend.*

Définition 3.2.2 (ILP)**.** *Le niveau de parallélisme d'instruction, de l'anglais Instruction Level Parallelism (*ILP*) décrit le potentiel qu'ont les instructions d'un programme d'être exécutées simultanément.*

*Une mesure utilisée pour caractériser l'*ILP *est l'*IPC *:*

$$\text{IPC}(i,c) = \frac{i}{c}.$$

*L'*IPC *est le nombre moyen d'instructions i pouvant être exécutées à chaque cycle c.*

Le niveau de parallélisme d'instruction révélé par une *exécution idéale* (c'est-à-dire une exécution sur une machine idéale) est une mesure significative du potentiel de performance d'un algorithme. À l'heure actuelle, aucune solution viable n'existe pour effectuer ces *mesures idéales*. Cependant, nous présenterons à la section 3.4 un outil permettant d'effectuer de telles mesures *en situation réelle* utilisant les compteurs matériels présents au sein des processeurs. Néanmoins imparfaite, c'est pour l'heure la méthode que l'on privilégiera dans la suite de nos travaux. Nous présenterons de plus un outil récent permettant d'évaluer les *performances idéales* d'algorithme à la section 3.5.

3.3 La non reproductibilité des mesures

Nous entendons par non reproductibilité des mesures, le fait d'obtenir des résultats différents d'une mesure à l'autre pour une application et un environnement donnés. Reprenons l'exemple du chapitre précédent avec l'algorithme SUM2. La table 3.1 illustre la non

reproductibilité des mesures de performances. Le tableau présente sept mesures du temps d'exécution pour SUM2 avec des résultats différents. On constate que plus on itère le processus de mesure, plus on obtient une valeur fine. Ce qui est d'autant plus nécessaire que l'exécution de SUM2 est très rapide (de l'ordre de quelques micro secondes). Il est donc important de faire des moyennes sur des centaines ou des millions d'exécutions pour obtenir des mesures représentatives et reproductibles.

Mesures	Temps d'exécution en μs
1^{re} (une exécution)	0
2^{e} (moyenne sur 1 000 exécutions)	0
3^{e} (moyenne sur 10 000 exécutions)	3
4^{e} (moyenne sur 10 000 exécutions)	2
5^{e} (moyenne sur 100 000 exécutions)	2,3
6^{e} (moyenne sur 1 000 000 d'exécutions)	2,31
7^{e} (moyenne sur 10 000 000 d'exécutions)	2,308

TABLE 3.1 – Non reproductibilité des mesures : exemple avec l'évaluation du temps d'exécution de l'algorithme SUM2 ($n = 1\,000$, environnement A.1).

Cependant, les variations présentes lors de la mesure de temps d'exécution ne permettent pas de donner de conclusions satisfaisantes sur les performances d'un algorithme. On mesure de plus, des variations entre les différents environnements utilisés. En effet, le compilateur et ses options ou encore la machine utilisée auront des effets sur comment le programme est construit et exécuté. Dans les faits, la non reproductibilité des mesures peut s'expliquer par bien d'autres raisons :

- Les effets du compilateur et de ses options. Les différents compilateurs disponibles et leurs nombreuses options ont des conséquences sur la manière dont le programme est transformé dans le langage de la machine.
- Le rôle de l'architecture. Les limitations imposées par le matériel (nombre de registre, taille du cache, ...) influencent les résultats d'une machine à l'autre. De plus, même sans changer de processeur, le non déterminisme dû au prédicteur de branchement et à l'ordonnancement peut d'une exécution à l'autre modifier les performances d'un algorithme. Pire encore, certains processeurs modernes peuvent réguler leur fréquence de fonctionnement en fonction des conditions externes.
- Les conditions externes. La température de la pièce dans laquelle la machine effectue les calculs peut affecter les résultats. Par exemple, si la chaleur est trop importante, un processeur peut baisser sa fréquence de fonctionnement afin de réduire sa température.

- Le système d'exploitation. Le processeur est une ressource partagée entre de nombreux processus. Ce partage a un effet significatif sur les mesures de performances puisque lors de la mesure d'une tache a, on mesure aussi des morceaux des taches s'étant exécutées (entièrement ou partiellement) en même temps que la tache mesurée.

Les mesures effectuées à partir de compteurs matériels, incorporés au sein même des processeur n'échappent pas à ces effets. Des études récentes mettent en doute leur fiabilité [WD10, WTM13] et montrent que ces mesures sont à interpréter avec réserves.

3.4 Une application de mesure des performances : PAPI

La librairie logicielle PAPI [MBDH99], permet de mesurer les performances d'un programme à l'aide des compteurs matériels présents au sein de l'architecture du processeur. Cette librairie permettre d'écrire du code portable en faisant l'interface entre du code de haut niveau (i.e. le code portable) et les compteurs matériels accessibles par des instructions de bas niveau, spécifiques à chaque processeur. PAPI s'applique en instrumentant le code source avec des appels aux fonctions de sa librairie. Le code 3.2 donne un exemple minimaliste qui permet d'obtenir les nombres de cycles et d'instructions totaux de l'algorithme SUM.

Code 3.2 : Instrumentation de l'algorithme SUM avec les fonctions PAPI pour la mesure du nombre d'instructions et de cycles. ◁

```
Ligne 1  int events[2] = {PAPI_TOT_INS, PAPI_TOT_CYC}; /* nb instr., nb cycles */
  -      long long values[2] = {0};
  -
  -      PAPI_start_counters(events, 2); /* démarrer les compteurs */
  5
  -      s = a[0];
  -      for(int i=1; i<n; i++)
  -        s += a[i];
  -
 10      PAPI_stop_counters(values, 2); /* stopper les compteurs et sauvegarder leur
             valeurs */
  -
  -      printf("Nombre␣d'instructions:␣%lld\n", values[0]);
  -      printf("Nombre␣de␣cycles:␣%lld\n", values[1]);
```

Effectuons maintenant des mesures d'ILP afin de caractériser les performances des algorithmes. Pour se faire nous avons besoin de connaître le nombre de cycles et d'instructions. Comme le montre la table 3.2, les mesures effectuées avec PAPI sont quasiment reproductibles et varient très peu selon les exécutions successives de l'algorithme (la valeur

d'IPC est constante à une valeur près). On constate que le nombre d'instructions mesurées est stable et qu'il n'y a seulement que de légères variations en ce qui concerne le nombre de cycles. De plus, pour les algorithmes étudiés, on observe en pratique que quelques milliers d'exécutions suffisent pour obtenir des mesures reproductibles.

	Environnement A.1			Environnement A.2		
Mesures	Instructions	Cycles	IPC	Instructions	Cycles	IPC
SUM2 (1re)	519	314	1,65	519	422	1,23
SUM2 (2e)	519	348	1,49	519	422	1,23
SUM2 (3e)	519	314	1,65	519	422	1,23
SUMDD (1re)	2 105	2 363	0,89	2 105	3 339	0,63
SUMDD (2e)	2 105	2 366	0,89	2 105	3 336	0,63
SUMDD (3e)	2 105	2 377	0,89	2 105	3 367	0,63

TABLE 3.2 – Non reproductibilité des mesures effectuées avec PAPI (3 mesures avec $n = 100$, moyennes obtenues sur 10 000 exécutions).

Le tableau 3.3 présente les IPC des algorithmes SUM, SUM2 et SUMDD mesurés avec PAPI. On constate de nouveau que les mesures ne sont pas reproductibles d'une machine à l'autre même au sein d'environnements identiques. On remarque néanmoins que SUM2 à un niveau de parallélisme d'instruction toujours plus élevé que celui de SUMDD. Les mesures varient du simple ou double selon la machine utilisée. Ces mesures ne font que confirmer la difficulté de reproduction et d'exploitation permettant de conclure sur les performances réelles des algorithmes.

	Environnement A.1					Environnement A.2				
n	10^2	10^3	10^4	10^5	10^6	10^2	10^3	10^4	10^5	10^6
SUM	1,65	1,66	1,66	1,66	1,26	1,23	1,23	1,25	0,76	0,75
SUM2	2,22	2,28	2,27	2,26	2,05	1,01	1,02	1,07	1,05	1,04
SUMDD	0,89	0,88	0,87	0,87	0,85	0,63	0,63	0,63	0,62	0,62
SUMDD/SUM2	0,40	0,38	0,38	0,38	0,41	0,62	0,61	0,59	0,59	0,59

TABLE 3.3 – Mesures du niveau de parallélisme d'instruction (en IPC) des algorithmes SUM, SUM2 et SUMDD mesurés avec PAPI (moyennes obtenues sur 10 000 exécutions).

Les mesures détaillées du nombre de cycles et d'instructions sont présentées au tableau B.1 de l'annexe B.

3.5 Vers des mesures reproductibles : PERPI

PERPI est un outil développé par B. GOOSSENS et D. PARELLO[1] [GLPP12]. Il permet d'analyser automatiquement l'ILP d'un programme comme s'il était exécuté sur une machine idéale. Basé sur PIN [LCM+05], il est dédié aux programmes x86 et permet d'en mesurer le nombre de cycle et d'instructions. Il fournit de plus des histogrammes d'instructions par cycles et des graphes de dépendances des instructions. Puisque les mesures sont effectuées dans le cadre d'une machine idéale, celles-ci sont indépendantes de la machine réelle, et donc parfaitement reproductibles d'une machine à l'autre (sous réserve d'utiliser les mêmes compilateurs et options de compilation). De plus, une seule exécution suffit pour obtenir les résultats. PERPI s'utilise en ligne de commande (ou via une interface graphique) et opère directement à partir d'un exécutable dont l'exécution est simulé afin de générer sa trace d'exécution (c'est-à-dire, la liste complète des instructions utilisées pour l'exécution sur la machine réelle). Les informations voulues sont ensuite calculées à partir de cette trace.

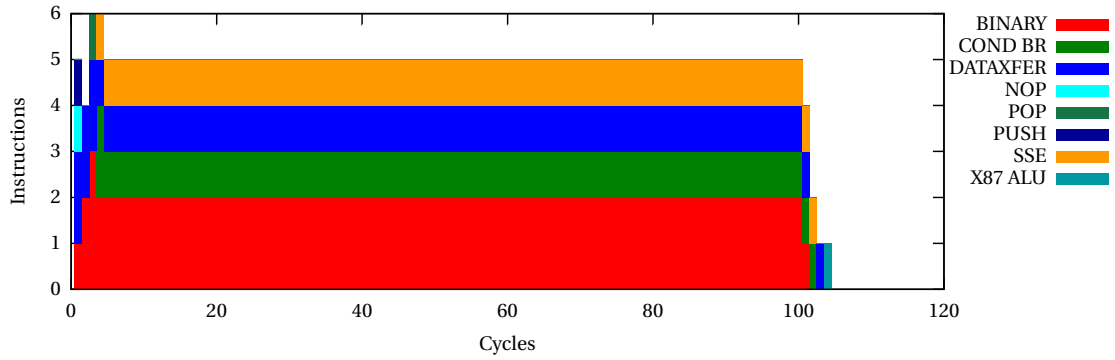
Les quelques mesures présentes à la table 3.4 montrent la reproductibilité des mesures effectuées avec PERPI. Une seule exécution suffit et les mesures restent les mêmes selon l'environnement utilisé.

	Environnement A.1			Environnement A.2		
Mesures	Instructions	Cycles	IPC	Instructions	Cycles	IPC
SUM2 (1re)	1 599	210	7,61	1 599	210	7,61
SUM2 (2e)	1 599	210	7,61	1 599	210	7,61
SUM2 (3e)	1 599	210	7,61	1 599	210	7,61
SUMDD (1re)	2 095	897	2,33	2 095	897	2,33
SUMDD (2e)	2 095	897	2,33	2 095	897	2,33
SUMDD (3e)	2 095	897	2,33	2 095	897	2,33

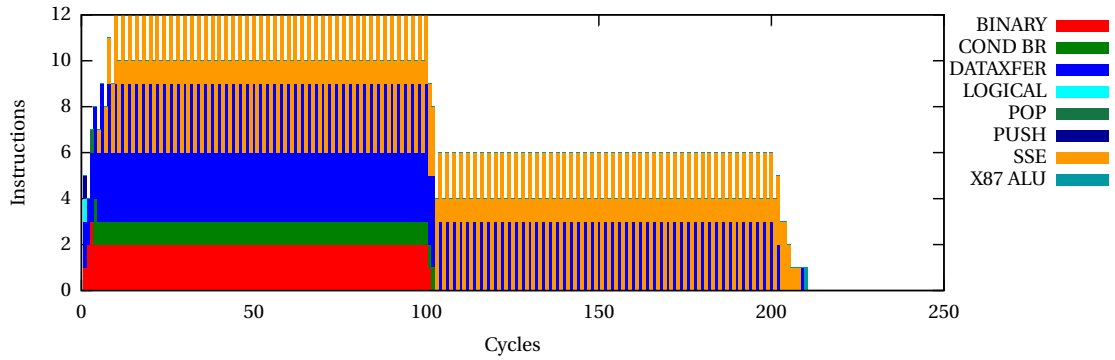
TABLE 3.4 – Reproductibilité des mesures effectuées avec PERPI (3 mesures avec $n = 100$).

La figure 3.4 illustre les possibilités de mesure de l'outil. Elle présente les graphes de parallélisme d'instruction idéaux des algorithmes de somme étudiés ici à titre d'exemple. On y voit pour chaque instruction, son type, et le cycle auquel elle peut être exécutée au plus tôt sur une machine idéale. Grâce à ce type de graphe, on visualise aisément la différence fondamentale entre les deux algorithmes SUM2 et SUMDD. Le premier est très parallélisable, alors que le second comporte des dépendances lecture après écriture qui le rend beaucoup plus séquentiel (210 contre 897 cycles).

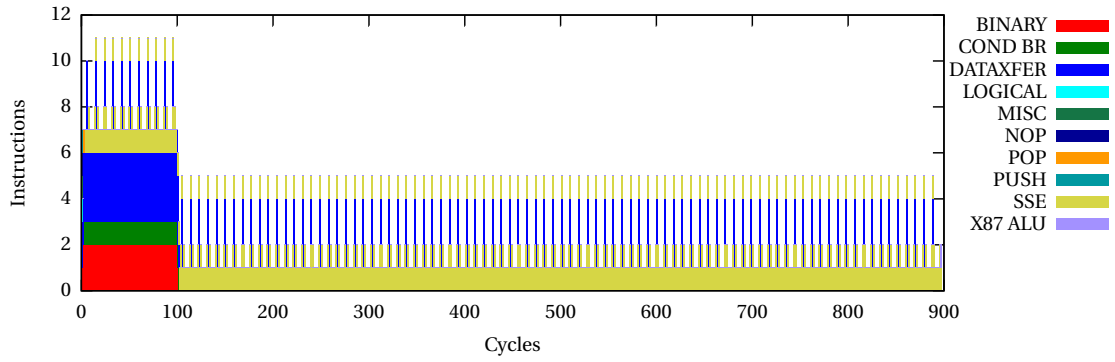
[1] `http://perso.univ-perp.fr/david.parello/perpi/`



(a) SUM : 104 cycles, 509 instructions et un IPC de 4,89



(b) SUM2 : 210 cycles, 1 509 instructions et un IPC de 7,61



(c) SUMDD : 897 cycles, 2 095 instructions et un IPC de 2,33

FIGURE 3.4 – Parallélisme d'instruction idéal des algorithmes SUM, SUM2 et SUMDD mesuré avec PERPI (somme de $n = 100$ termes).

Le tableau 3.5 présente les IPC des algorithmes SUM, SUM2 et SUMDD mesurées avec PERPI. On constate de nouveau que les mesures sont reproductibles d'une machine à l'autre. Les mesures idéales effectuées avec PERPI mettent en évidence les performances maximales des algorithmes de sommation étudiés ici. Par exemple sur une machine idéale, SUM2 (avec $n = 10^4$) a un IPC de 7,99 et est l'algorithme ayant le plus grand potentiel de parallélisation (SUMDD à quant à lui un IPC de 2,33). Mis en corrélation avec les mesures effectuées en réel sur machine, on constate que SUM2 reste l'algorithme le plus performant même si dans nos mesures, les machines utilisées ne parviennent à exploiter qu'entre 12 et 28% du parallélisme d'instruction idéal. On constate de plus que lorsque $n > 10^4$, le nombre de termes n'a plus d'influence sur les performances de ces algorithmes.

	Environnement A.1					Environnement A.2				
n	10^2	10^3	10^4	10^5	10^6	10^2	10^3	10^4	10^5	10^6
SUM	4,89	4,98	4,99	4,99	4,99	4,89	4,98	4,99	4,99	4,99
SUM2	7,61	7,95	7,99	7,99	7,99	7,61	7,95	7,99	7,99	7,99
SUMDD	2,33	2,33	2,33	2,33	2,33	2,33	2,33	2,33	2,33	2,33
SUMDD/SUM2	0,30	0,29	0,29	0,29	0,29	0,30	0,29	0,29	0,29	0,29

TABLE 3.5 – Mesures du niveau de parallélisme d'instruction (en IPC) des algorithmes SUM, SUM2 et SUMDD mesurés avec PERPI.

Les mesures détaillées du nombre de cycles et d'instructions sont présentés au tableau B.2 de l'annexe B.

3.6 Conclusion

Nous avons présenté deux notions importantes liées à la manipulation de programmes. Premièrement, nous avons évoqué les techniques de transformations classiques comme la compilation, la transformation source à source et la synthèse de code. Nous avons présentés leurs principales caractéristiques afin de justifier le choix que nous avons fait d'utiliser la synthèse de code. Dans une seconde partie, nous mettons en évidence les problèmes liés à la mesure des performances des programmes exécutés sur nos ordinateurs : les temps d'exécutions caractérisés soit, par un temps en secondes, par un nombre de cycles ou encore par un IPC. Nous avons notamment présenté les techniques de mesures les plus précises que nous utilisons dans nos travaux pour garantir des mesures fiables et reproductibles. En conclusion, nous pouvons dire que le niveau de parallélisme d'instruction, mesuré sur machine réelle (notamment à l'aide de compteurs matériels) et mis en corrélation avec les mesures effectuées sur machine idéale, est à ce jour le meilleur moyen de donner une idée réaliste du comportement intrinsèque d'un algorithme.

Deuxième partie

Vers des compromis performance – précision

CHAPITRE 4

CAS DE LA SOMME : ÉTUDE DES RELATIONS ENTRE PERFORMANCE ET PRÉCISION

4.1 Introduction

DANS l'optique d'améliorer les performances, les expressions arithmétiques sont réécrites suivant des lois mathématiques usuelles telles que l'associativité ou la distributivité. Ces lois ne s'appliquant pas aux nombres flottants, certaines de ces réécritures voient leur précision numérique modifiée. Souvent, cette modification est néfaste dans le sens où les performances ne font pas bon ménage avec la précision (voir par exemple [Dem92, BBDN10]). L'approche proposée ici met en évidence les relations existantes entre performance et précision numérique par le biais d'une étude exhaustive de toutes les réécritures possibles d'une somme de n termes [LMT10b, LMT10a]. Cette étude vise à montrer que des compromis intéressants existent entre performance et précision. Selon l'application souhaitée, et pour des jeux de données spécifiques, il est possible de choisir une expression (parmi l'ensemble des réécritures), en relâchant la contrainte sur les performances, qui produit de bons résultats numériques (ou vice-versa).

La section 4.2 présente l'approche générale mise en place lors de cette étude. Les détails des expériences et de leurs résultats sont présentés à la section 4.3.

4.2 Présentation de l'approche

L'approche consiste à générer exhaustivement toutes les réécritures d'une somme de n termes afin d'en étudier la performance et la précision. Plus particulièrement, l'étude tente de mettre en évidence les relations existantes entre performance et précision selon des jeux de données spécifiques. Dans cette approche, la performance d'une expression est caractérisée par sa latence, sur une machine idéale, en nombre d'opérations flottantes. C'est-à-dire, qu'une expression de n termes, sommant séquentiellement, aura une latence de $n-1$ opérations flottantes. La même expression, sommant ses termes de façon complètement parallèle, aura une latence de $\lceil log_2(n-1) \rceil$ opérations flottantes (voir tableau 4.1). Cette mesure correspond à la notion du niveau de parallélisme (l'ILP, vu au chapitre 3) mesuré sur une machine idéale. Plus une expression est performante, plus son ILP est important.

	Expression	Performance (Latence en *flop*)
(i)	$(((a_1 + a_2) + a_3) + \cdots a_{n-1}) + a_n$	$n-1$
(ii)	$(((a_1 + a_2) + (a_3 + a_4)) + \cdots + (a_{\frac{n}{2}-1} + a_{\frac{n}{2}})) + (((a_{\frac{n}{2}+1} + a_{\frac{n}{2}+2}) + (a_{\frac{n}{2}+3} + a_{\frac{n}{2}+4})) + \cdots + (a_{n-1} + a_n))$	$\lceil log_2(n-1) \rceil$

TABLE 4.1 – Définition de la performance d'une expression de n termes sur machine idéale. Exemple d'une expression, (i) séquentielle, et (ii) parallèle. La latence est exprimée en nombre d'opérations flottantes (*flop*).

Cette section aborde le problème combinatoire rencontré lors de la génération exhaustive des réécritures (représentant toutes les expressions équivalentes sommant n nombres flottants) ainsi que les données choisies pour nos expériences.

4.2.1 Une combinatoire importante

L'étude exhaustive des réécritures d'une expression de n opérandes pose un problème d'ordre combinatoire. L'arithmétique flottante ne garantissant pas l'associativité ni la distributivité des opérations, il est possible de réécrire une expression de nombreuses manières et d'obtenir des résultats différents en termes de précision et de parallélisme.

Pour évaluer toutes les expressions mathématiquement équivalentes à une expression donnée, il est nécessaire de générer tous les parenthésages et les permutations d'opérandes possibles. Le nombre de parenthésages que peut prendre une somme de n opérandes est défini par la suite d'entier A000108 (`http://oeis.org/A000108`). Cette suite correspond au nombre de CATALAN et définit le nombre de parenthésages pour une expression de $n' = n-1$ opérateurs (voir équation 4.1).

$$C(n') = \binom{2n'}{n'} \times \frac{1}{n'+1} = \frac{(2n')!}{n'!(n'+1)!} \tag{4.1}$$

Néanmoins, il faut aussi tenir compte des permutations des opérandes afin de générer exhaustivement les réécritures d'une expression. Par exemple, la somme de quatre termes peut être parenthésée de deux façons, $((\cdot+\cdot)+\cdot)+\cdot$ et $(\cdot+\cdot)+(\cdot+\cdot)$. De plus, au sein d'un même parenthésage, il existe différentes manières de permuter les termes, telles que par exemple $((a_1+a_2)+a_3)+a_4 \neq ((a_1+a_2)+a_4)+a_3$, mais sans tenir compte des permutations telles que $((a_1+a_2)+a_3)+a_4 = ((a_2+a_1)+a_3)+a_4$ qui ne génèrent pas d'expressions équivalentes différentes. La suite A001147 (`http://oeis.org/A001147`) définit le nombre total de réécritures pour une expression de n opérandes (parenthésages et permutations compris). Cette suite correspond à la double factorielle des nombres impairs et est définie par l'équation 4.2.

$$(2n-1)!! = 1 \times 3 \times 5 \times \cdots \times (2n-1) \tag{4.2}$$

La table 4.2 donne les premières valeurs des deux suites d'entiers. Par exemple, la somme de dix termes peut être réécrite de 34 459 425 façons différentes (dont 4 862 parenthésages).

n	$C(n')$	$(2n-1)!!$
1	1	1
2	1	1
3	2	3
4	5	15
5	14	105
6	132	945
7	429	10 395
8	1 430	135 135
9	4 862	2 027 025
10	16 796	34 459 425
11	58 786	654 729 075
12	208 012	13 749 310 575

TABLE 4.2 – Nombres de CATALAN et nombres de réécritures totales pour une somme de n opérandes ($n' = n-1$).

Nous avons limité l'étude à la génération des expressions équivalentes de la somme de 10 termes devant la difficulté a générer exhaustivement les expressions équivalentes des sommes de $n > 10$ termes. En effet, la génération des 34,5 millions de réécritures pour la

somme de 10 termes est le maximum envisageable à ce jour avec les moyens de calcul dont nous disposons. Les algorithmes qui génèrent ces réécritures sont très gourmands, soit en mémoire, soit en temps de calcul. Ne serait-ce que pour les réécritures de la somme de 10 termes, nous avons dû utiliser les ressources d'un super-calculateur[1] disposant d'assez de mémoire vive pour les générer.

4.2.2 Des données représentatives

> Dans les expériences de ce chapitre, une donnée x est vue comme un simple scalaire mais représente dans les faits un intervalle $\mathbf{X}$ de valeurs de l'ordre de grandeur de x (les variations autour de x ne dépassent pas 10%). Selon le même raisonnement, une erreur e est le maximum des bornes (en valeur absolue) de l'intervalle d'erreur $\mathbf{E}$. Cet intervalle majore l'erreur absolue sur le résultat pour chaque réécritures [LMT10b].

La caractérisation de la précision numérique des réécritures d'une somme de n termes nécessite aussi de couvrir un large spectre de données, le plus représentatif possible. Pour ce faire, nous proposons de porter l'étude sur 8 jeux de données différents (notés $D1$, $D2$, $\cdots$, $D8$) qui illustrent les comportements indésirables de l'arithmétique flottante. Ces comportements, tels l'absorption et l'élimination, surgissent lorsque les opérandes sont d'ordres de grandeur significativement différents (voir section 1.3.2).

De ce fait, nous déterminons trois ordres de grandeur suffisamment larges pour produire les phénomènes indésirables causant d'importantes pertes de précision (élimination catastrophiques et absorption). Ces ordres de grandeur sont définis dans le format *binary64* de l'arithmétique IEEE 754 et valent, $1e^{-16}$ pour les petites valeurs (pv), 1 pour les valeurs moyennes (mv) et $1e^{16}$ pour les grandes valeurs (gv). Le tableau 4.3 présente les caractéristiques de chaque jeu de données. Sauf indication contraire, ces jeux de données sont utilisés dans toutes les expériences présentées dans ce chapitre, et sont générés aléatoirement suivant leurs caractéristiques.

	Signe	Répartition			Observations
		pv	mv	gv	
$D1$	+	80%	$\emptyset$	20%	Ces valeurs produisent de l'absorption et les expressions les plus précises devraient sommer en premier lieu les petites valeurs en ordre croissant.
$D2$	−	80%	$\emptyset$	20%	Observations similaires au cas précédent.

TABLE 4.3 – *Suite à la page suivante*

[1] Un noeud du centre de calcul HPC@LR disposant de 64 Go de mémoire vive.

Suite de la page précédente

	Signe	Répartition *pv*	Répartition *mv*	Répartition *gv*	Observations
$D3$	+	80%		20%	Les expressions sommant dans l'ordre croissant devraient être les meilleures.
$D4$	−	80%		20%	Observations similaires au cas précédent.
$D5$	±	80%	∅	20%	Les algorithmes précis devraient sommer en premier lieu les deux grosses valeurs (afin d'éviter les problèmes d'absorption et d'élimination). Dans un cas plus général, les meilleures expressions devraient sommer les valeurs dans l'ordre décroissant des valeurs absolues. C'est un cas classique de données mal conditionnées (voir [Hig02]).
$D6$	±	20%	40%	40%	Les grandes valeurs sont les seules à s'éliminer. Les meilleures expressions devraient sommer dans l'ordre décroissant des valeurs absolues.
$D7$	±	20%	40%	40%	Similaire au cas précédent, mais les grandes et les moyennes valeurs s'éliminent.
$D8$	±	20%	40%	40%	Similaire au jeu de données $D6$, mais seules les valeurs moyennes s'éliminent.

TABLE 4.3 – Descriptif des jeux de données $D1$ à $D8$.

4.3 Résultats expérimentaux

4.3.1 Approche générale

La figure 4.1 illustre les résultats obtenus lors de l'évaluation de chacune des réécritures d'une expression de 8 termes en arithmétique flottante. Les données utilisées sont toutes des valeurs moyennes (*mv*). Bien que ne présentant pas de gros problèmes de précision, la figure met en évidence la grande variété d'erreurs commises lors de l'évaluation de ces expressions. La figure détaille la répartition des erreurs pour chacune des 135 135 réécritures (somme de 8 termes), et ce pour chaque niveau de performance. Un niveau de performance définit un groupe de réécritures dont la latence, sur machine idéale, est caractérisée par le nombre d'opérations flottantes nécessaires à son évaluation. C'est-à-dire, tous les groupes allant de l'évaluation complètement séquentielle, avec une latence de $n-1$

opérations flottantes à l'évaluation totalement parallèle ayant une latente de $\lceil log_2(n-1) \rceil$ opérations flottantes.

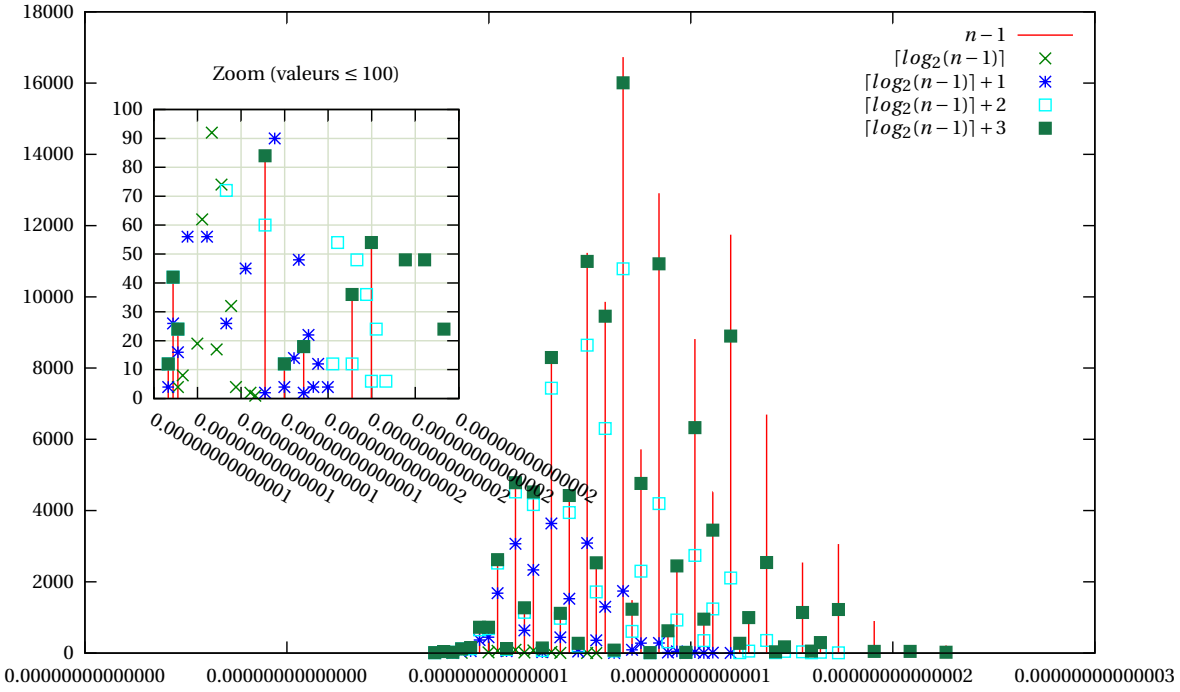


FIGURE 4.1 – Répartition des erreurs pour la somme de 8 nombres flottants selon différents niveaux de parallélisme. Les abscisses représentent l'erreur (calculée par la méthode d'analyse directe *Running Error Bound*) pour chaque réécriture dont le nombre est représenté en ordonnée.

Le processus de l'étude consiste à chercher, parmi les réécritures présentant les meilleures performances (donc le plus de parallélisme), celles qui présentent le plus de précision. Pour ce faire, nous cherchons parmi les différents niveaux de parallélisme, les réécritures qui présentent la meilleure précision afin d'en déterminer la proportion. Cette précision est caractérisée par le calcul d'une borne d'erreur selon la méthode d'analyse directe *Running Error Bound*. Bien que pessimiste, ce choix est motivé par le besoin de majoration de l'erreur dû à l'utilisation de l'arithmétique d'intervalles. Par exemple, dans le cas des résultats de la figure 4.1, nous constatons que les réécritures présentant les plus petites erreurs sont peu nombreuses. Similairement, nous constatons que les réécritures présentant les plus grosses erreurs sont peu nombreuses, et il s'avère que la plupart des réécritures présentent des erreurs comprises entre ces deux extrêmes. De plus, de part les différents niveaux de parallélisme, le nombre des réécritures présentant les caractéristiques recherchées est soit nul, soit très petit. Ceci est illustré par le zoom de la figure 4.1 qui montre que non seulement les réécritures ayant le meilleur niveau de parallélisme sont

peu nombreuses, mais surtout qu'elles ne présentent pas la précision maximale obtenue sur l'ensemble des réécritures.

Le tableau 4.4 exhibe les détails des observations tirées de la figure 4.1. Nous constatons que le niveau de parallélisme optimal, c'est-à-dire le niveau qui permet une latence minimale sur machine idéale, ne permet pas de générer une réécriture ayant la meilleure précision. Cependant, en relâchant légèrement la contrainte sur les performances, et en prenant en compte les réécritures d'un niveau de parallélisme moindre (non-optimal, mais s'en rapprochant), il est possible de générer une réécriture présentant la meilleure précision.

Performances (latence idéale)	Nombres de réécritures	Valeurs d'erreurs différentes	Erreurs minimales et maximales (nombre de réécritures concernées entre parenthèses)	
$\leq n-1$	135 135	45	$\mathbf{8.66e^{-15}(12)}$	$2.13e^{-14}(216)$
$\lceil log_2(n-1) \rceil$	315	11	$9.1e^{-15}(4)$	$1.26e^{-14}(1)$
$\leq \lceil log_2(n-1) \rceil + 1$	21 735	33	$\mathbf{8.66e^{-15}(12)}$	$1.6e^{-14}(4)$
$\leq \lceil log_2(n-1) \rceil + 2$	69 615	42	$\mathbf{8.66e^{-15}(12)}$	$1.8e^{-14}(6)$
$\leq \lceil log_2(n-1) \rceil + 3$	114 975	45	$\mathbf{8.66e^{-15}(12)}$	$2.13e^{-14}(24)$

TABLE 4.4 – Détails du nombres de réécritures et des erreurs minimales et maximales pour chaque niveau de parallélisme relatifs à la figure 4.1.

C'est dans ce contexte, et à l'aide de ces constatations préliminaires, que l'idée de rechercher des compromis entre performance et précision est apparue. En effet, nous verrons qu'il existe des réécritures qui présentent de bonnes performances et une bonne précision. À condition cependant que l'utilisateur soit prêt à consentir les légers sacrifices nécessaires sur ces critères permettant de trouver un bon compromis. Enfin, nous constatons que le nombre de réécritures présentant la meilleure précision est faible. Par exemple, dans l'ensemble des réécritures de la somme de 8 termes étudiée précédemment, le nombre d'expressions présentant la meilleure précision est de 12, soit environ moins de 1 pour 10 000.

Les résultats présentés dans ce chapitre permettent de montrer qu'il existe des compromis exploitables entre performance et précision. Cependant, cette approche n'a pas été retenue dans la suite de nos travaux. Une étude exhaustive, n'étant pas envisageable car trop complexe, nous avons porté nos efforts sur une méthode de transformation de code qui sera détaillée dans les chapitres suivants. Citons toutefois le travail de A. IOUALALEN et M. MARTEL [IM12] portant sur l'amélioration de la précision à base de réécritures qui pourrait prendre en compte dans le futur de tels compromis. En effet, ces travaux permettent

d'explorer en un temps polynomial des ensembles de réécritures possédant de telles combinatoires grâce à l'interprétation abstraite. Le choix d'une réécriture se fait selon le critère unique de la précision. Nous pourrions alors envisager une sélection multi-critères incorporant le temps d'exécution dans le choix d'une solution.

4.3.2 Étude étendue

L'étude étendue consiste à itérer le processus décrit précédemment sur les réécritures d'une somme de 10 termes et les jeux de données $D1$ à $D8$. Le nombre de réécritures s'élève à 34 459 425, soit autant de façons possibles d'évaluer une somme de 10 termes en arithmétique flottante. La figure 4.2 donne un exemple des diverses répartitions d'erreurs pour chacun des jeux de données. La figure 4.2(a) met en évidence, pour chaque jeu de données, une suite de traits verticaux qui représentent le nombre de réécriture présentant l'erreur indiquée en ordonnée. Il faut cependant remarquer qu'il existe une multitude d'autres traits, trop petits pour être visibles sur les figures avec les échelles linéaires. Par exemple, on dénombre 8 traits verticaux pour le jeu de données $D5$ alors qu'il y en a en réalité 571. C'est pourquoi la figure 4.2(b) montre les mêmes résultats suivant des échelles logarithmiques. Ces deux représentations utilisant des échelles différentes permettent de se faire une bonne idée de la répartition des erreurs.

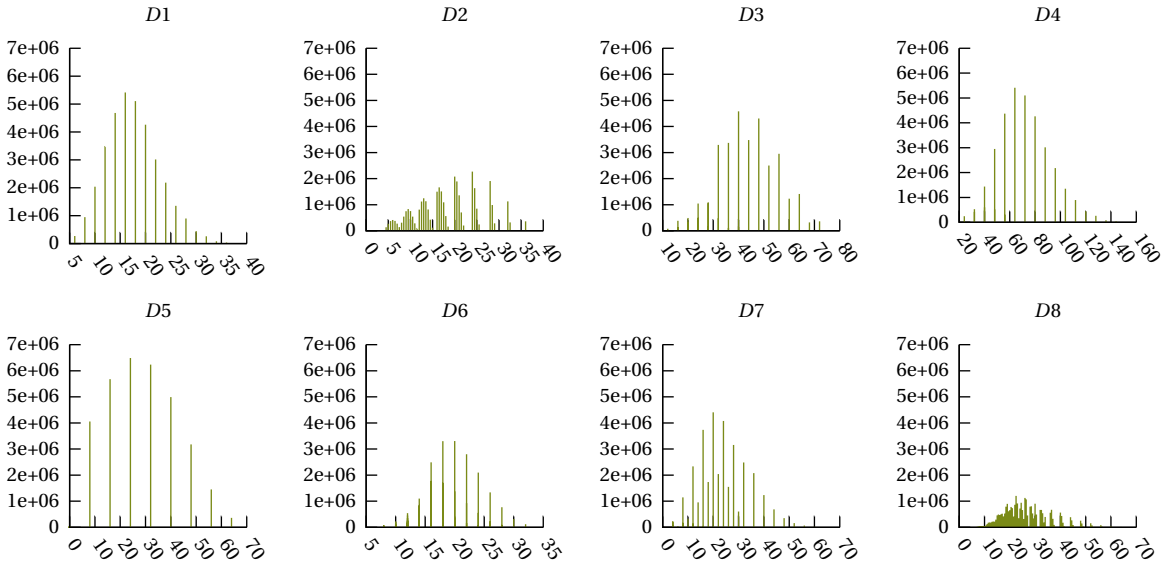
Les jeux de données sont générés aléatoirement suivant les caractéristiques qui leurs sont propres (voir tableau 4.3). Nous relevons ensuite la proportion de réécritures optimales pour chaque niveau de parallélisme étudié : de l'expression la plus séquentielle, à la plus parallèle en passant par tous les paliers intermédiaires. C'est-à-dire les réécritures, qui pour un niveau de parallélisme donné, obtiennent la meilleure précision qu'il est possible d'obtenir parmi l'ensemble complet des réécritures.

La figure 4.3 donne la proportion des expressions présentant la précision optimale selon chacun des niveaux de parallélisme et des jeux de données $D1$ à $D8$ pour les réécritures de la somme de 10 termes.

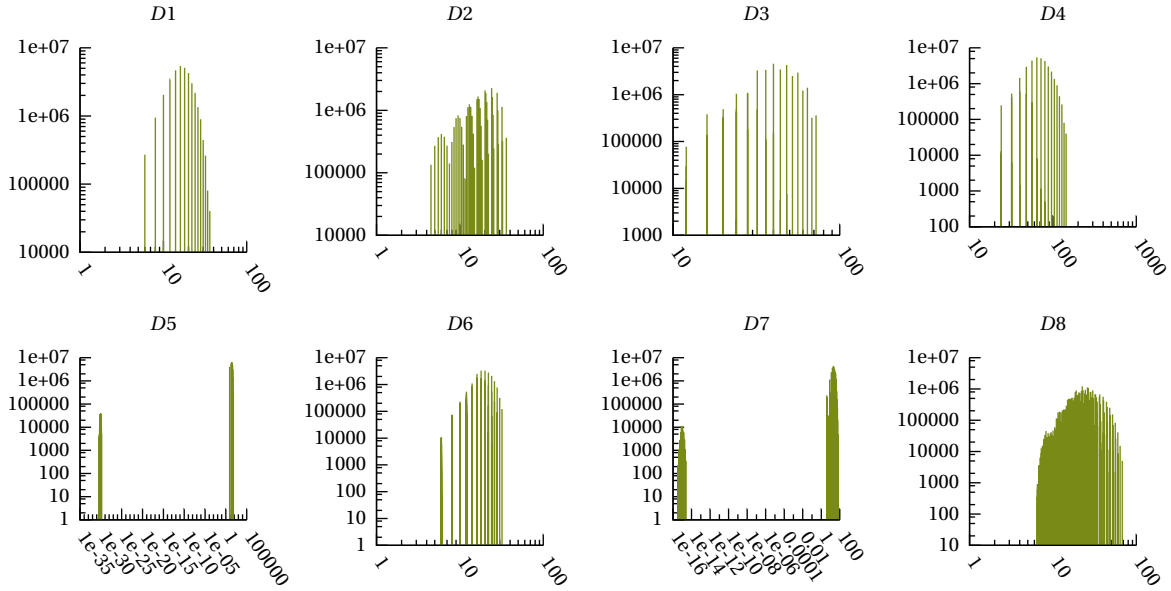
Soit u le nombre total d'expressions, et v_i le nombre d'expressions présentant la précision optimale selon chaque niveau $i \in \{\lceil log_2(n-1)\rceil,\ \lceil log_2(n-1)\rceil+1,\ \lceil log_2(n-1)\rceil+2\lceil log_2(n-1)\rceil+3,\ \lceil log_2(n-1)\rceil+4,\ n-1\}$ de parallélisme. La proportion p_i des expressions présentant la précision optimale est définie par :

$$p_i = \frac{v_i}{u} \times 100.$$

Nous remarquons que la proportion des expressions présentant la précision optimale est très petite (toujours inférieure à 0,5%). Elle est même le plus souvent nulle dans le cas du niveau de parallélisme $\lceil log_2(n-1)\rceil$ (exception faite des jeux de données $D6$ et $D7$). De plus, malgré la faible proportion générale, nous remarquons que plus on relâche la



(a) Échelle linéaire.



(b) Échelle logarithmique.

FIGURE 4.2 – Exemple de répartition des erreurs pour les réécritures de la somme de 10 termes selon les jeux de données $D1$ à $D8$. Les mêmes valeurs sont tracées deux fois selon deux échelles différentes (ordonnées : nombre de réécritures, abscisses : bornes d'erreurs).

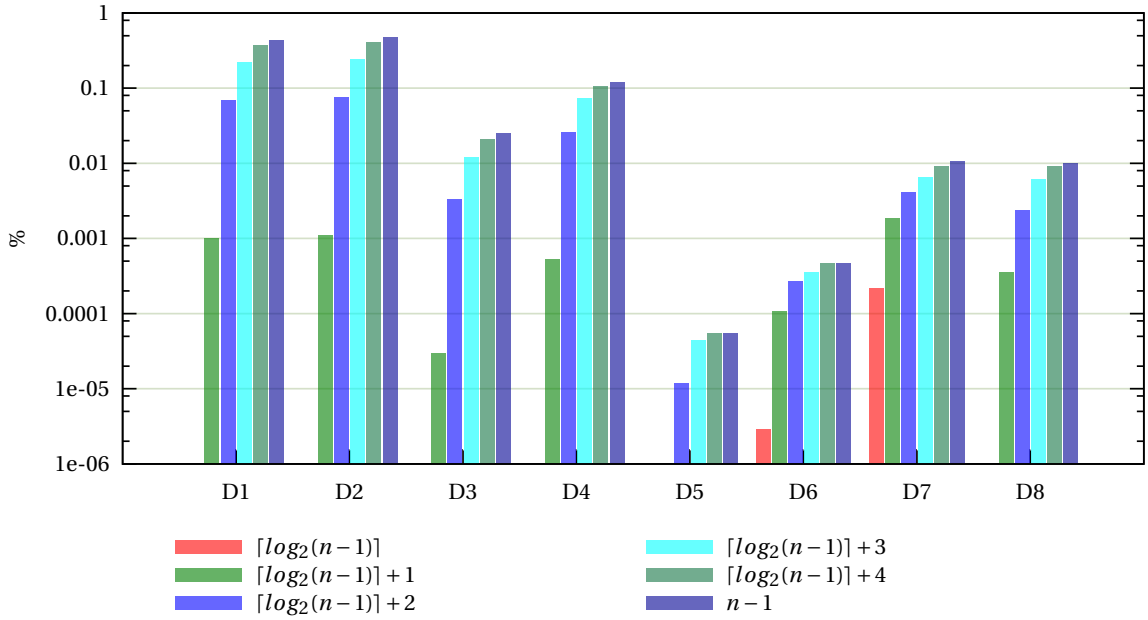


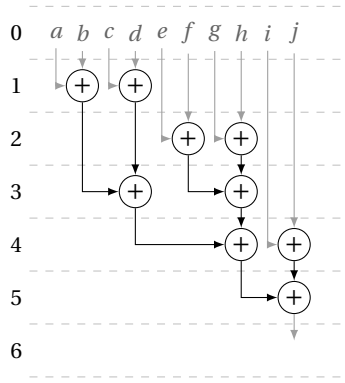
FIGURE 4.3 – Proportion (en %) des expressions présentant la précision optimale selon chacun des niveaux de parallélisme et des jeux de données $D1$ à $D8$ pour les réécritures de la somme de 10 termes (moyenne effectuée sur 10 jeux de données).

contrainte sur les performances, plus la proportion est importante, et plus il devient aisé de trouver une expression présentant la précision optimale. Ces résultats concordent avec ceux présentés au tableau 4.4 pour $n = 8$.

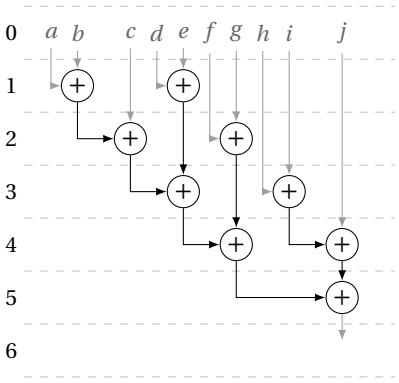
En conclusion, nous pouvons affirmer que, parmi l'ensemble des réécritures, le nombre d'expressions présentant la précision optimale est très faible (i.e. la meilleure précision). Cette proportion diminue à mesure que l'on considère l'ensemble des réécritures par les niveaux de parallélisme de plus en plus fort. Elle tend même vers zéro pour les expressions les plus parallèles. De ce fait, produire des expressions rapides à évaluer a une influence significativement négative sur la précision. On remarque cependant qu'en relâchant cette contrainte sur les performances il devient possible, et de plus en plus facile, de trouver une expression ayant une bonne précision. Ceci met en évidence le fait que des compromis existent entre performance et précision. Cependant, il faut relativiser l'incidence des compromis sur les performances en pratique, comme nous allons le voir ci-dessous.

4.4 Incidence des compromis en pratique

Les performances réelles des évaluations des expressions permettent de relativiser les pertes de performances dues à la recherche de compromis entre performance et précision. En effet, même si sur machine idéale, les performances ont été réduites, il s'avère qu'en pratique les limitations matérielles permettent de les compenser en partie, voir complètement. Ceci est illustré à la figure 4.4 par l'exemple de deux expressions sommant différemment 10 termes. L'une, complètement parallèle mais ne garantissant pas la précision maximale, a une latence idéale de $\lceil log_2(n-1) \rceil = 4$ opérations flottantes. Supposons que la machine sur laquelle cette expression est évaluée dispose uniquement de deux unités de calcul capables d'effectuer des opérations en parallèle. Il faut à cette machine « limitée » 5 cycles pour exécuter cette première expression (figure 4.4(a)). L'autre, a une latence légèrement réduite ($\lceil log_2(n-1) \rceil + 1 = 5$ opérations flottantes) sur machine idéale mais a besoin de 5 cycles pour s'exécuter sur notre machine hypothétique (comme le montre la figure 4.4(b)). Cette expression présente cependant une précision optimale. Sur cette machine réelle, le compromis entre performance et précision a comme seule influence de produire une meilleure précision sans avoir de conséquences sur les performances.



(a) Exemple d'expression présentant un parallélisme maximal (latence de $\lceil log_2(n-1) \rceil = 4$ opérations sur machine idéale) et une précision non-optimale.



(b) Exemple d'expression présentant un niveau de parallélisme réduit (latence de $\lceil log_2(n-1) \rceil + 1 = 5$ opérations sur machine idéale)) et une précision maximale.

FIGURE 4.4 – Graphe de flot de données d'expressions de 10 termes avec un parallélisme limité à 2 opérations flottantes par cycle (IPC maximal = 2).

CHAPITRE 5

AMÉLIORATION AUTOMATIQUE DE LA PRÉCISION

5.1 Introduction

AMÉLIORER automatiquement la précision d'algorithmes numériques consiste à minimiser ou à supprimer les erreurs d'approximations survenant lors des calculs. Il existe différentes méthodes automatiques permettant d'améliorer la précision. Les expansions, et avec elles, l'arithmétique *double-double* en sont un bon exemple. Elles permettent de générer rapidement des algorithmes disposant de k fois plus de précision (k étant la taille des expansions). Néanmoins, comme nous l'avons vu au chapitre 2, cette méthode est coûteuse en temps de calcul et affecte les performances. Il existe cependant des méthodes prenant en compte les performances finales d'un algorithme, tout en répondant à des critères plus stricts au niveau de la précision numérique. Ces méthodes sont souvent implémentées via des outils de compilation ou de synthèse de code. Citons l'outil CGPE qui permet une génération de code rapide et précis pour des évaluations polynomiales en arithmétique fixe [MR11]. Très récemment, C. RUBIO-GONZÀLEY *et al.*[1] ont mis au point un outil, PRECIMONIUS, permettant d'améliorer automatiquement la précision de calculs en arithmétique flottante. Cet outil est intégré dans l'infrastructure de compilation LLVM. Leur méthode se base sur une technique de *delta-debugging* et permet de sélectionner la précision minimale dont a besoin une variable flottante pour être représentée précisément

[1] J. DEMMEL, W. KAHAN et D. BAILEY étant reconnus pour leurs contributions de longue date dans ce domaine font partie des auteurs de ces travaux

(*float, double ou long double* en C) [RGNN+13]. En réduisant la taille de la représentation des nombres flottants, cette méthode permet d'améliorer la précision et les performances. Dans le futur, cette méthode pourrait prendre en compte d'autres moyens d'améliorer la précision, comme l'utilisation des compensations que nous allons décrire dans ce chapitre.

Nous proposons ici d'améliorer la précision numérique d'un calcul en compensant automatiquement les opérations élémentaires qui le composent [LMT12]. Cette approche consiste à calculer l'erreur induite par chaque opération élémentaire d'un calcul afin d'en compenser le résultat. Nous traiterons ici des opérations $+$, $-$ et $\times$ entre opérandes flottantes. Les opérations $\div$ et $\sqrt{}$ ne seront pas étudiées ici, mais quelques pistes seront néanmoins évoquées à la section 5.2.3.

Le principe de cette automatisation, est d'ajouter pour chaque opération élémentaire, un calcul d'erreur. Cette erreur est composée d'une part par l'erreur engendrée par l'opération elle-même, et, d'autre part, par l'héritage des erreurs entachant les opérandes. Par exemple, l'erreur de $\hat{a} \circ \hat{b}$ avec $\circ \in \{+, -, \times\}$ sera calculée en fonction de l'erreur de l'opération, notée $\delta_\circ$, et des erreurs entachant les opérandes notées $\delta_{\hat{a}}$ et $\delta_{\hat{b}}$. Lors d'une suite d'opérations élémentaires, les erreurs calculées pour chacune d'entre elles, sont accumulées pour ensuite être appliquées au résultat final. Cette dernière opération correspond à la phase de compensation du résultat final du calcul avec l'erreur totale ainsi calculée.

5.1.1 Notations

Définissons le format des nombres que manipule notre approche. Dans le cadre de la compensation automatique, un nombre flottant $\hat{x}$ se verra entaché de diverses erreurs, soit générées, soient héritées, lors des calculs du programme. L'accumulation de toutes les erreurs entachant le nombre $\hat{x}$, sera notée $\delta_{\hat{x}} \in \mathbb{F}$. Le nombre flottant $\delta_{\hat{x}}$ est une approximation de l'erreur commise par l'opération élémentaire ayant produit $\hat{x}$, plus une certaine fonction des erreurs héritées par les opérandes de cette opération. Le calcul de $\delta_{\hat{x}}$ est détaillé à la section 5.2.2 pour la somme et le produit.

Le couple $(\hat{x},\ \delta_{\hat{x}})$ est alors appelé nombre *auto-comp* [2]. Le nombre *auto-comp* représente le nombre flottant $\hat{x}$ ayant bénéficié du processus de compensation automatique (détaillé dans la section suivante). La notation utilisée pour décrire un tel nombre suit la relation (5.1).

$$(\hat{x},\ \delta_{\hat{x}}) = [\mathbf{x}] \tag{5.1}$$

On appelle $\hat{x},\ \delta_{\hat{x}}$ les composantes de $[\mathbf{x}]$.

[2] Abrégé ainsi pour nombre *auto*matiquement *comp*ensé.

Ce format prend alors en compte les erreurs d'approximation entachant les nombres flottants et permet, au fur et à mesure des calculs, de représenter la somme des erreurs héritées par les opérandes, et celles engendrées lors des diverses opérations flottantes.

Le format des nombres *auto-comp* est inspiré du format *double-double* mais ne prend pas en compte toutes leurs caractéristiques. Le format *auto-comp* permet la représentation des nombres jusqu'à deux fois plus de précision, mais à l'inverse des *double-double*, les composantes des *auto-comp* peuvent se chevaucher. De plus, nous constaterons que dans le cadre des algorithmes linéaires, la compensation automatique garantit un résultat final deux fois plus précis. En effet, toutes les erreurs du premier ordre sont corrigées par la compensation.

Nous verrons dans ce chapitre (et le suivant), le niveau d'approximation que $\delta_{\hat{x}}$ permet d'obtenir. De la correction de toutes les erreurs du premier ordre dans le cas de l'application complète de la compensation, à une correction incomplète lors de l'application partielle des compensations en vue de compromis avec les performances (voir chapitre 6).

5.2 Automatisation de la compensation

L'automatisation de la compensation est évoquée par J.M. MULLER *et al.* dans [MBdD+10, p. 182] :

> Some numerical algorithms that are simple enough (we need some kind of "linearity") can be transformed into compensated algorithms automatically.

Dans le livre [MBdD+10], cette citation est illustrée par un exemple antérieur, originellement dû aux travaux de P. LANGLOIS et de F. NATIVEL qui ont proposés la méthode CENA [Nat98, Lan00, BL02], capable d'effectuer la compensation automatique d'algorithmes linéaires. De plus, dans cette méthode, une borne d'erreur sur le résultat compensé est calculée grâce à une méthode de différenciation automatique. La définition 5.2.1 énumère les caractéristiques que doit avoir un algorithme au sein de l'arithmétique flottante (ensemble des nombres noté $\mathbb{F}$), pour être considéré comme un système linéaire.

Définition 5.2.1 (Algorithme linéaire dans $\mathbb{F}$)**.** *Un algorithme linéaire dans $\mathbb{F}$ est un algorithme qui contient uniquement les opérations élémentaires $\{+,-,\times,\div,\sqrt{}\}$, et vérifie que :*

- *la multiplication $fl(\hat{x}_i \times \hat{x}_j)$ satisfait $\hat{x}_i = x_i$ ou $\hat{x}_j = x_j$;*
- *la division $fl(\hat{x}_i \div \hat{x}_j)$ satisfait $\hat{x}_j = x_j$;*
- *et la racine carrée $fl(\sqrt{\hat{x}})$ satisfait $\hat{x} = x$.*

Les propriétés des algorithmes linéaires garantissent que les résultats obtenus à l'aide des compensations automatiques corrigent toutes les erreurs du premier ordre et per-

mettent ainsi de doubler la précision numérique des résultats. Si toutefois la compensation est appliquée dans le cadre d'algorithmes non linéaires, les résultats ne peuvent alors pas être garantis.

L'application automatique des compensations peut se faire par simple surcharge d'opérateurs, par exemple en FORTRAN dans le cas de la méthode CENA. Il existe aussi des outils spécifiques de transformation de code, comme celui développé dans le cadre de nos travaux (voir chapitre 8). Cette section décrit, dans un premier temps, le principe d'automatisation que nous avons développé et les algorithmes utilisés pour compenser les opérations élémentaires $+$, $-$, et $\times$. Les opérations $\div$ et $\sqrt{}$ seront ensuite brièvement évoquées. Enfin, nous conclurons cette section par un exemple d'utilisation de cette méthode.

5.2.1 Le principe de l'automatisation

La méthode que nous avons développée pour automatiser la compensation suit le principe décrit ci-dessous. Cette méthode est implémentée dans les outils que nous avons spécialement créés afin d'appliquer automatiquement la compensation dans des programmes écrit en langage C. Ces outils seront décrit plus en détail dans la section 7.3 du chapitre 7.

(i) Chaque nombre flottant $\hat{x}$ devient un couple de valeur $(\hat{x}, \delta_{\hat{x}})$, c'est-à-dire le nombre *auto-comp* [**x**], tel que décrit précédemment par la relation (5.1).

(ii) Les opérations élémentaires constituant les séquences de calculs sont remplacées par des opérateurs « compensés ». En effet, afin d'appliquer la compensation, il faut ajouter du calcul pour (a), calculer l'erreur engendrée par l'opération élémentaire, et (b), accumuler toutes les erreurs destinées à compenser le résultat ou les résultats des calculs. C'est pourquoi, il faut dans un premier temps détecter les séquences d'opérations flottantes au sein du programme à compenser, c'est-à-dire les suites d'opérations dépendantes nécessaires à l'obtention d'un ou plusieurs résultats. Pour ensuite remplacer les opérations élémentaires par des opérateurs *compensés* prenant en entrée deux nombres *auto-comp* pour fournir en résultat un nombre *auto-comp* représentant le résultat de l'opération élémentaire et la somme des erreurs accumulées par l'opération et les opérandes. Ces opérateurs sont décrits à la section 5.2.2 via les algorithmes 5.1 et 5.2 compensant respectivement la somme et le produit.

(iii) Enfin, il est nécessaire d'appliquer les compensations aux bons endroits. Ceci est assuré par l'opération de *fermeture* dont la description est donnée par la définition 5.2.2. Cette opération est la phase de compensation proprement dite. En effet, c'est à cet instant que l'erreur calculée lors des calculs est ajoutée aux résultats produit par la séquence d'opérations.

5.2.2 Compensation automatique de la somme et du produit

Les algorithmes 5.1 et 5.2 effectuent respectivement la somme et le produit de deux nombres *auto-comp*. L'erreur générée par l'opération élémentaire est calculée à l'aide d'une transformation sans erreur (EFT) pour être ensuite accumulée avec les erreurs héritées par les opérandes, à la manière du calcul des bornes d'erreurs de la méthode d'analyse directe *Running Error Bound* (voir tableau 1.3 du chapitre 1).

Nous considérons dorénavant que tous les nombres manipulés appartiennent à l'ensemble $\mathbb{F}$ des flottants. Par soucis de clarté, nous utiliserons la notation $\hat{x}$ pour signaler un nombre flottant uniquement si des nombres réels sont aussi utilisés.

La somme

L'algorithme 5.1 permet d'effectuer la somme de deux nombres flottant *compensés*.

1. **fonction** AUTOCOMP_TWOSUM($(a,\ \delta_a),\ (b,\ \delta_b)$)
2. $[s,\ \delta_\circ]$ = TWOSUM$(a,\ b)$
3. $\delta_s \leftarrow RN((\delta_a + \delta_b) + \delta_\circ)$
4. **retourner** [**s**] $\leftarrow (s,\ \delta_s)$
5. **fin fonction**

Algorithme 5.1 – AUTOCOMP_TWOSUM.

Démonstration. Soient [**a**] et [**b**] deux nombres *auto-comp* : deux nombres flottants a et b entachés d'une erreur δ_a et δ_b.

À partir de la borne d'erreur dérivée du modèle standard de l'arithmétique flottante (1.4), on déduit l'erreur δ_s qui entache le résultat s de l'opération d'addition par la relation suivante (le raisonnement est identique pour la soustraction) :

$$\delta_s = \delta_\pm + \delta_a + \delta_b.$$

Le calcul de l'erreur générée par l'opération, $\delta_\pm$ est assuré par la transformation sans erreur TWOSUM. Les erreurs δ_a et δ_b sont quant à elles les erreurs héritées par les opérandes. De ce fait, δ_s est l'approximation de l'erreur totale propagée par l'opération d'addition. □

Le produit

L'algorithme 5.2 permet d'effectuer le produit de deux nombres flottant *compensés*.

1. **fonction** AUTOCOMP_TWOPRODUCT($(a,\ \delta_a),\ (b,\ \delta_b)$)
2. $[s,\ \delta_\times] = \text{TWOPRODUCT}(a,\ b)$
3. $\delta_s \leftarrow RN(((a \times \delta_b) + (b \times \delta_a)) + \delta_\times)$
4. **retourner** $[\mathbf{s}] \leftarrow (s,\ \delta_s)$
5. **fin fonction**

Algorithme 5.2 – AUTOCOMP_TWOPRODUCT.

Démonstration. Soient [**a**] et [**b**] deux nombres *auto-comp* : deux nombres flottants a et b entachés d'une erreur δ_a et δ_b.

À partir de la borne d'erreur dérivée du modèle standard de l'arithmétique flottante (1.4), on déduit l'erreur δ_s qui entache le résultat s de l'opération de multiplication par la relation suivante :

$$\delta_s = \delta_\times + a \times \delta_b + b \times \delta_a + \delta_a \times \delta_b.$$

Le calcul de l'erreur générée par l'opération, $\delta_\times$ est assuré par la transformation sans erreur TWOPRODUCT. Les erreurs δ_a et δ_b sont quant à elles les erreurs héritées par les opérandes. Le produit des erreurs de second ordre $\delta_a \times \delta_b$ étant suffisamment petit devant les autres termes d'erreurs peut être négligé. De ce fait,

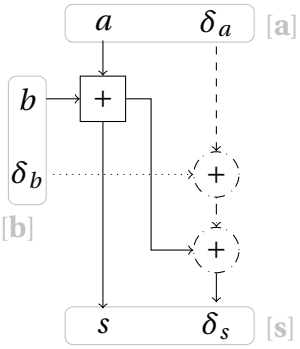
$$\delta_s = \delta_\times + a \times \delta_b + b \times \delta_a$$

est l'approximation au premier ordre de l'erreur totale propagée par l'opération de multiplication. □

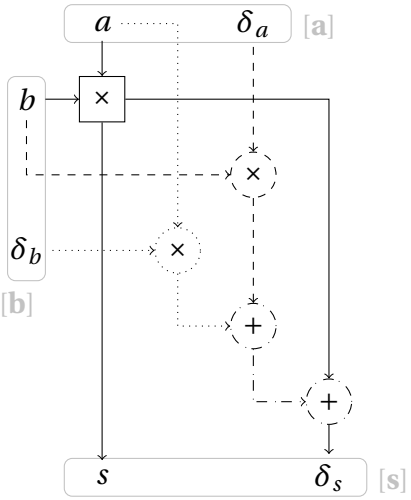
Implémentation

Nous avons implémenté les algorithmes 5.1 et 5.2 au sein d'un outil de synthèse de code qui sera présenté en détail au chapitre 8. Ils peuvent cependant être aisément appliqués par surcharge des opérateurs si le langage de programmation utilisé le permet. La figure 5.1 donne la représentation graphique des algorithmes 5.1 et 5.2 ainsi que leurs variantes. Ces variantes, obtenues en supprimant les constructions en tirets ou en pointillés de la figure 5.1, sont utilisées en fonction de la nature des entrées des opérateurs, c'est-à-dire des nombres flottants ou des nombres *auto-comp*. L'annexe C donne un exemple complet de code pour la compensation automatique dans le cas de l'algorithme HORNER.

Les algorithmes 5.1 et 5.2 utilisent des transformations sans erreur pour calculer exactement l'erreur générée lors de l'opération élémentaire $\circ \in \{+, -, \times\}$. Cette erreur $\delta_\circ$ est ensuite accumulée avec les erreurs héritées par les opérandes si celles-ci sont préalablement entachées d'erreurs, soit d'approximations sur les données, soit d'évaluations antérieures.



(a) AUTOCOMP_TWOSUM.



(b) AUTOCOMP_TWOPRODUCT.

FIGURE 5.1 – Représentation des algorithmes 5.1 et 5.2 pour la compensation automatique de, (a) la somme, et (b) le produit. Les dessins en tirets ou en pointillés doivent être respectivement supprimés pour obtenir la représentation des algorithmes quand a ou b ne sont pas tirés de nombres *auto-comp* mais de simples nombres flottants.

De ce fait, plusieurs variantes de ces algorithmes sont implémentées dans le but de réduire le coût induit par l'application de la compensation. Le tableau 5.1 récapitule les caractéristiques en terme de nombre d'opérations flottantes et de latence idéale des diverses variantes de ces algorithmes. De plus, pour comparaison, ces mêmes mesures sont présentées pour les algorithmes *double-double*.

Les algorithmes de compensation automatique sont toujours plus performants que les algorithmes *double-double*, tant en terme de nombres d'opérations flottantes, qu'en latence sur machine idéale. La principale différence réside dans la latence nécessaire à l'obtention du résultat du calcul original. En effet, les opérateurs *double-double*, de part leur phase de normalisation impliquent une forte latence qui réduit grandement la capacité des codes ainsi transformés à se paralléliser ensuite lors de leurs exécutions (voir chapitre 2.5). Les opérateurs *auto-comp* sont libres de cet inconvénient et procurent un meilleur potentiel à la parallélisation. De plus, si l'on dispose d'un FMA, les performances de l'algorithme AUTOCOMP_TWOPRODUCT peuvent être améliorées (voir algorithme 2.5). Ces observations se confirment en pratique lors de l'évaluation des performances des programmes ayant subit ces différents processus. De telles mesures sont présentées à la section 5.3.

La compensation automatique des algorithmes linéaires permet de corriger les résultats numériques des calculs flottants, garantissant une précision du même ordre de gran-

Algorithme	δ_a	δ_b	*flop*		Latence	
			s	δ_s	s	δ_s
5.1 : AUTOCOMP_TWOSUM	✓	✓	1	8	1	6
	✓	✗	1	7	1	6
	✗	✗	1	6	1	5
5.2 : AUTOCOMP_TWOPRODUCT	✓	✓	1	21	1	9
	✓	✗	1	19	1	9
	✗	✗	1	17	1	8
2.11 : QDLIBDD_TWOSUM	✓	✓	18	20	11	13
	✓	✗	8	10	7	9
	✗	✗	1	6	1	5
2.12 : QDLIBDD_TWOPRODUCT	✓	✓	22	24	11	13
	✓	✗	20	22	10	12
	✗	✗	1	17	1	8

TABLE 5.1 – Latence idéale et nombre d'opérations flottantes (*flop*) des algorithmes de compensation automatique 5.1 et 5.2 et des algorithmes *double-double* 2.11 et 2.12 avec détails selon la nature des opérandes (présence de δ_a et de δ_b).

deur que la précision des calculs effectués en *double-double*. Remarquez que dans ce contexte, l'algorithme 5.2, dans sa variante présentant δ_a et δ_b ne peut pas être utilisé car il ne répond pas aux critères de linéarité.

5.2.3 Compensation automatique de la division et de la racine carrée

Dans le cadre de ces travaux, les opérations de division et de racine carrée n'ont pas été étudiées. De ce fait, nous n'avons pas développé d'algorithmes pour effectuer les compensations automatiques de ces opérations. Nous avons vu au chapitre 2 que ces opérations disposent de transformations sans erreur. Néanmoins ces EFT ne permettent pas de calculer simplement les erreurs engendrées par les opérations de division ou de racine carrée. Par exemple, l'EFT proposée par M. PICHAT et J. VIGNES [PV93] pour la division ne permet pas de calculer l'erreur qui est commise lors de cette opération, mais un quotient q et un reste r.

Cependant d'autres méthodes peuvent être employées pour améliorer la précision de ces opérations élémentaires, comme par exemple, des approximations par la méthode de NEWTON-RAPHSON. Ces points seront abordés lors de la réalisation de nos futures perspectives.

5.2.4 Utilisation

L'exemple de la figure 5.2 représente la compensation automatique de l'expression $x = RN(((((a+b)+c)\times d)+e)-(f\times g))$. La transformation se fait en deux temps :

(i) par l'application des opérateurs *auto-comp* à chaque opérations flottantes de la séquence $((((a+b)+c)\times d)+e)-(f\times g)$. Cette étape est représentée par les cadres en pointillés sur la figure 5.2(b) ; et

(ii) à la *fermeture des séquences* d'opérations flottantes ainsi transformées. C'est l'opération de compensation proprement dite comme spécifiée à la définition 5.2.2. Cette étape est représentée par le cadre en tirés sur la figure 5.2(b).

Définition 5.2.2 (Fermeture d'une séquence)**.** *Une séquence est une suite d'opérations dépendantes nécessaires à l'obtention d'un ou plusieurs résultats.*

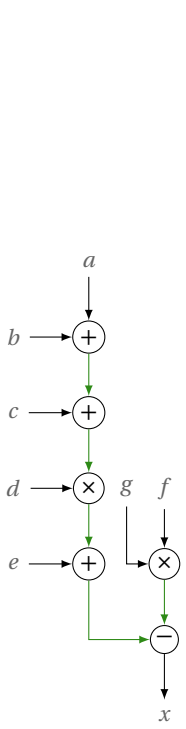
La fermeture d'une séquence d'opérations de compensations automatiques permet la somme flottante des composantes du nombre auto-comp du ou des résultats. C'est à ce moment que les résultats du calcul sont compensés de leurs erreurs. La fermeture est noté $\Gamma(\cdot)$ *et est définie par :*

$$\Gamma([\mathbf{x}]) = RN(x+\delta_x).$$

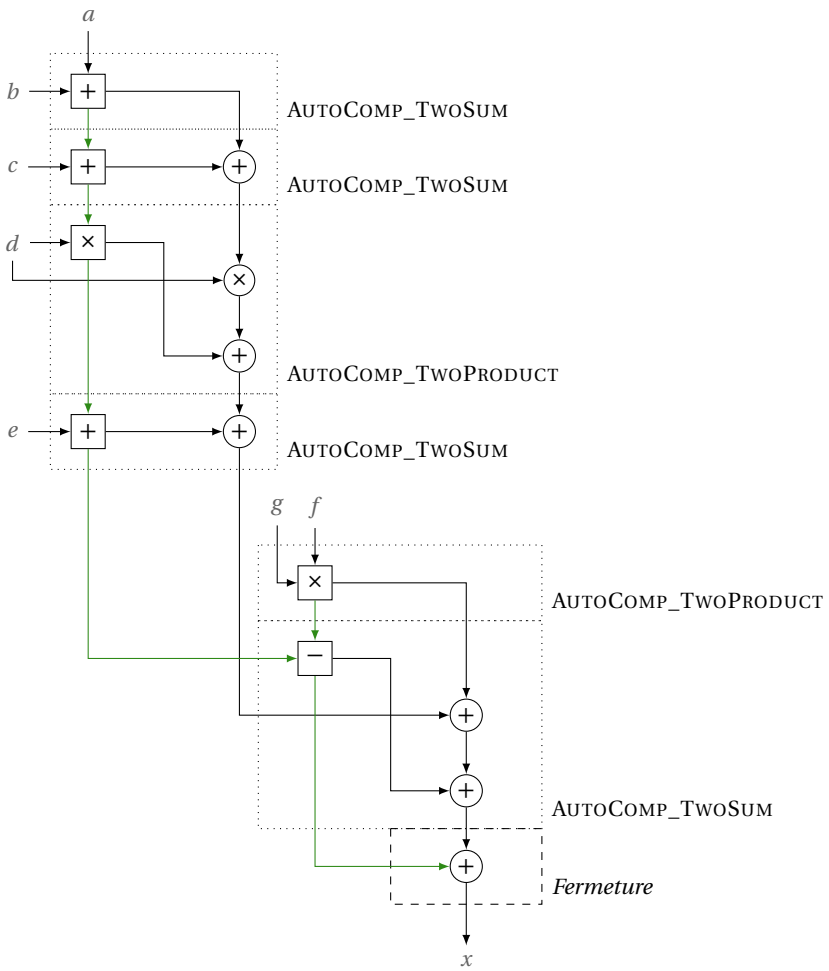
5.3 Résultats expérimentaux

Cette section présente les résultats de nos expériences visant à comparer notre méthode aux algorithmes compensés « à la main » présents dans la littérature. Cette comparaison est effectuée au niveau des performances et de la précision. La performance est exprimée en nombre d'instructions et de cycles, et est calculée à l'aide des outils PAPI et PERPI. La précision quant à elle, est exprimée en fonction du nombre de bits significatifs que contient les différents résultats. Ce nombre est calculé à partir de l'équation (1.2) présentée au chapitre 1 et utilise l'erreur relative pour fournir ces résultats. Cette erreur relative est calculée à partir d'un résultat approché, et d'une valeur de référence, considérée comme le résultat réel. La valeur de référence est calculée avec 1024 bits de précision grâce à MPFR.

Nous avons étudiés l'algorithme de somme SUM [ROO05] ainsi que les algorithmes d'évaluation polynomiale HORNER [GLL09], HORNERDER [JGH+13], DECASTELJAU, DECASTELJAUDER [JLCS10], CLENSHAWI et CLENSHAWII [JBL+11]. Nous détaillerons dans cette section les résultats des algorithmes SUM, HORNER et CLENSHAWI. Enfin, pour conclure ce chapitre, nous présenterons une synthèse des résultats obtenus sur l'ensemble des algorithmes cités ci-dessus.



(a) Séquence initiale.



(b) Séquence compensée.

FIGURE 5.2 – Exemple d'application automatique des opérateurs *auto-comp* sur l'expression $x = RN(((((a+b)+c)\times d)+e)-(f\times g))$.

5.3.1 La somme

S. RUMP *et al.* ont développés l'algorithme SUM2 [ROO05] (voir algorithme 2.8). Cet algorithme est la version compensée de l'algorithme 2.7 de sommation SUM. L'implémentation de l'algorithme SUM, compensé automatiquement est donnée par le code 5.1. Le code généré est identique au code de l'algorithme compensé SUM2 (voir code 2.1). Chaque itération de la boucle à un coût de 7 opérations flottantes, soit un coût total de $7(n-1)+1$ opérations flottantes.

Code 5.1 : SUMAC : compensation automatique du code 3.1. ◁

```
Ligne 1  s = a[0];
       -  delta_s = 0.0;
       -
       -  for(i=1; i<n; i++)
       5    {
       -      /* [s, delta_s] = AutoComp_TwoSum(s, delta_s, a[i]) */
       -      /* [s, delta_plus] = TwoSum(s, a[i]) */
       -      tmp_s = s;
       -      s = s + a[i];
      10      t = s - tmp_s;
       -      delta_plus = (tmp_s - (s - t)) + (a[i] - t);
       -
       -      /* propagation des erreurs */
       -      delta_s = delta_s + delta_plus;
      15    }
       -
       -  /* fermeture */
       -  s = s + delta_s;
```

Le tableau 5.2 présente les performances des algorithmes de sommation. Les mesures confirment que les performances de l'algorithme SUMAC, qui a été généré automatiquement, sont égales aux performances de l'algorithme SUM2. En comparant ces résultats avec les performances d'un autre algorithme généré automatiquement, SUMDD, nous pouvons remarquer que dans le cas de la compensation automatique, les performances sont améliorées jusqu'à correspondre aux performances de l'algorithme compensé « manuellement ».

Les mesures effectuées grâce à PAPI montrent que sur l'environnement étudié, les performances de l'algorithme compensé automatiquement sont équivalentes à celles de l'algorithme SUM2. Les mesures montrent un léger avantage pour SUMAC (nombre de cycles). Cependant, comme le confirme les mesures effectuées avec PERPI, cet avantage est uniquement dû à une imprécision de mesure car les deux algorithmes présentent exactement les mêmes performances.

La figure 5.3 fournie la comparaison des algorithmes SUM, SUM2, SUMDD et SUMAC au niveau de la précision de leurs résultats. Cette précision est exprimée en nombre de bits corrects, c'est-à-dire le nombre de bits de précision significatifs du résultat. Nous avons étudié les résultats de ces algorithmes appliqués à 256 jeux de données de $n = 10^3$ et $n = 10^5$ nombres flottants selon des conditionnements allant jusqu'à 10^{40}. Afin de générer les

	PAPI			PERPI		
	Instructions	Cycles	IPC	Instructions	Cycles	IPC
SUM	500 019	302 051	1,66	500 009	100 004	4,99
SUM2	1 600 009	711 290	2,25	1 599 999	200 010	7,99
SUMDD	2 100 005	2 406 581	0,87	2 099 995	899 997	2,33
SUMAC	1 600 009	710 006	2,25	1 599 999	200 010	7,99
SUMAC/SUM2	1,00	1,00	1,00	1,00	1,00	1,00
SUMAC/SUMDD	0,76	0,30	2,58	0,76	0,22	3,42

TABLE 5.2 – Mesures de performances des algorithmes SUM, SUM2, SUMDD et SUMAC avec $n = 10^5$ (les chiffres obtenus avec PAPI sont les moyennes de 10 000 mesures réalisées dans l'environnement A.1).

données mal conditionnées, nous avons utilisés le générateur proposé par P. LANGLOIS *et al.* dans [LPGP12].

La précision du résultat de l'algorithme de sommation classique $\sum_{i=1}^{n} x_i$ (SUM) est borné par $(n-1) \times cond \times u$, avec $cond$ le nombre de conditionnement définit par :

$$cond = \frac{\sum |x_i|}{|\sum x_i|}.$$

Dans le format *binary64* de l'arithmétique flottante, le résultat d'une somme ne contient aucun bits significatifs lorsque le conditionnement est élevé, c'est-à-dire quand $n \times cond$ est supérieur à 10^{16}.

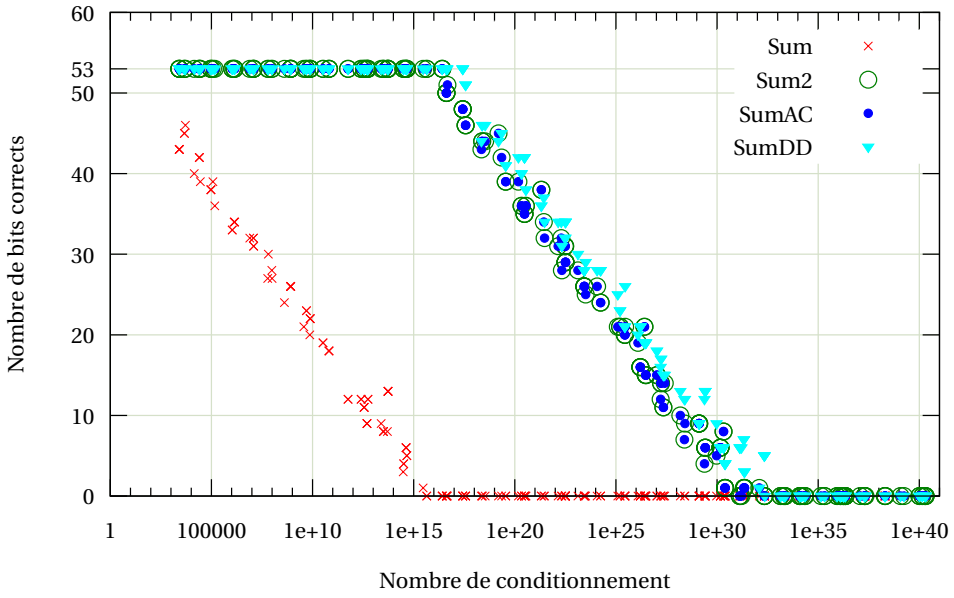
La figure 5.3 permet de remarquer que les algorithmes SUM2, SUMDD et SUMAC offrent la même précision numérique, avec cependant un léger avantage à SUMDD. Nous observons de plus, un comportement identique suivant le nombre d'opérations et le conditionnement pour chacun des algorithmes améliorant la précision.

5.3.2 L'évaluation polynomiale

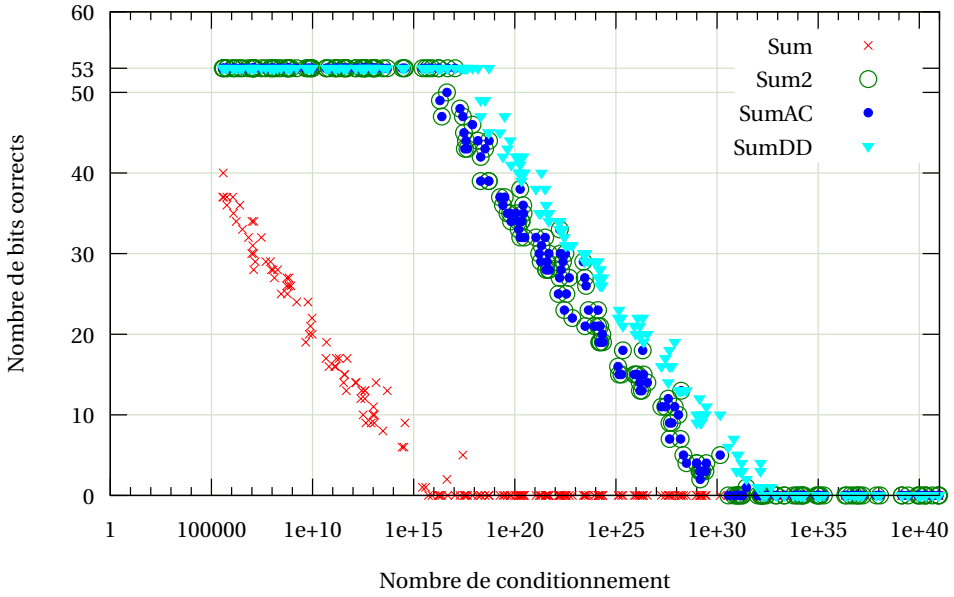
Dans cette partie, nous allons aborder les algorithmes d'évaluation polynomiale suivants :

- l'algorithme 5.3 qui évalue un polynôme $p_H(x)$ suivant le schéma de HORNER ;
- et l'algorithme 5.4 qui évalue un polynôme $p_C(x)$ suivant le schéma de TCHEBYCHEV.

S. GRAILLAT *et al.* et JIANG *et al.* en ont respectivement proposés des algorithmes compensés, COMPHORNER et COMPCLENSHAWI dans [GLL09] et [JBL+11].



(a) $n = 10^3$



(b) $n = 10^5$

FIGURE 5.3 – Nombres de bits corrects du résultat de l'évaluation de la somme de 256 jeux de données de $n \in \{10^3, 10^5\}$ nombres flottants selon des conditionnements allant jusqu'à 10^{40}.

1. **fonction** HORNER(p, x)
2. $r_n \leftarrow a_n$
3. **pour** $i = n-1 : -1 : 0$ **faire**
4. $r_i \leftarrow fl(r_{i+1} \times x + a_i)$
5. **fin pour**
6. **retourner** r_0
7. **fin fonction**

Algorithme 5.3 – HORNER.

1. **fonction** CLENSHAWI(p, x)
2. $b_{n+2} \leftarrow b_{n+1} \leftarrow 0$
3. **pour** $j = n : -1 : 1$ **faire**
4. $b_j \leftarrow fl(2xb_{j+1} - b_{j+2} + a_j)$
5. **fin pour**
6. $b_0 \leftarrow fl(xb_1 - b_2 + a_0)$
7. **retourner** b_0
8. **fin fonction**

Algorithme 5.4 – CLENSHAWI.

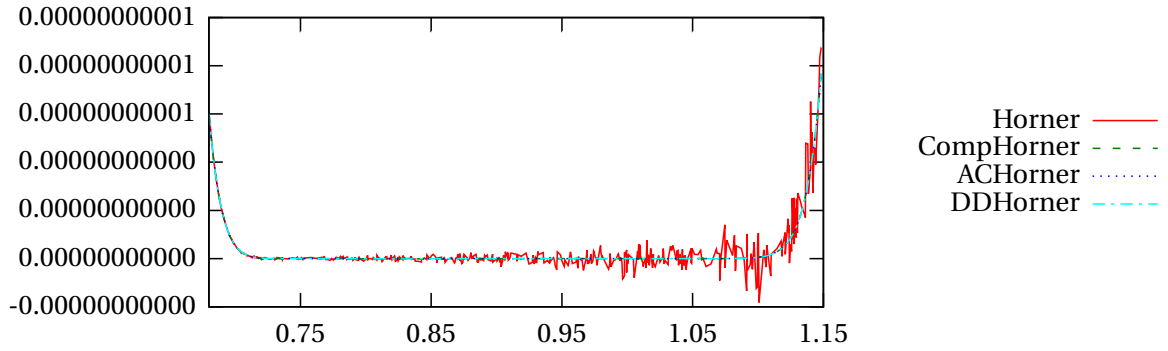
Le cas de l'algorithme HORNER

L'algorithme COMPHORNER [GLL09] permet d'effectuer l'évaluation polynomiale compensée suivant le schéma de HORNER. Considérons DDHORNER, la version *double-double* de l'algorithme 5.3. La figure 5.4(a) présente les résultats de l'évaluation du polynôme $p_H(x) = (x-0,75)^5(x-1,0)^{11}$ pour 512 données choisies aléatoirement entre 0,68 et 1,15 selon une distribution uniforme. Les résultats de l'évaluation par l'algorithme HORNER sont peu précis et oscillent de façon importante autour des résultats exacts. Néanmoins, ce manque de précision est en partie résolu par les algorithmes compensés et *double-double* fournissant le double de précision. La figure 5.4(b) présente ces mêmes résultats sur des petits intervalles centrés autour des racines du polynôme. Nous pouvons y faire les mêmes observations que précédemment mais nous remarquons néanmoins que, bien que la précision soit améliorée, elle n'est pas suffisante pour fournir les résultats exacts.

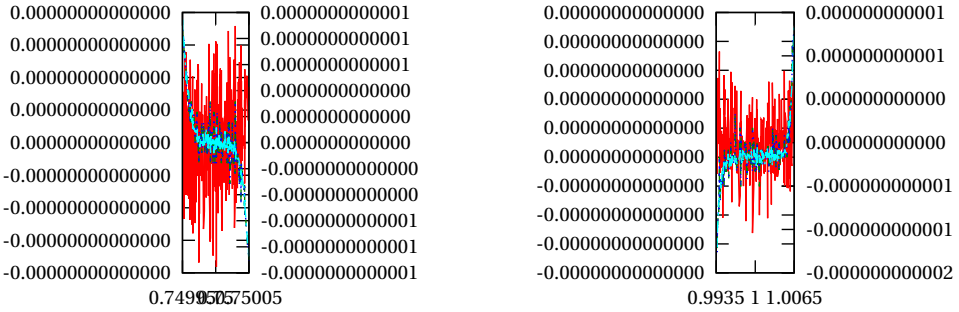
Les courbes de la figure 5.5 montrent le nombre de bits corrects des résultats des différentes variantes des algorithmes de HORNER. Sur les 512 évaluations[3], l'algorithme HORNER offre en moyenne 0,61 bits de précision contre 41,75 pour COMPHORNER, 42,5 pour DDHORNER et 41,74 pour ACHORNER. Ces résultats confirment que les algorithmes compensés COMPHORNER et ACHORNER améliorent la précision des résultats à l'instar de l'algorithme DDHORNER.

L'annexe C contient les codes en langage C des algorithmes compensés de l'évaluation de HORNER. Le code C.1 implémentant l'algorithme COMPHORNER est la version fournie dans [Lou07, p. 63] (compensée et optimisée manuellement). Le code C.2, implémentant la version compensée automatiquement ACHORNER effectue sensiblement les mêmes calculs. La différence qui existe entre la version compensée manuellement et la version compensée automatiquement réside dans le fait que cette dernière version ne prends pas en compte les invariants de boucles créés lors de la transformation de code (voir code C.2 et

[3] Nombre comparable à ceux trouvés dans [GLL09].



(a) $p_H(x)$ avec $x \in \{0,68 : 1,15\}$.

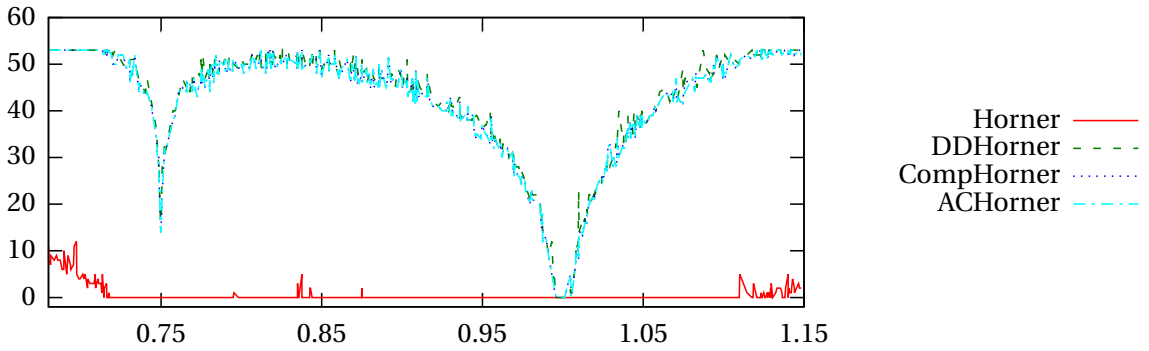


(b) $p_H(x)$ avec $x \in \{0,74995 : 0,75005\}$ et $x \in \{0,9935 : 1,0065\}$. Les ordonnées de gauche correspondent aux courbes des algorithmes COMPHORNER, ACHORNER et DDHORNER alors que celles de droite correspondent à HORNER.

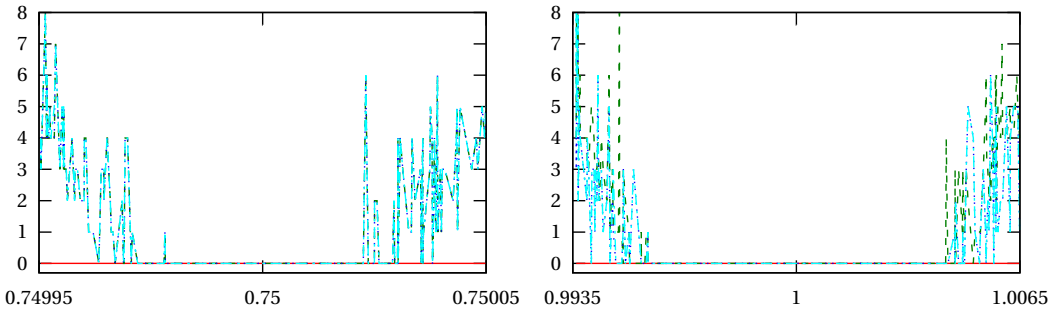
FIGURE 5.4 – Résultats de l'évaluation du polynôme $p_H(x) = (x-0,75)^5(x-1,0)^{11}$ avec l'algorithme HORNER et de ces versions *double-double* (DDHORNER), compensée (COMPHORNER) et compensée automatiquement (ACHORNER).

le calcul de *split(x)* aux lignes 19, 20 et 21). En effet, dans le code C.2 le calcul du SLIPT de x (algorithme 2.4) est répété à chaque itération de la boucle. Ce calcul étant indépendant des autres calculs effectués dans la boucle, il peut être retiré ce celle ci et déplacé avant pour n'être effectué qu'une seule fois. Néanmoins un outil de transformation de code serait à même d'effectuer cette optimisation et de générer un code optimisé similaire à COMPHORNER. De plus, dans le cas présent, le compilateur utilisé se charge déjà d'effectuer cette optimisation lors de la compilation du code C.2. Le tableau 5.3 présente les performances des algorithmes étudiés ci-dessus.

Les mesures effectuées avec PAPI mettent en évidence que les performances de l'algorithme compensé automatiquement ACHORNER sont, bien que légèrement moins bonnes,



(a) $p_H(x)$ avec $x \in \{0,68 : 1,15\}$.



(b) $p_H(x)$ avec $x \in \{0,74995 : 0,75005\}$ et $x \in \{0,9935 : 1,0065\}$.

FIGURE 5.5 – Nombre de bits corrects du résultat de l'évaluation du polynôme $p_H(x) = (x - 0,75)^5(x - 1,0)^{11}$ selon l'algorithme HORNER et de ces versions *double-double* (DDHORNER), compensée (COMPHORNER) et compensée automatiquement (ACHORNER).

semblables aux performances de l'algorithme COMPHORNER, compensé et optimisé manuellement. De plus, ce sont les algorithmes qui présentent les meilleures performances. Ces algorithmes compensés, dans cet environnement, procurent un niveau de parallélisme d'instruction au moins deux fois supérieur à celui de l'algorithme DDHORNER. Les mesures effectuées avec PERPI confirment que l'algorithme le plus performant est celui qui a été compensé et optimisé manuellement. Vient ensuite l'algorithme compensé automatiquement, qui possède presque 4 fois plus de parallélisme d'instruction que l'algorithme en *double-double* DDHORNER.

En conclusion, nous avons montré que l'algorithme compensé automatiquement présente le même niveau de précision que ces équivalents COMPHORNER et DDHORNER tout

	PAPI			PERPI		
	Instructions	Cycles	IPC	Instructions	Cycles	IPC
HORNER	42	57	0,73	76	37	2,05
COMPHORNER	532	277	1,99	566	62	9,12
DDHORNER	658	920	0,72	676	325	2,08
ACHORNER	553	303	1,82	581	77	7,54
AC/COMP	1,04	1,09	0,95	1,02	1,24	0,82
AC/DD	0,84	0,33	2,55	0,85	0,23	3,62

TABLE 5.3 – Mesures de performances des algorithmes HORNER, COMPHORNER, DDHORNER et ACHORNER (les chiffres obtenus avec PAPI sont les moyennes de 10^6 mesures réalisées dans l'environnement A.1).

en améliorant les performances par rapport à DDHORNER et s'approchant de celles de l'algorithme COMPHORNER.

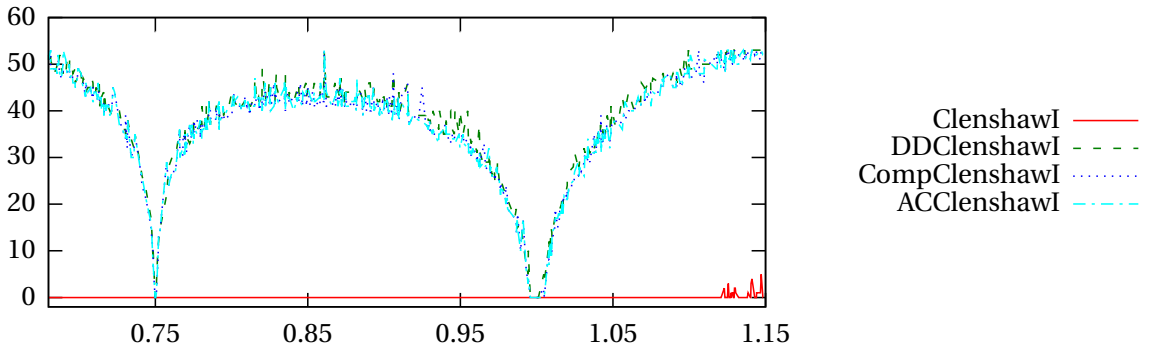
Enfin, pour plus d'informations, une description détaillée des algorithmes COMPHORNER et DDHORNER est présente dans la thèse de N. LOUVET [Lou07].

Le cas de l'algorithme CLENSHAWI

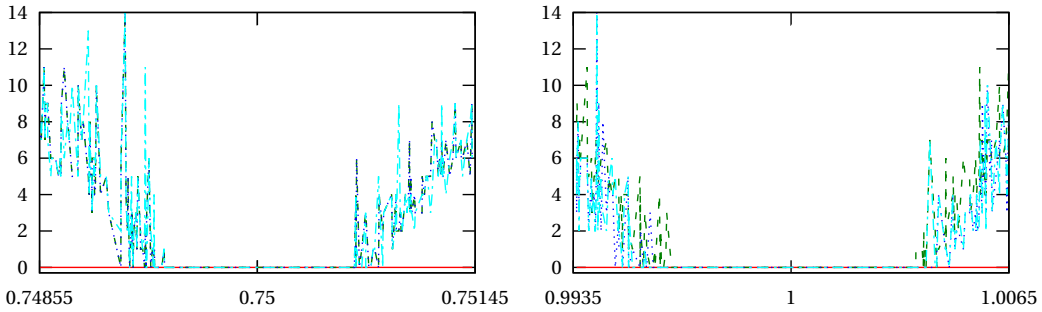
L'algorithme COMPCLENSHAWI [JBL+11] effectue l'évaluation polynomiale compensée suivant le schéma de TCHEBYCHEV. Considérons DDCLENSHAWI, la version *double-double* de l'algorithme 5.4.

Soit $p_C(x) = (x-0,75)^7(x-1,0)^{10}$ le polynôme considéré dans le cas présent. L'évaluation du polynôme se fait à l'aide de 512 données choisies aléatoirement entre 0,68 et 1,15 selon une distribution uniforme. Nous faisons les mêmes observations que pour l'algorithme HORNER, tant sur la qualité numérique que sur les performances de ces algorithmes. Aussi, nous ne présentons que brièvement les résultats portant sur l'étude de ce cas.

Les courbes de la figure 5.6 montrent le nombre de bits corrects des résultats des différentes variantes des algorithmes de CLENSHAWI. Sur les 512 évaluations, l'algorithme CLENSHAWI offre en moyenne 0,06 bits de précision contre 36,72 pour COMPCLENSHAWI, 38 pour DDCLENSHAWI et 36,71 pour ACCLENSHAWI. Ces résultats confirment que les algorithmes compensés COMPCLENSHAWI et ACCLENSHAWI améliorent la précision des résultats à l'instar de l'algorithme DDCLENSHAWI.



(a) $p_C(x)$ avec $x \in \{0,68 : 1,15\}$.



(b) $p_C(x)$ avec $x \in \{0,74855 : 0,75145\}$ et $x \in \{0,9935 : 1,0065\}$.

FIGURE 5.6 – Nombre de bits corrects du résultat de l'évaluation du polynôme $p_C(x) = (x - 0,75)^7(x-1,0)^{10}$ selon l'algorithme CLENSHAWI et de ces versions *double-double* (DDCLENSHAWI), compensée (COMPCLENSHAWI) et compensée automatiquement (ACCLENSHAWI).

Le tableau 5.4 présente les performances des algorithmes étudiés ci-dessus. Les mesures effectuées avec PAPI mettent en évidence que les performances de l'algorithme compensé manuellement sont légèrement moins bonnes que celles de l'algorithme compensé automatiquement. En effet, même s'il présente un IPC identique, il comporte aussi un plus grand nombre d'instructions et de cycles. Les mesures effectuées avec PERPI confirment celles effectuées avec PAPI. Cependant, l'environnement à joué ici un rôle favorable à l'algorithme ACCLENSHAWI, compensé automatiquement (et dont les nombres d'instructions et de cycles sont presque identiques aux performances de l'algorithme compensé manuellement), alors que sur machine idéale il présente de moins bonnes performances que l'algorithme COMPCLENSHAWI en terme d'ILP (chose que les mesures effectuées avec PAPI ne montrent pas). Néanmoins, ces algorithmes compensés, sur cet environnement, procurent un niveau de parallélisme d'instruction supérieur à celui de l'algorithme DDCLENSHAWI.

Ces observations sont confirmées par les mesures effectuées avec PERPI, et indiquent que les algorithmes compensés offrent, sur machine idéale, un niveau de parallélisme d'instruction entre 3,15 et 3,39 fois supérieur à l'algorithme *double-double*.

	PAPI			PERPI		
	Instructions	Cycles	IPC	Instructions	Cycles	IPC
CLENSHAWI	177	179	0,98	259	93	2,78
COMPCLENSHAWI	1 110	658	1,69	1 241	145	8,55
DDCLENSHAWI	1 552	1 955	0,79	1 697	672	2,52
ACCLENSHAWI	1 150	679	1,69	1 282	161	7,96
AC/COMP	1,04	1,03	1,00	1,03	1,11	0,93
AC/DD	0,74	0,35	2,13	0,75	0,23	3,15

TABLE 5.4 – Mesures de performances des algorithmes CLENSHAWI, COMPCLENSHAWI, DDCLENSHAWI et ACCLENSHAWI (les chiffres obtenus avec PAPI sont les moyennes de 10^6 mesures réalisées dans l'environnement A.1).

5.3.3 Synthèse des résultats

La figure 5.7 présente les ratios d'instructions, de cycles et d'IPC entre les algorithmes compensés automatiquement (AC) et ; d'une part, les algorithmes compensés tirés de la littérature (COMP), et d'autre part, les algorithmes *double-double* (DD). La figure 5.8 présente quant à elle, les différences de nombre de bits corrects entre ces mêmes algorithmes. Les données utilisées lors de ces mesures sont décrites dans le tableau 5.5. Le détail des résultats est présenté dans les tableaux C.1 et C.2 de l'annexe C.

Les résultats des figures 5.7 et 5.8 sont obtenus en faisant la moyenne des résultats des différentes évaluations de chaque algorithme suivant l'ensemble des données proposées au tableau 5.5. Par exemple, la différence AC-COMP du nombre de bits corrects de $Sum(d_1)$ représente la différence entre les moyennes des résultats de l'évaluation de l'algorithme SUMAC et de l'algorithme SUM2 sur l'ensemble des 32 jeux de données d_1. De plus, nous avons choisi les intervalles de valeurs pour les données x_1 et x_2 de telle façon que leur conditionnement soit favorable à ce que l'évaluation des polynômes en *binary64* permette des résultats ayant un nombre de bits exacts supérieur à 0 et inférieur à 53. Ceci permet aux versions compensés et *double-double* des algorithmes étudiés d'en améliorer la précision de manière significative (i.e. un conditionnement inférieur ou avoisinant 10^{16}).

La figure 5.7 montre que les performances des algorithmes compensés automatiquement sont proches de celles des algorithmes compensés manuellement. Ces algorithmes

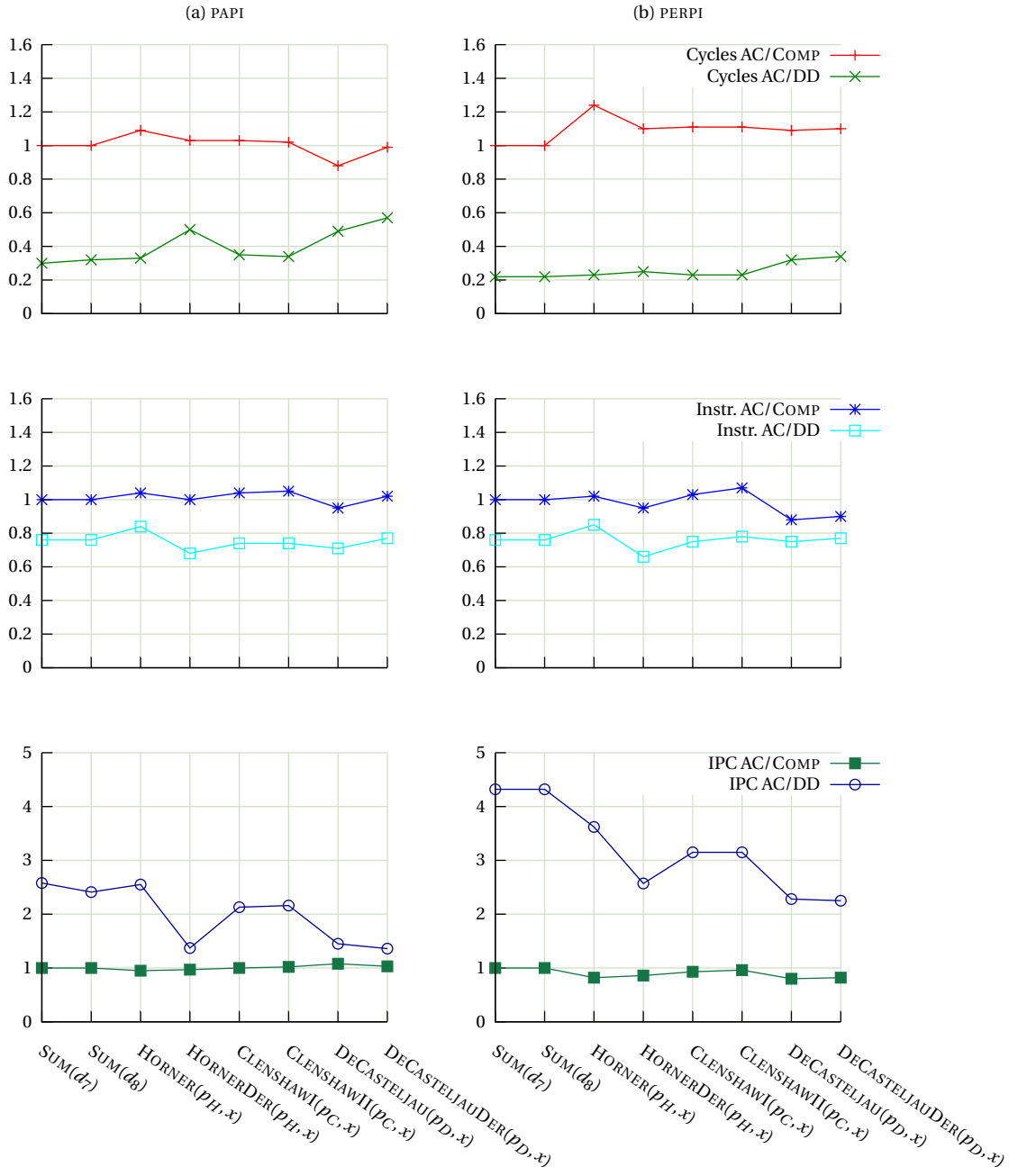


FIGURE 5.7 – Ratio des performances entre les algorithmes compensés automatiquement (AC) et ; d'une part, les algorithmes compensés présent dans la littérature (COMP), et d'autre part, les algorithmes *double-double* (DD) (les chiffres obtenus avec PAPI sont les moyennes de 10^6 mesures réalisées dans l'environnement A.1).

Données	Description
d_1	32 jeux de 10^4 valeurs de conditionnement 10^8
d_2	32 jeux de 10^5 valeurs de conditionnement 10^8
d_3	32 jeux de 10^6 valeurs de conditionnement 10^8
d_4	32 jeux de 10^4 valeurs de conditionnement 10^{16}
d_5	32 jeux de 10^5 valeurs de conditionnement 10^{16}
d_6	32 jeux de 10^6 valeurs de conditionnement 10^{16}
d_7	10^5 nombres flottants
d_8	10^6 nombres flottants
p_H	polynôme $p_H(x) = (x-0,75)^5(x-1,0)^{11}$
p_C	polynôme $p_C(x) = (x-0,75)^7(x-1,0)^{10}$
p_D	polynôme $p_D(x) = (x-0,75)^7(x-1,0)$
x_1	256 valeurs tirées uniformément dans $\{0,85 : 0,95\}$
x_2	256 valeurs tirées uniformément dans $\{1,05 : 1,15\}$
x	un nombre flottant dans $\{0,65 : 1,15\}$

TABLE 5.5 – Description des données relatives aux figures 5.7 et 5.8.

se révèlent toujours plus performant que les algorithmes *double-double*. Nous remarquons de plus, que la précision des divers algorithmes étudiés est similaire, comme le montre les différences de la figure 5.8. Toutefois, nous remarquons que les algorithmes *double-double* se révèlent être plus précis lorsque le nombre de conditionnement atteint 10^{16} dans le cas de la somme. En effet, dans les cas étudiés ici, les différences AC-DD peuvent atteindre 9 bits (en faveur des algorithmes *double-double*) sur les 53 bits de précision qu'offre le format double précision de l'arithmétique flottante.

5.4 Conclusion

Nous avons présenté dans ce chapitre une méthode pour compenser automatiquement certains algorithmes utilisant l'arithmétique flottante. De plus, au travers de diverses expériences, nous avons montré que l'automatisation de la compensation permet d'obtenir une précision équivalente à la précision des algorithmes compensés manuellement ainsi que des algorithmes *double-double* obtenus automatiquement. Enfin, nous pouvons conclure que la génération automatique d'algorithmes compensés permet d'améliorer les performances par rapport aux algorithmes *double-double*, allant même jusqu'à s'approcher des performances des algorithmes compensés manuellement. La méthode présentée dans ce chapitre tire donc le meilleur parti des deux méthodes, c'est-à-dire une génération automatique d'algorithmes précis et performants.

Nous allons voir dans le chapitre suivant, comment la compensation automatique peut

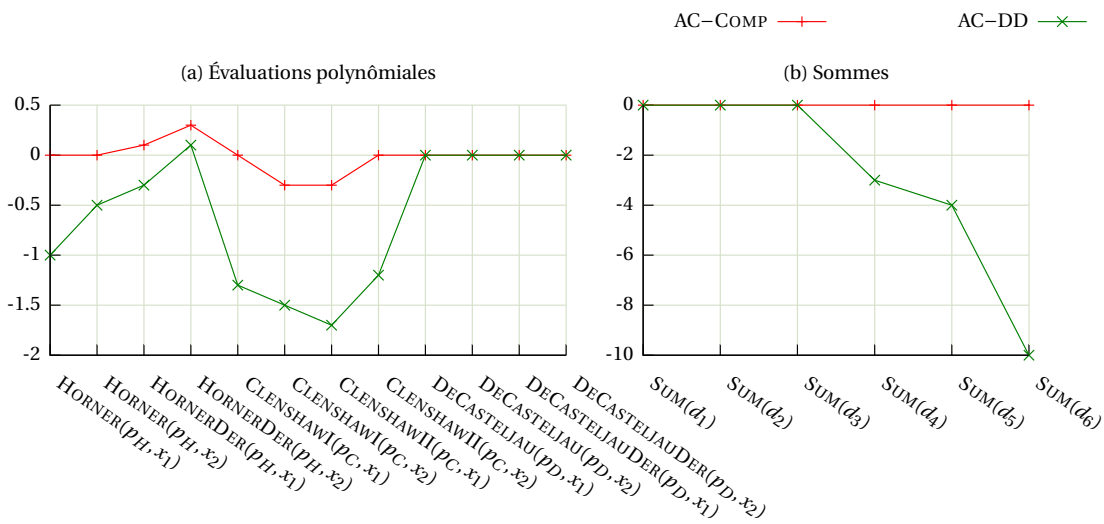


FIGURE 5.8 – Différences du nombre de bits corrects entre les algorithmes compensés automatiquement (AC) et ; d'une part, les algorithmes compensés présent dans la littérature (COMP), et d'autre part, les algorithmes *double-double* (DD).

être utilisée dans le cadre de l'optimisation multi-critères performance-précision. L'amélioration de la précision étant assurée par l'ajout des compensations, nous présenterons des moyens qui permettent de réduire l'effet de ces compensations sur les performances.

CHAPITRE 6

STRATÉGIES D'OPTIMISATION MULTI–CRITÈRES

6.1 Introduction

OPTIMISER un programme suivant plusieurs critères implique la création de certaines règles, qui spécifient les limites des interactions entre les différentes options (priorité, limitations, etc.). Ces règles correspondent à des compromis entre les différents critères. Les critères étudiés dans le cadre de nos travaux sont, d'un côté, la précision numérique, et de l'autre, les performances d'un programme. Cependant, ces critères sont quelque peu opposés, puisque améliorer la précision nécessite en règle générale d'ajouter des calculs supplémentaires, comme pour les méthodes utilisant les compensations ou les expansions [GLL09, ROO05, She97, FM01] (voir chapitre 2). À l'inverse, nous avons vu au chapitre 4, que l'amélioration des performances peut avoir des conséquences sur la précision des calculs [LMT10b]. Certains travaux [Dem92, LSCK09, BBDN10] mettent d'ailleurs en évidence ces compromis entre performances et précision.

L'optimisation a pour but d'améliorer au mieux les propriétés intrinsèques d'un programme. Par exemple, l'optimisation des performances d'un programme aura comme but de réduire son coût en ressources de calcul. Au sein des compilateurs, les optimisations s'appliquent de manière automatique via des techniques génériques que l'on applique sur un programme cible. De par sa nature, le processus d'optimisation, peut avoir un effet contraire au résultat recherché (voir par exemple le manuel de GCC [Fou14] et les options `-funroll-loops` ou `-funroll-all-loops`). Cependant, une simple vérification sur le gain

(du ou des critères optimisés) permet de mettre en garde l'utilisateur sur l'effet des optimisations effectuées. Il est de plus possible de délaisser certains critères pour en améliorer d'autres, coûte que coûte. L'option `-funsafe-math-optimizations` disponible dans le compilateur GCC en est un bon exemple. Elle permet au compilateur de réécrire les expressions arithmétiques sans tenir compte de règles telle que l'associativité des nombres flottants, afin d'améliorer les performances.

Objectifs : Les optimisations que nous souhaitons mettre en place doivent répondre aux spécifications suivantes :

- elles doivent être automatiques afin de s'adresser à un public principalement non-expert en ces domaines (i.e. les performances et la précision numérique) ;
- elles doivent améliorer la précision numérique des programmes ;
- et, elles doivent améliorer les performances des programmes par rapport aux autres méthodes existantes d'amélioration automatique de la précision.

Nos programmes optimisés doivent être plus performants que les programmes en *double-double*, car dans le cas contraire l'optimisation n'apporte rien. La précision, quant à elle, doit être aussi élevée que possible, suivant un compromis avec le temps de calcul spécifié par l'utilisateur. La précision maximale raisonnablement atteignable étant la précision des programmes en *double-double*. Les spécifications de ces compromis sont présentées au chapitre 7. Nous nous contentons ici de présenter les techniques de transformation de programmes développées afin de répondre à la recherche de tels compromis.

Tout au long de ce chapitre, la notion de « performances » signifie les performances en temps de calcul. Sauf indication contraire, nous entendons par performances, le nombre de cycles réel (voir chapitre 3) nécessaires à l'exécution d'un programme.

Un modèle simplifié de représentation de programme, nécessaire à la présentation des stratégies d'optimisation, est présenté à la section 6.2. La section 6.3 présente comment l'automatisation de la compensation y est intégrée. Les sections 6.4 et 6.5 présentent les stratégies d'optimisation prenant en compte les critères de performances et de précision. Enfin, avant de conclure, nous présentons quelques résultats d'expériences à la section 6.6.

6.2 Modèle de représentation de programme

Nous présentons un modèle de représentation de programme qui sera utilisé pour décrire les transformations définies dans les sections suivantes.

- La représentation de la figure 6.1(a) permet de **décrire graphiquement** des programmes et correspond à la trace d'exécution des instructions dans l'ordre de leurs évaluations. L'exemple présente en l'occurrence un programme comprenant une

boucle **for** de $i = 10$ itérations. Cette représentation correspond Dans cette représentation graphique, les itérations du corps d'une boucle sont séparées par une ligne en pointillés.

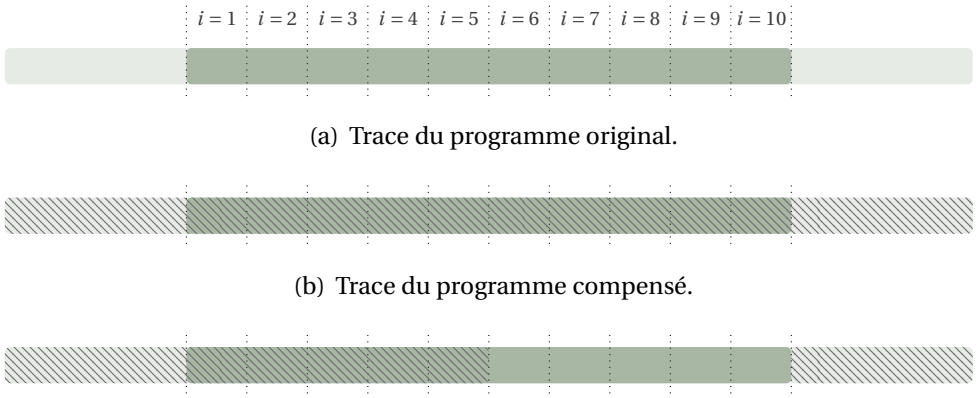
- Nous définissons de plus, la notion de **taille** qui intervient régulièrement dans le cas des boucles. La taille d'une boucle est son nombre d'itérations.

Les programmes seront présentés avec une syntaxe minimale, où les boucles sont exprimées à l'aide d'une boucle **for** (voir code 6.1).

Code 6.1 : Syntaxe de boucle *for* utilisée dans ce chapitre ◁

```
for i = min to max do
  /* instructions */
done
```

Sans perte de généralité, on convient que `min` est toujours inférieur à `max` et que l'itérateur de boucle avance par pas constant de 1. En pratique, notre outil est capable d'effectuer des transformations de programmes en langage C. Ces limitations sont présentées au chapitre 7. Des exemples concrets de code en langage C sont présentés à la section 6.6.



(a) Trace du programme original.

(b) Trace du programme compensé.

(c) Trace du programme partiellement compensé.

FIGURE 6.1 – Traces d'exécution d'un programme comprenant une boucle **for** de $i = 10$ itérations. Les espaces entre les lignes *pointillées* correspondent aux itérations de la boucle. Les *hachures* correspondent aux parties compensées.

6.3 Compensation automatique

L'optimisation de la précision se fait grâce à la compensation automatique présentée au chapitre 5. Nous avons constaté que l'ajout automatique des compensations affecte les performances. Pourtant leurs utilisations présentent l'avantage d'être plus performant que les expansions *double-double*, et ce pour une précision numérique équivalente.

De ce fait, compenser automatiquement un programme est une première optimisation. La précision numérique est doublée par rapport au programme original, et les performances sont moins dégradées qu'avec les expansions. La compensation automatique complète d'un code est donc utilisée lorsque le compromis performances–précision voulu ne transige pas sur la précision numérique. La figure 6.1 définie la présentation graphique d'un code compensé.

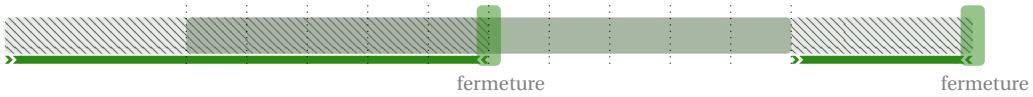
Notre méthode permet alors au maximum de doubler la précision car elle n'est pas encore adaptée à en fournir plus. Dans ce cas l'utilisation d'expansions de taille $k > 2$ serait requise. Cependant, nous pouvons prévoir comme perspective la compensation automatique de certains programmes afin d'obtenir k fois plus de précision (voir par exemple l'algorithme SUMK dans [ROO05]) et ainsi proposer une alternative automatique plus performante à des expansions de taille k. C'est pourquoi les problèmes traités par notre outil devront présenter un conditionnement adapté aux transformations proposées : le conditionnement permettra que des résultats calculés avec deux fois plus de précision soient exacts (i.e. l'ensemble des chiffres significatifs est correct).

Ainsi, pour obtenir encore de meilleures performances, il faut relever la contrainte de la précision. Pour ce faire, nous avons développés plusieurs stratégies qui compensent *partiellement* un programme original (voir sections 6.4 et 6.5). Par défaut, la compensation partielle s'applique uniquement au sein des boucles. Tous les calculs externes aux boucles sont automatiquement compensés. Enfin, la compensation partielle d'un programme ouvre une nouvelle problématique : comment propager les erreurs lorsque les calculs ne sont plus compensés ?

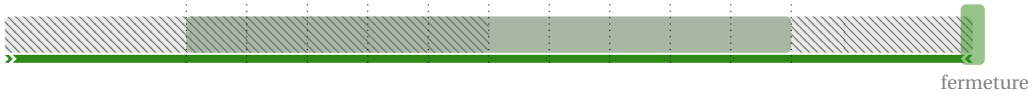
6.3.1 Propagation des erreurs

Afin d'enrichir les stratégies de compensation partielle, nous proposons deux moyens de propager les erreurs. Premièrement, la propagation p_1 propage les erreurs de façon classique, uniquement dans les parties de calculs compensés d'un programme. À la fin de chaque partie compensée, une fermeture (voir définition 5.2.2) est effectuée afin de poursuivre les calculs (non transformés) avec le résultat intermédiaire compensé obtenu par l'opération de fermeture. Ce type de propagation est présentée à la figure 6.2(a).

La propagation p_2 propage les erreurs tout le long des calculs du programme. Il n'y a donc pas de fermeture intermédiaire, et l'erreur est propagée dans les calculs, grâce aux



(a) Propagation et fermetures multiples (notée p_1).



(b) Propagation continue et fermeture unique (notée p_2).

FIGURE 6.2 – Types de propagation des erreurs (représentée par les traits pleins).

algorithmes 6.1 pour la somme et 6.2 pour le produit. Évidemment, la propagation p_2 implique un coût plus élevé en terme d'opérations flottantes et de latence par rapport à l'utilisation de p_1 car elle introduit des calculs supplémentaires. Ces coûts sont présentés au tableau 6.1. Ils correspondent au calcul de la propagation des erreurs héritées par les opérandes. Ce type de propagation est présentée à la figure 6.2(b).

Les algorithmes 6.1 et 6.2 effectuent respectivement la propagation automatique de la somme et du produit de nombres *auto-comp*. La propagation de l'erreur est similaire à la propagation présente dans les algorithmes de compensation automatique 5.1 et 5.2. Seule l'erreur de l'opération élémentaire concernée n'est pas prise en compte puisque l'opération n'est pas compensée.

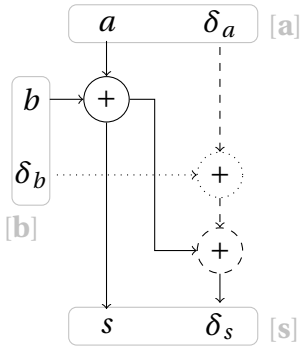
1. **fonction** AUTOPROP_TWOSUM($(a,\ \delta_a),\ (b,\ \delta_b)$)
2. $s \leftarrow RN(a+b)$ ▷ *opération élémentaire inchangée*
3. $\delta_s \leftarrow RN(\delta_a+\delta_b)$ ▷ *propagation des erreurs héritées*
4. **retourner** [**s**] $\leftarrow (s,\ \delta_s)$
5. **fin fonction**

Algorithme 6.1 – AUTOPROP_TWOSUM.

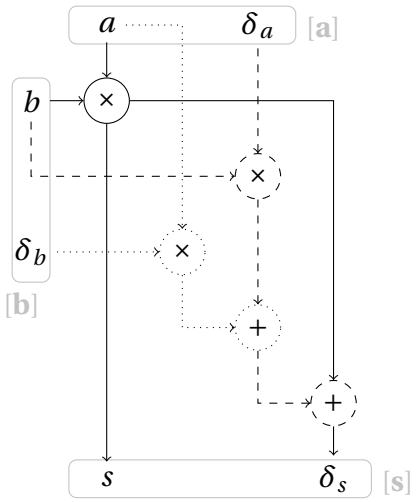
1. **fonction** AUTOPROP_TWOPRODUCT($(a,\ \delta_a),\ (b,\ \delta_b)$)
2. $s \leftarrow RN(a\times b)$ ▷ *opération élémentaire inchangée*
3. $\delta_s \leftarrow RN(a\times\delta_b+b\times\delta_a)$ ▷ *propagation des erreurs héritées*
4. **retourner** [**s**] $\leftarrow (s,\ \delta_s)$
5. **fin fonction**

Algorithme 6.2 – AUTOPROP_TWOPRODUCT.

La figure 6.3 donne la représentation graphique des algorithmes 6.1 et 6.2 ainsi que leurs variantes. Ces variantes sont utilisées en fonction de la nature des entrées des opérateurs, c'est-à-dire des nombres *auto-comp* $[x] = (x, \delta_x)$ ou des nombres flottants x. Les algorithmes de ces variantes sont obtenus en supprimant les calculs faisant intervenir les δ_x des entrées flottantes concernées des algorithmes 6.1 et 6.2.



(a) AUTOPROP_TWOSUM.



(b) AUTOPROP_TWOPRODUCT.

FIGURE 6.3 – Représentation des algorithmes 6.1 et 6.2 pour la propagation automatique de, (a) la somme, et (b) le produit. Les dessins en tirets ou en pointillés doivent être respectivement supprimés pour obtenir la représentation des algorithmes quand a ou b ne sont pas des nombres *auto-comp* mais de simples nombres flottants.

Le tableau 6.1 présente les performances en terme de nombre d'opérations flottantes et de latence idéale des algorithmes de propagation automatique des erreurs.

6.4 Transformation de boucle

Une grande partie des calculs est effectué au sein de boucles. Nous introduisons deux techniques de transformation de boucles permettant la compensation partielle de programmes. Ces transformations sont inspirées par le dépliage et la fission de boucles [DRV00]. Ces deux techniques appliquent la compensation suivant des répartitions différentes. Une première répartition *simple*, qui applique la compensation en une fois, sur une partie de la boucle. Et, une seconde répartition *intermittente*, qui applique la compensation en plusieurs fois, par intervalles réguliers au sein de la boucle.

Algorithme	δ_a	δ_b	*flop*		Latence	
			s	δ_s	s	δ_s
6.1 : AUTOPROP_TWOSUM	✓	✓	1	1	1	1
	✓	✗	1	0	1	0
	✗	✗	1	0	1	0
6.2 : AUTOPROP_TWOPRODUCT	✓	✓	1	3	1	2
	✓	✗	1	2	1	1
	✗	✗	1	0	1	0

TABLE 6.1 – Latence idéale et nombre d'opérations flottantes (*flop*) des algorithmes de propagation automatique d'erreurs 6.1 et 6.2 avec détails selon la nature des opérandes (présence de δ_a ou δ_b).

6.4.1 Par répartition simple

Le découpage par répartition simple (`rs`) d'une boucle b de taille t consiste à la diviser en 2 boucles. Le découpage est caractérisé par une position $\rho \in \{\top, \bot\}$ et par un ratio $0 \leq r \leq 1$ d'itérations à compenser.

La figure 6.4 décrit cette transformation au sein de notre modèle de représentation. Cette transformation permet de compenser les opérations flottantes par blocs de tailles variables, soit au début, soit à la fin de la boucle. La relation (6.1) définit la stratégie s relative à cette transformation.

$$s = \texttt{rs}(\rho,\ r) \tag{6.1}$$

où $\rho \in \{\top, \bot\}$ et r désigne le ratio d'itérations à compenser. Si $\rho = \top$ et $r = \frac{1}{2}$, alors la première moitié des itérations de la boucle est compensée alors que la seconde ne l'est pas. Inversement, si $\rho = \bot$ et $r = \frac{1}{2}$, c'est uniquement la seconde moitié de la boucle qui est compensée.

6.4.2 Par répartition intermittente

Le découpage par répartition intermittente (`ri`) d'une boucle b consiste à la diviser en $2n$ boucles, où $n > 0$. Le découpage est caractérisé par une position $\rho \in \{\top, \bot\}$, une taille t et une fréquence de répétition f d'itérations à compenser.

La figure 6.5 décrit cette transformation au sein de notre modèle de représentation. Cette transformation permet de compenser les opérations flottantes par intervalles réguliers de taille variable. La relation (6.2) définit la stratégie s relative à cette transformation.

$$s = \texttt{ri}(\rho,\ t,\ f) \tag{6.2}$$

Code 6.2 : Avant `rs`. ◁

```
for i = min to max do
  /* instructions */
done
```

Code 6.3 : Après `rs` (cas où $\rho = \top$). ◁

```
s = floor((max - min) * r) /* partie entière */
for i = min to s + min do
  /* instructions compensées */
done
for i = s + min + 1 to max do
  /* instructions */
done
```

Code 6.4 : Après `rs` (cas où $\rho = \bot$). ◁

```
s = floor((max - min) * (1 - r)) /* partie entière
    */
for i = min to s + min do
  /* instructions */
done
for i = s + min + 1 to max do
  /* instructions compensées */
done
```

FIGURE 6.4 – Transformation de boucle `for` par répartition simple (`rs`, pour $0 \leq r \leq 1$ le ratio des itérations à compenser et $\rho \in \{\top, \bot\}$ la position du bloc compensé).

où $\rho \in \{\top, \bot\}$ désigne la position des blocs de taille t itérations à compenser à intervalles réguliers de f itérations. C'est-à-dire que si $\rho = \top$ alors t itérations sont compensées à intervalles réguliers toute les f itérations. Les autres itérations restent inchangées. Inversement, si $\rho = \bot$, alors t itérations sont compensées à intervalles réguliers toutes les $f - t$ itérations.

La figure 6.6 donne cinq exemples de compensations partielles utilisant les transformations de boucle par répartition simple (cas (a) et (b)), et par répartition intermittente (cas (c), (d) et (e)).

Notons enfin que certains codes transformés par ces stratégies, pourront être plus lent que le code entièrement compensé, malgré un plus petit nombre d'opérations flottantes. En effet, les structures de contrôles supplémenaires introduites par ces transformations (dans notre cas : des boucles **for** et des tests de la forme `condition ? alors : sinon`), introduisent du calcul entier et réduisent ainsi le niveau de parallélisme du programme.

Code 6.5 : Avant `ri`. ◁

```
for i = min to max do
  /* instructions */
done
```

Code 6.6 : Après `ri` (cas où $\rho = \top$). ◁

```
for j = min to max do
  end1 = j + t - 1 > max ? j + t - 1 : max
  end2 = j + f - 1 > max ? j + f - 1 : max
  for i = j to end1 do
    /* instructions compensées */
  done
  for i = end1 + 1 to end2 do
    /* instructions */
  done
  j = j + f
done
```

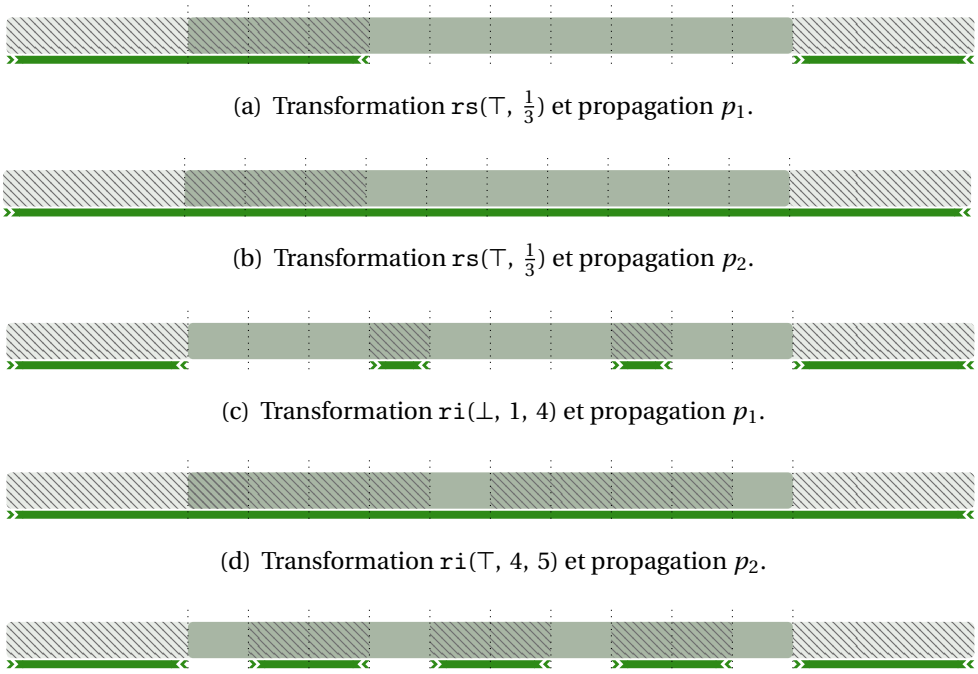
Code 6.7 : Après `ri` (cas où $\rho = \bot$). ◁

```
t = f - t
for j = min to max do
  end1 = j + t - 1 > max ? j + t - 1 : max
  end2 = j + f - 1 > max ? j + f - 1 : max
  for i = j to end1 do
    /* instructions compensées */
  done
  for i = end1 + 1 to end2 do
    /* instructions */
  done
  j = j + f
done
```

FIGURE 6.5 – Transformation de boucle `for` par répartition intermittente (`ri`, pour t le nombre d'itérations à compenser à la fréquence de f itérations et de la position $\rho \in \{\top, \bot\}$ des blocs compensés).

6.5 Sélection par précision

Nous introduisons maintenant, une première piste pour une méthode de sélection moins « statique », dans le sens où elle propose de sélectionner les opérations à compenser en fonction des erreurs qu'elles provoquent. De ce fait, en sélectionnant les n premières opérations causant les plus grandes erreurs, cette stratégie permet de compenser partiellement un code en transformant directement les parties du programme où sont situées les plus grandes erreurs.



(a) Transformation `rs`($\top$, $\frac{1}{3}$) et propagation p_1.

(b) Transformation `rs`($\top$, $\frac{1}{3}$) et propagation p_2.

(c) Transformation `ri`($\bot$, 1, 4) et propagation p_1.

(d) Transformation `ri`($\top$, 4, 5) et propagation p_2.

(e) Transformation `ri`($\bot$, 2, 3) et propagation p_1.

FIGURE 6.6 – Trace d'exécution des programmes transformés par les transformations de boucle, (`rs`) répartition simple et (`ri`) répartition intermittente.

Contrairement aux stratégies « par répartition » présentée précédemment, cette sélection par précision s'utilise uniquement avec la propagation p_2. La sélection a pour comme effet de transformer des opérations flottantes isolées les unes des autres au sein d'un même programme. La propagation p_1 est inutile dans ce cas car la sélection par précision transforme des calculs qui ne se suivent pas en général. La relation (6.3) définit la stratégie s relative à cette transformation, où n est le nombre d'opérations à compenser.

$$s = \mathtt{sp}(n) \tag{6.3}$$

La figure 6.7 présente un exemple de transformation utilisant cette sélection. Les parties hachurées représentent ici encore, les parties du programme qui on été compensées. L'opération 2 à l'intérieur d'une boucle, est compensée à chaque itération de celle-ci lors de l'exécution du programme.

La sélection par précision peut être mise en œuvre de différentes façons. La procédure que nous avons mis en place est présentée au travers de l'algorithme 6.3.

Entrée(s) : Un programme p, un ensemble D de jeux de données et n le nombre d'opérations élémentaires à retourner.

Sortie(s) : I l'ensemble de taille n des opérations élémentaires ayant provoquées les plus grandes erreurs absolues entre le programme p et son équivalent en *double-double*.

pour tout $d \in D$ **faire**

▷ *Exécution du programme original et du programme en double-double avec les données d.*

$\hat{P}_d \leftarrow$ l'ensemble des résultats intermédiaires des calculs flottants du programme p.

$P_d \leftarrow$ l'ensemble des résultats intermédiaires des calculs flottants du programme p en *double-double*. ▷ *Valeurs de références.*

pour tout $(\hat{i}, i) \in (\hat{P}_d, P_d)$ **faire**

$e^d_{\hat{i}} \leftarrow E_{\texttt{abs}}(\hat{i}_d)$ ▷ *l'erreur absolue entre le résultat original $\hat{i}_d$ et double-double de référence i.*

fin pour

fin pour

▷ *e contient maintenant les erreurs absolues engendrées par chaque opérations du programme p pour toutes les données d. L'étape suivante est de faire la moyenne pour chaque opérations des erreurs qu'elles ont engendrées.*

$e_{\text{moy}} \leftarrow \texttt{mean}(e)$

$e_{\text{moy}} \leftarrow \texttt{sort}(e_{\text{moy}})$ ▷ *On tri ensuite ces erreurs de la plus grande à la plus petite.*

pour tout $i \in \{1, n\}$ **faire**

$I \leftarrow \texttt{pop}(e_{\text{moy}})$ ▷ *Sélection des n premières opérations causant les plus grandes erreurs (en considérant que la fonction `pop` retourne l'opération du programme orignal ayant causée l'erreur concernée).*

fin pour

retourner I ▷ *Les algorithmes de compensation automatique seront appliquées sur ces opérations élémentaires dans le programme original.*

Algorithme 6.3 – Sélection par précision.

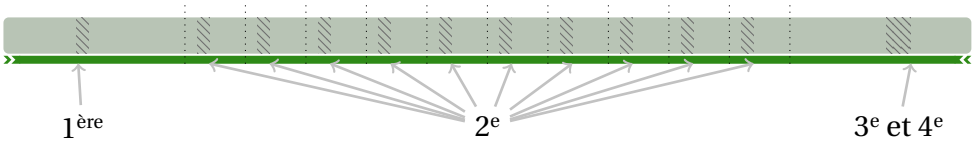


FIGURE 6.7 – Trace d'exécution d'un programme transformé par la sélection par précision (`sp`). L'exemple donné de transformation correspond à la sélection `sp(4)` (la propagation p_2 est la seule propagation autorisée pour cette transformation).

Notre implémentation propose une première solution « naïve » pour la sélection par précision. Cet aspect de transformation de programmes devrait faire l'objet d'études plus approfondies. Par exemple, pour des raisons d'optimums locaux, cette stratégie ne garantit pas que les plus grandes erreurs soient effectivement corrigées. De plus, corriger une erreur peut d'un côté en générer ou en faire émerger d'autres ailleurs. Rien ne garantit non plus, que cette erreur soit minimisée au cours du reste des calculs.

Il faut donc garder à l'esprit que, même si la sélection par précision est plus perspicace que les stratégies basées sur les transformations de boucle, il n'en faut pas moins vérifier ses effets. Toutefois cette sélection présente l'avantage de fournir un seul et unique programme par nombre d'erreurs à corriger, et ce, sans toucher à la structure de contrôle. À l'inverse du nombre potentiellement très grand de programmes qui sont générées par les transformations de boucles.

6.6 Résultats expérimentaux

Cette section présente quelques résultats[1] d'application sélective des compensations sur certains des algorithmes étudiés au chapitre 5. Plus particulièrement, nous allons montrer qu'il est possible, via les transformations présentées dans ce chapitre, d'améliorer les performances et la précision mais que le paramétrage de ces stratégies n'est pas évident a priori.

Dans cette section, « améliorer la précision », doit se comprendre comme : améliorer la précision numérique des résultats par rapport à l'algorithme original et « améliorer les performances », doit se comprendre comme : améliorer la performance, soit le nombre de cycle (i.e. le temps d'exécution), par rapport à la version entièrement compensée de l'algorithme étudié. Les codes remplissant ces critères feront alors parti des codes présentant des compromis encore plus poussés entre performances et précision. Nous verrons que la réponse à ces critères, est elle aussi indécidable a priori. Nous proposons au chapitre 7 une méthode permettant de nous aider dans ces choix : la synthèse de code.

[1] Tous les résultats présentés ont été obtenus dans l'environnement A.1.

Nous avons vu au chapitre précédent que la compensation systématique des opérations flottantes au sein d'un programme permet d'en améliorer la précision et de fournir des performances meilleures à ce qui est aujourd'hui atteignable en utilisant des méthodes automatiques, c'est-à-dire l'arithmétique *double-double* (pour une même précision). Ce premier résultat est déjà un compromis avec les performances, mais nous allons voir que nous pouvons encore faire mieux en sacrifiant une partie de la précision.

6.6.1 Exemples de programmes C transformés automatiquement

Les transformations de boucle

Le code 6.8 implémente l'algorithme 2.7 (SUM) en C. Le cœur du programme se situe aux lignes [15-17], le reste des instructions étant destinées à lire un fichier de données et à afficher le résultat de la somme. Ici aussi, nous pouvons compter $n-1$ opérations flottantes.

Code 6.8 : Algorithme SUM en C. ◁

```
Ligne 1  #include <stdio.h>
       - #include <stdlib.h>
       -
       - int main(int argc, char** argv) {
     5   int i, n=atoi(argv[1]);
       -   double s, tab[n];
       -   FILE *file;
       -
       -   file = fopen (argv[2], "rb");
    10   fread(&tab, sizeof(double), n, file);
       -   fclose (file);
       -
       -   s = tab[0];
       -
    15   for(i=1; i<n; i++) {
       -     s = s + tab[i];
       -   }
       -
       -   printf("%.16e\n", s);
    20   return 0;
       - }
```

Ce programme effectue donc la somme séquentielle de n nombres flottants. Comme nous avons pu le voir au chapitre précédent, la compensation automatique d'un tel programme permet d'effectuer des calculs avec le double de précision avec $7(n-1)+1$ opérations flottantes (soit de meilleures performances que les *double-double*). Nous allons

maintenant voir sur deux exemples l'effet des transformations `rs` et `ri` sur ce même programme[2].

La figure 6.8 présente deux exemples de transformations. Premièrement, une transformation par répartition simple paramétrée pour compenser 50% des premières itérations de la boucle en utilisant la propagation p_1, soit `rs`$(\top, \frac{1}{2})$ (code 6.9). Ensuite, une transformation par répartition intermittente paramétrée pour compenser une itération sur deux de la boucle en utilisant la propagation p_2, soit `ri`$(\top, 1, 2)$ (code 6.10).

Pour cet exemple, les codes transformés automatiquement, ont vu leur nombre de lignes doubler. Cependant le code reste relativement lisible et nous pouvons remarquer les parties suivantes.

- Pour le code 6.9 :
  - le cœur du programme se situe aux lignes [23-36], dont les lignes [24-30] correspondent à l'algorithme AUTOCOMP_TWOSUM ;
  - puisque la propagation p_1 est utilisée, une fermeture est présente aux lignes [32-33], puis la suite des calculs se fait entre nombres flottants avec les opérations élémentaires classiques ;
  - le nombre d'opérations flottantes atteint ici $\frac{7(n-1)+(n-1)}{2} + 1$, soit $4n-3$.
- Pour le code 6.10 :
  - le cœur du programme se situe aux lignes [22-38], dont les lignes [26-32] correspondent à l'algorithme AUTOCOMP_TWOSUM ;
  - puisque la propagation p_2 est utilisée, les opérations non compensées sont effectuées à l'aide des opérateurs de propagation automatique. Il s'agit ici en l'occurrence de l'algorithme AUTOPROP_TWOSUM présent aux lignes [35-36] ;
  - le nombre d'opérations flottantes atteint ici aussi $4n-3$.

Le nombre d'opérations flottantes est identique dans les deux cas (la moitié des opérations est compensée) mais la différence se situe au niveau des opérations elles-mêmes. Dans le premier cas, la moitié est compensée d'un seul tenant, alors que pour le second cas, c'est une opération sur deux qui l'est. Cependant ils ne présenteront certainement pas les mêmes performances, ni la même précision. Notez que les codes générés, comportent quelques opérations inutiles qui seront supprimées lors de la compilation (notre outil de génération de code n'élimine pas le corps mort).

[2] Nous aborderons plus en détail comment fonctionne la transformation du code C au chapitre 7.

Code 6.9 : Code 6.8, après la transformation $\texttt{rs}(\top, \frac{1}{2})$ et la propagation p_1. ◁

```
Ligne 1  int main(int argc, char** argv) {
  -        int i, n = atoi(argv[1]);
  -        double s;
  -        double tab[n] ;
  5        int ac_end;
  -        double s_dt;
  -        double __ac__tmp1;
  -        double __ac__tmp1_dt;
  -        double s_twosum;
  10       double s_ts1;
  -        double s_ts2;
  -        FILE * file;
  -
  -
  15
  -        file = fopen(argv[2], "rb");
  -        fread(&tab, sizeof( double ), n, file);
  -        fclose(file);
  -
  20       s = tab[0];
  -        s_dt = 0.0;
  -        ac_end = (int) n * 0.5;
  -        for(i = 1; i < ac_end; i++) {
  -          __ac__tmp1 = s;
  25         __ac__tmp1_dt = s_dt;
  -          s = __ac__tmp1 + tab[i];
  -          s_ts1 = s - tab[i];
  -          s_ts2 = s - s_ts1;
  -          s_twosum = (__ac__tmp1 - s_ts1) + (tab[i]
                 - s_ts2);
  30         s_dt = s_twosum + __ac__tmp1_dt;
  -        }
  -        s += s_dt;
  -        s_dt = 0.0;
  -        for(i = ac_end; i < n; i++) {
  35         s = s + tab[i];
  -        }
  -
  -
  -
  40       printf("%.16e\n", s + s_dt);
  -        return 0;
  -      }
```

Code 6.10 : Code 6.8, après la transformation $\texttt{ri}(\top, 1, 2)$ et la propagation p_2. ◁

```
int main(int argc, char** argv) {                          Ligne 1
  int i, n = atoi(argv[1]);                                  -
  double s;                                                  -
  double tab[n] ;                                            -
  int ac_i;                                                  5
  int ac_end2;                                               -
  int ac_end3;                                               -
  double s_dt;                                               -
  double __ac__tmp1;                                         -
  double __ac__tmp1_dt;                                      10
  double s_twosum;                                           -
  double s_ts1;                                              -
  double s_ts2;                                              -
  FILE * file;                                               -
                                                             15
  file = fopen(argv[2], "rb");                               -
  fread(&tab, sizeof( double ), n, file);                    -
  fclose(file);                                              -
                                                             -
  s = tab[0];                                                20
  s_dt = 0.0;                                                -
  for(ac_i = 1; ac_i < n; ac_i = ac_i + 2) {                 -
    ac_end2 = ac_i + 1 < n ? ac_i + 1 : n;                   -
    ac_end3 = ac_i + 2 < n ? ac_i + 2 : n;                   -
    for(i = ac_i; i < ac_end2; i++) {                        25
      __ac__tmp1 = s;                                        -
      __ac__tmp1_dt = s_dt;                                  -
      s = __ac__tmp1 + tab[i];                               -
      s_ts1 = s - tab[i];                                    -
      s_ts2 = s - s_ts1;                                     30
      s_twosum = (__ac__tmp1 - s_ts1) + (tab[i               -
          ] - s_ts2);
      s_dt = s_twosum + __ac__tmp1_dt;                       -
    }                                                        -
    for(i = ac_end2; i < ac_end3; i++) {                     -
      s = s + tab[i];                                        35
      s_dt = s_dt;                                           -
    }                                                        -
  }                                                          -
                                                             -
  printf("%.16e\n", s + s_dt);                               40
  return 0;                                                  -
}                                                            -
```

FIGURE 6.8 – Exemples de transformations automatiques de code sur l'algorithme de sommation SUM (voir algorithme 2.7 et code 6.8). Le code 6.9 est le résultat de la transformation $\texttt{rs}(\top, \frac{1}{2})$ (avec la propagation p_1) et le code 6.10 est le résultat de la transformation $\texttt{ri}(\top, 1, 2)$ (avec la propagation p_2) du code 6.8.

La sélection par précision

Le code 6.11 est un extrait de l'algorithme 5.4 (CLENSHAWI) expurgé des instructions ici inutiles : déclarations, initialisations, gestion des entrées et sorties. Le nombre d'opérations flottantes est de $4(n-1)+3$. Avec notre outil, la version complètement compensée contient $2(n-1)+1$ AUTOCOMP_TWOPRODUCT[3], $2(n-1)+2$ AUTOCOMP_TWOSUM[4] et une fermeture pour un nombre total d'opérations flottantes égal à $46(n-1)+36$.

Le code 6.12 représente ce que donne la transformation par sélection de la précision paramétrée pour compenser les deux opérations causant la plus forte erreur telle que définie à la section 6.5, soit `sp(2)`. Pour cet exemple, le nombre de lignes a plus que triplé en comparaison avec le programme original. Le code grandit d'autant plus qu'il y a d'opérations flottantes élémentaires. Bien que toujours lisible, le code devient de moins en moins accessible.

- Les lignes [1-21] contiennent la boucle du cœur de calcul du programme où deux opérations ont été transformées par la sélection par précision. Considérons qu'il s'agisse des opérations suivantes :
  - `__ac_4 * b[j+1]` à la ligne 4 ;
  - et `__ac_3 - b[j+2]` à la ligne 15.

  Ces opérations élémentaires ont respectivement été remplacées par les algorithmes AUTOCOMP_TWOPRODUCT aux lignes [4-14] et AUTOCOMP_TWOSUM aux lignes [15-19].
- Le reste des opérations est quant à lui remplacé par les algorithmes de propagation automatique AUTOPROP_TWOSUM et AUTOPROP_TWOPRODUCT aux lignes [2-3,20-21,24-32].
- Une fermeture est présente à la ligne 34.

Le programme généré contient alors un algorithme AUTOCOMP_TWOPRODUCT, un algorithme AUTOCOMP_TWOSUM et deux algorithmes de propagation automatique par itération de la boucle. Le calcul extérieur à la boucle étant remplacé par des algorithmes de propagation automatique, le nombre total d'opérations flottantes s'élève ici à $31(n-1)+6$.

6.6.2 Résultats expérimentaux entre performances et précision

Les figures 6.9, 6.10 et 6.11 exhibent les résultats de transformations de codes selon quelques stratégies de transformation. Ces résultats concernent la précision, en nombre de bits exacts, et les performances, en nombre de cycles, mesurées avec PAPI (voir section 3.4 du chapitre 3) dans l'environnement A.1. Ces mesures sont exprimés comparativement

[3] Le nombre d'opérations peut varier selon la nature des entrées.

Code 6.11 : Algorithme CLENSHAWI en C (extrait). ◁

```
Ligne 1  for(j = n; j >= 1; j--) {
-          b[j] = 2 * x * b[j+1] - b[j+2] + P[j];
-        }
-
5        b[0] = x * b[1] - b[2] + P[0];
-
-        res = b[0];
-
-        printf("%.16e␣%.16e\n", x, res);
```

Code 6.12 : Transformation automatique du code 6.11, d'après la transformation sp(2) (extrait). ◁

```
Ligne 1  for(j = n; j >= 1; j--) {
-          __ac_4 = 2 * x;
-          __ac_4_dt = 0.0;
-          __ac_3 = __ac_4 * b[j + 1];
5          __ac_4_spl_1 = 134217729 * __ac_4;
-          __ac_4_spl_2 = __ac_4 - __ac_4_spl_1;
-          __ac_4_spl_h = __ac_4_spl_1 + __ac_4_spl_2;
-          __ac_4_spl_l = __ac_4 - __ac_4_spl_h;
-          b_spl_1 = 134217729 * b[j + 1];
10         b_spl_2 = b[j + 1] - b_spl_1;
-          b_spl_h = b_spl_1 + b_spl_2;
-          b_spl_l = b[j + 1] - b_spl_h;
-          __ac_3_twoprod = ((((-__ac_3) + (__ac_4_spl_h * b_spl_h)) + (__ac_4_spl_h * b_spl_l)) + (
               __ac_4_spl_l * b_spl_h)) + (__ac_4_spl_l * b_spl_l);
-          __ac_3_dt = __ac_3_twoprod + ((__ac_4_dt * b[j + 1]) + (b_dt[j + 1] * __ac_4));
15         __ac_2 = __ac_3 - b[j + 2];
-          __ac_2_ts1 = __ac_2 - (-b[j + 2]);
-          __ac_2_ts2 = __ac_2 - __ac_2_ts1;
-          __ac_2_twosum = (__ac_3 - __ac_2_ts1) + ((-b[j + 2]) - __ac_2_ts2);
-          __ac_2_dt = __ac_2_twosum + (__ac_3_dt + (-b_dt[j + 2]));
20         b[j] = __ac_2 + P[j];
-          b_dt[j] = __ac_2_dt;
-        }
-
-        __ac_1 = x * b[1];
25       __ac_1_dt = b_dt[1] * x;
-        __ac_0 = __ac_1 - b[2];
-        __ac_0_dt = __ac_1_dt + (-b_dt[2]);
-        b[0] = __ac_0 + P[0];
-        b_dt[0] = __ac_0_dt;
30       res = b[0];
-        res_dt = b_dt[0];
-
-        printf("%.16e␣%.16e\n", x, res + res_dt);
```

à celles obtenues en arithmétique *double-double*. La précision et les performances sont exprimées à l'aide de ratios :

$$r_\alpha = \frac{\text{résultats en } \textit{auto-comp}}{\text{résultats en } \textit{double-double}}. \tag{6.4}$$

Ainsi, les meilleurs résultats concernant la précision sont obtenus lorsque $r_{\text{bits exacts}}$ tend vers 1, et, lorsque r_{cycles} tend vers 0 pour les performances.

Sauf pour certains cas signalés dans le tableau 6.2), les jeux de données utilisés ont été choisis afin que les résultats en arithmétique *double-double* comportent 53 bits corrects. Ainsi, doubler la précision du calcul fournit des résultats exacts[4]. Les mesures présentées sont obtenues à partir de jeux de données, c'est-à-dire que les résultats sont des moyennes sur l'ensemble des résultats obtenus par chaque donnée contenue dans ceux-ci.

Données	Description	Précision en *double-double*
d_1	32 jeux de 10^4 valeurs de conditionnement 10^8	
d_2	32 jeux de 10^5 valeurs de conditionnement 10^8	53 (bits exacts)
d_3	32 jeux de 10^6 valeurs de conditionnement 10^8	
d_4	256 valeurs tirées uniformément dans $\{0,6:0,7\}$	53 pour p_H et 52 pour p_C
d_5	256 valeurs tirées uniformément dans $\{0,8:0,9\}$	49 pour p_H et 43 pour p_C
d_6	256 valeurs tirées uniformément dans $\{1,8:1,9\}$	53
p_H	polynôme $p_H(x) = (x-0,75)^5(x-1,0)^{11}$	voir d_4, d_5 et d_6
p_C	polynôme $p_C(x) = (x-0,75)^7(x-1,0)^{10}$	voir d_4, d_5 et d_6

TABLE 6.2 – Description des données relatives aux figures 6.9, 6.10 et 6.11 avec nombre moyen de bits exacts (maximum 53 en *binary64*).

Cas de l'algorithme de sommation La figure 6.9 présente les ratios de performances et de précision entre les codes transformés suivant des stratégies de transformation et le code en *double-double* dans le cas de l'algorithme 2.7 SUM. Une sélection de quinze stratégies choisies de façon arbitraire est étudiée dans ce cas (le choix des stratégies sera étudiée au chapitre 7).

Cette sélection comporte des stratégies par répartition simple (`rs`) et par répartition intermittente (`ri`), paramétrées pour compenser 50% et 80% (pour les `rs`, ou 50/100 et 80/100 pour les `ri`) des calculs selon les deux propagations d'erreurs possibles. Une sélection par précision (`sp`) est aussi présente et paramétrée pour transformer une opération

[4]Dans l'état actuel de nos recherches, nous sommes uniquement capable de doubler la précision à l'instar des *double-double*. Une perspective serait de faire plus que doubler, voir section 6.3.

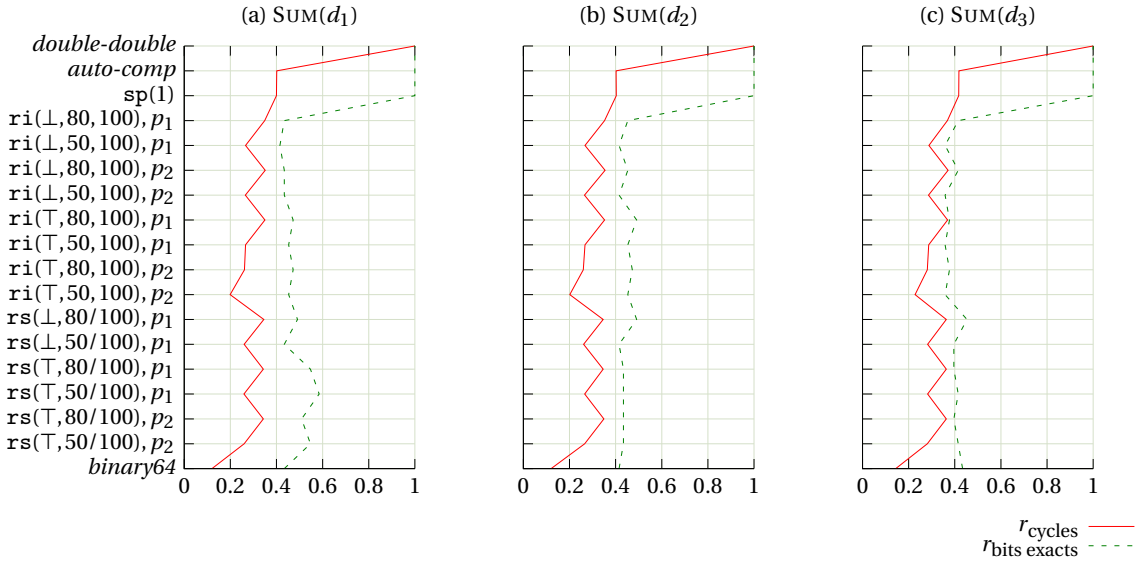


FIGURE 6.9 – SUM (algorithme 2.7, code 6.8) : Ratios r_{cycles} et $r_{\text{bits corrects}}$ (vs. *double-double*) des stratégies de compensation partielle (idéal lorsque $r_{\text{cycles}} \to 0$ et $r_{\text{bits corrects}} \to 1$).

élémentaire. En effet, l'algorithme de sommation ne contient qu'une seule opération flottante répétée n fois à l'aide d'une boucle. La stratégie par sélection de la précision sp(1) est alors équivalente à la transformation complète.

La première observation que nous faisons de la figure 6.9 concerne les performances. Dans les trois cas présentés (a), (b) et (c), les ratios de performances restent les mêmes quelques soient le nombre de termes à sommer. Ces résultats confirment ce que l'on pouvait attendre de ces transformations : elles sont effectuées sur le code source original et sont donc indépendantes des données. De plus, l'algorithme étant indépendant de la précision des calculs, le ratios de cycles entre le code *double-double* et les codes transformés suivants une stratégie choisie restent les mêmes quelque soit le nombre de données à traiter.

D'un autre côté, les résultats concernant la précision sont quant à eux sensibles aux données. De ce fait l'effet d'une stratégie sur la précision finale des résultats est peu intuitive. Même avec de grandes connaissances sur l'algorithme et les données nous ne pouvons avoir que des intuitions sur la nature des résultats. Comme vu au chapitre précédent, la compensation complète du code de l'algorithme 2.7 permet de retrouver la précision disponible à l'aide des *double-double* pour un nombre de cycles réduit de 60%. La compensation partielle permet de réduire encore le nombre de cycles en « sacrifiant » la précision. Pour l'exemple de la somme, nous pouvons constater que dans le cas (a), ce sont

les stratégies de répartition simple qui semblent s'en sortir le mieux. La répartition optimale, c'est-à-dire le plus de précision pour le moins de cycles, soit l'écart entre les courbes $r_{\text{bits exacts}}$ et r_{cycles}, dans ce cas est la répartition $\texttt{rs}(\top, 50/100)$ utilisant la propagation p_1. Elle permet de supprimer environ 70% de cycles par rapport à la version *double-double* tout en permettant en moyenne 60% de précision possible (contre 85% de cycles en moins pour 44% de précision pour le programme original). Parmi les stratégies étudiées, nous pourrions penser que plus une stratégie compense un grand nombre d'opérations, plus l'amélioration de la précision est importante. Par exemple, la stratégie $\texttt{rs}(\top, 80/100)$ qui compense plus de calculs que $\texttt{rs}(\top, 50/100)$ pourrait présenter une meilleure précision. Ce n'est cependant pas le cas dans cet exemple.

Les stratégies par répartition intermittente ici observées, bien qu'efficace au niveau des performances (pour certaines), ne permettent pas de gagner de précision. Nous rencontrons même certains cas où elles la détériorent. Ces effets sont d'autant plus importants lorsque l'on traite des données de plus en plus nombreuses. Les stratégies concernées deviennent alors complètement inappropriées car elles diminuent la précision des calculs.

Cas de l'algorithme HORNER La figure 6.10 présente les ratios de cycles et de précision des programmes (implémentant l'algorithme 5.3 HORNER) transformés automatiquement.

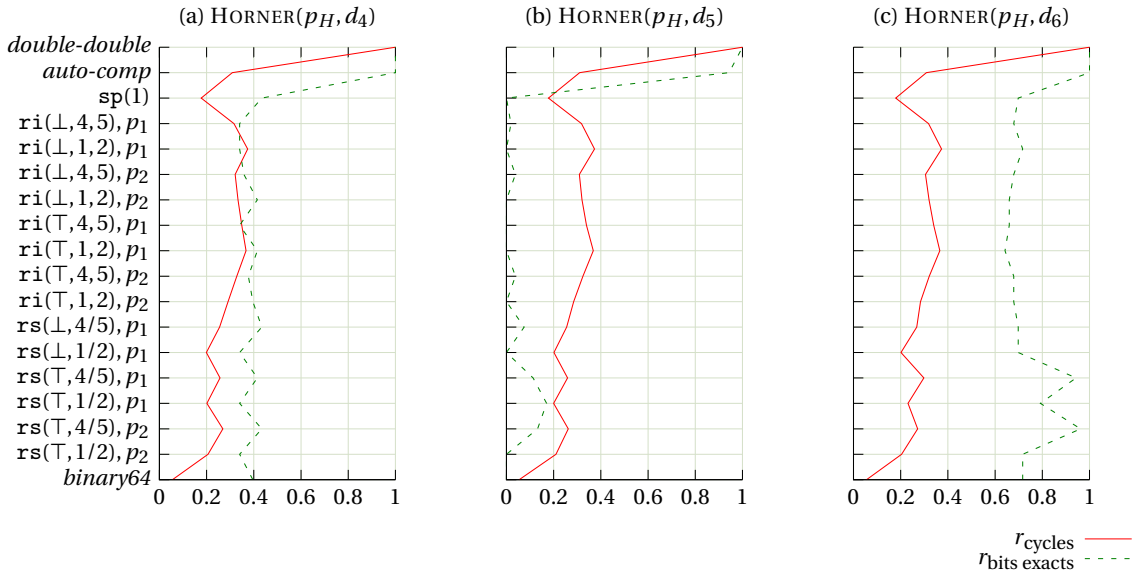


FIGURE 6.10 – HORNER (algorithme 5.3) : Ratios r_{cycles} et $r_{\text{bits corrects}}$ (vs. *double-double*) des stratégies de compensation partielle (idéal lorsque $r_{\text{cycles}} \to 0$ et $r_{\text{bits corrects}} \to 1$).

Les stratégies utilisées ici ont elles aussi été choisies arbitrairement. Les stratégies de transformation par répartition simple (`rs`) génèrent des programmes compensant 1/2 des opérations selon des paramètres de position des compensations $\top$ et $\bot$ ou de propagation d'erreur p_1 et p_2 différents (les stratégies par répartition intermittente `ri` compensant 4/5 des opérations).

Concernant les performances, nous faisons des observations similaires au cas précédent de l'algorithme de sommation. Le ratio du nombre de cycles r_{cycles} présente les mêmes caractéristiques quelque soit les données utilisées. Là encore, la compensation automatique complète permet de bons résultats. Ces programmes transformés présentent le même niveau de précision que les algorithmes en *double-double* pour seulement 30% des performances de celui-ci. Les programmes partiellement compensés, permettent une amélioration de la précision moindre pour un nombre de cycle encore plus réduit.

Du côté de la précision, nous constatons les faits suivants pour les différents cas exposés de la figure 6.10.

Cas (a). Les stratégies choisies ne permettent pas l'émergence d'une solution satisfaisant un compromis performances–précision avec les données d_4. En effet, la plupart des stratégies ont un effet, soit négatif, soit minime sur l'amélioration de la précision, et ce, pour des gains de performances quasi-nuls en comparaison avec le programme entièrement transformé. Cependant, nous observons que la stratégie `sp`(1) présente un petit gain de précision pour des performances proche des performances du programme original.

Cas (b). Ce cas illustre les limites actuelles de notre approche. Les données d_5 sont mal conditionnées, et de fait les résultats en *double-double* sont eux-mêmes entachés d'erreurs (voir tableau 6.2). Néanmoins, les résultats *double-double* sont considérés exacts et servent de référence pour le calcul de la précision des programmes transformés par notre outil. Nous observons alors comment se comportent les différentes stratégies malgré un conditionnement trop grand. Effectivement, nous observons que les stratégies par répartition simple (`rs`) semblent être à même d'améliorer la précision. Par exemple, la stratégie `rs`($\top$,1/2) avec la propagation p_1 permet d'obtenir 18% de la précision du programme en *double-double* pour 20% de son nombre de cycles.

Cas (c). Les programmes transformés suivant ces stratégies, étudiés conjointement avec les données d_6, permettent de meilleurs compromis performances–précision que dans les cas précédents. Le programme sans transformation dispose de 70% de la précision du programme en *double-double*. De ce fait, il est plus facile de générer des programmes plus performants, tout en gardant un maximum de précision. La figure montre quelques stratégies qui permettent presque la même précision que le *double-double* tout en permettant de légers gain côté performances par rapport au programme transformé entièrement compensé.

Enfin, là encore, une classe de stratégies permettant des compromis entre performances et précision n'émerge pas pour ce problème. Le choix a priori d'une stratégie est difficile, non intuitive et dépend de l'environnement et des ressources utilisés.

Cas de l'algorithme CLENSHAWI La figure 6.11 présente les ratios de cycles et de précision des programmes (implémentant l'algorithme 5.4 CLENSHAWI) transformés automatiquement.

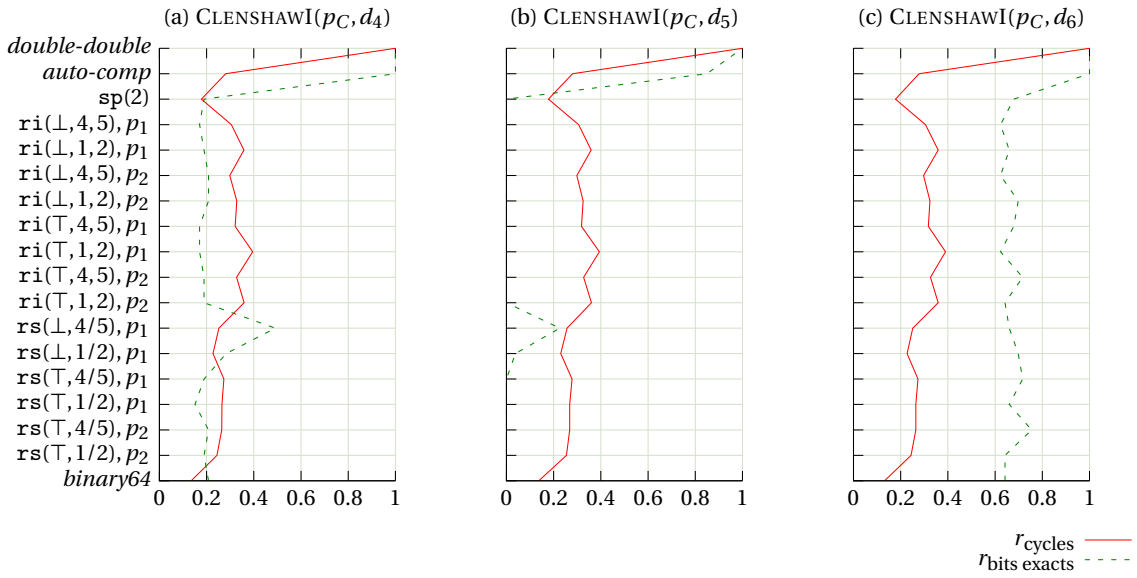


FIGURE 6.11 – CLENSHAWI (algorithme 5.4, code 6.11) : Ratios r_{cycles} et $r_{\text{bits corrects}}$ (vs. *double-double*) des stratégies de compensation partielle (idéal lorsque $r_{\text{cycles}} \to 0$ et $r_{\text{bits corrects}} \to 1$).

Pour les mêmes raisons évoquées lors du cas précédent, nous faisons des observations similaires, tant au niveau des performances que de la précision dans le cas du présent algorithme. Nous ne détaillerons alors pas les différents cas étudiés dans la figure 6.11 mais nous en tirons cependant les mêmes conclusions.

Interprétation générale des résultats De part, les différentes expériences réalisées dans cette section, nous pouvons affirmer que le choix des stratégies est sensible à l'algorithme utilisé mais surtout aux données que l'on traite. Il est de plus intuitivement difficile de déterminer a priori la stratégie qui répondra le mieux aux attentes d'un utilisateur. L'architecture de la machine ayant des conséquences sur les performances, celle-ci joue aussi un rôle dans le choix et le paramétrages des stratégies. C'est pourquoi le chapitre 7 présente une première approche dans le choix automatique de ces stratégies.

6.7 Conclusion

Nous avons présenté dans ce chapitre trois méthodes de transformation de code permettant la génération de programmes partiellement compensés. Ces méthodes sont : la répartition simple (`rs`), la répartition intermittence (`ri`) et la sélection par précision (`sp`). Ces méthodes sont une première proposition à la transformation de code, et d'autres techniques pourront faire le sujet de travaux futurs. Une des priorités pourrait concerner la méthode de sélection par précision qui peut être améliorée.

Nous avons de plus, présenté quelques résultats d'application partielle des compensations via les diverses méthodes présentées dans ce chapitre. Il s'est avéré qu'il est a priori difficile de décider quelle stratégie utiliser pour optimiser les performances et la précision. En effet, ces critères dépendent de nombreux facteurs, tels que les algorithmes transformés, les données utilisées, ou encore les architectures employées... C'est pourquoi, la recherche d'un programme transformé répondant à un certain compromis entre les performances et la précision doit être, elle aussi, automatisée.

Une première solution pour pallier ce problème est proposée au chapitre suivant où nous allons présenter comment rechercher un « bon » compromis performances–précision parmi l'ensemble des moyens que nous avons développés. Cette solution est la *synthèse de code*.

CHAPITRE 7

SYNTHÈSE DE CODE : RECHERCHE D'UN COMPROMIS TEMPS–PRÉCISION

7.1 Introduction

TRANSFORMER un code source afin d'en fournir une version améliorant la précision tout en tentant de contrôler l'impact sur les performances, nécessite l'application des stratégies d'optimisation développée au chapitre 6. Il présente plusieurs stratégies de transformation de programme permettant la compensation partielle suivant des critères de temps et de précision. Ce même chapitre a permis de conclure que le choix de la transformation à appliquer, dépend de différents facteurs a priori difficiles à déterminer car dépendants de l'environnement et des données. Comme ces choix ont différentes répercussions sur les critères que nous tentons d'améliorer : le temps de calcul et la précision numérique, la recherche d'un programme transformé répondant aux critères désirés, et de plus dans un temps raisonnable, ne peut se faire qu'automatiquement.

Nous présentons les travaux mis en place afin d'effectuer la sélection automatique d'un programme partiellement compensé, optimisé selon des critères de performances et de précision, via la *synthèse de code* [SGF10, Gul10]. Cette technique rend possible la recherche automatique d'un programme répondant de façon satisfaisante à des critères de précision et de performances. Néanmoins, comme de nombreuses techniques d'optimisation, celle-ci peut échouer, soit à cause d'une faiblesse dans le processus de synthèse, soit parce qu'une solution optimisée n'existe pas. De façon générale, les différents facteurs inhérents à la synthèse de code sont : (i) des spécifications reflétant l'expression des at-

tentes de l'utilisateur ; (ii) un espace de recherche noté E, c'est-à-dire un ensemble de programmes transformés sur lequel opérer ; (iii) et des techniques de recherche permettant de trouver un programme selon les spécifications souhaitées.

Ce chapitre est organisée de la manière suivante. La section 7.2 est consacrée à la synthèse de code et en présente les facteurs qui la composent : les spécifications, l'espace et les techniques de recherche. Nous y décrivons les moyens sélectionnés pour la réalisation de ces différentes étapes. Plus particulièrement, la section 7.2.3 explique comment sont exprimés les compromis entre performances et précision. De plus, cette section explique comment les résultats sont traités afin de permettre le choix du « bon » programme transformé selon les attentes de l'utilisateur. La section 7.3 présente les outils que nous avons développés pour répondre aux problématiques de la synthèse de code. *Les résultats des expériences effectuées grâce à la synthèse de code seront présentés au chapitre 8.*

7.2 Synthèse de code

La synthèse de code consiste à produire un code à partir des attentes d'un utilisateur, exprimées sous la forme de spécifications [Gul10]. Les synthétiseurs réalisent des recherches sur un ensemble de codes pour déterminer celui qui répond le mieux aux spécifications. Trois aspects interviennent alors dans la réalisation de la synthèse, soit autant d'étapes que nous allons décrire dans les sections suivantes. Les spécifications ou les attentes de l'utilisateur sont traitées à la section 7.2.1. L'aspect concernant l'espace de recherche est présenté à la section 7.2.2. Enfin, la section 7.2.3 présente notre première approche dans les techniques de recherche du programme répondant le mieux aux spécifications de l'utilisateur.

7.2.1 Spécifications ou attentes d'un utilisateur

Un de nos buts est de permettre à l'utilisateur, d'améliorer des programmes écrits en C, sans qu'il ait pour autant de grandes connaissances en arithmétique flottante. La synthèse de code permet d'automatiser la recherche d'un tel programme optimisé. Pour améliorer son code C, l'utilisateur devra néanmoins :

- réunir un jeu de données représentatif de l'utilisation du programme à optimiser ;
- et spécifier les compromis autorisés.

Le code C doit respecter la norme ANSI/C89 [KR88]. De plus, nous avons restreint le support de nos outils aux fonctionnalités suivantes : les boucles doivent être de type **for** et les appels de fonctions ne sont pas gérées. En effet, nous nous sommes focalisés sur les notions principales de fond. Ces notions permettent toutefois l'intégration rapide de ce qui n'est pas encore supporté par nos outils et pourront être ajoutées lors de travaux futurs (comme évoqué au chapitre des conclusions et perspectives). Nous rappelons de plus que

les données ne doivent pas dépasser un certain conditionnement (voir chapitre 6). Enfin, le code doit être enrichi de certaines directives de pré-compilation pour nos traitements. Ces directives permettent de spécifier les données importantes, afin de signaler les résultats dont on souhaite étudier la précision. D'autres directives permettent de donner des indications sur les boucles à transformer lors de l'application des stratégies `rs` et `ri`.

Spécification des compromis entre performances et précision Définissons les fonctions r_{cycles} et $r_{\text{bits exacts}}$, qui appliquées à une stratégie $s \in E$ donnent les ratios r_{cycles} et $r_{\text{bits exacts}}$ (définis par la relation (6.4)) d'un programme p ayant subit une transformation par la stratégie s. Définissons de plus la fonction $r_{\text{bits faux}}$ qui donne le ratio de bits faux définit par $r_{\text{bits faux}}(s) = 53 - r_{\text{bits exacts}}(s)$. La spécification des compromis entre performances et précision se fait selon les deux aspects suivants.

Précision d'abord. L'aspect « précision d'abord » signifie que l'utilisateur souhaite avant tout améliorer la précision de son programme. Dans ce cas, l'utilisateur n'a pas de contrainte particulière sur les performances. Il souhaite cependant que son programme transformé soit plus rapide que le même programme en *double-double*. Ce choix entraînera, soit la compensation complète du programme de l'utilisateur, soit la compensation partielle si il existe un programme présentant la même précision pour de meilleures performances. Il devra alors présenter le meilleur ratio performances-précision définit par la relation suivante :

$$R^0_{\text{p–p}} = \min_{s \in \{\max_{s \in E} r_{\text{bit exacts}}(s)\}} r_{\text{cycles}}(s) \tag{7.1}$$

Performances d'abord. L'aspect « performances d'abord » signifie que l'utilisateur souhaite garder les meilleures performances possibles pour son programme tout en gagnant le maximum de précision. C'est-à-dire que l'utilisateur souhaite un programme plus performant encore que le programme entièrement compensé et gagner en précision si possible. Il devra alors présenter le meilleur ratio performances-précision définit par les relations suivantes :

$$R^1_{\text{p–p}} = \min_{s \in E} \frac{\alpha \cdot r_{\text{cycles}}(s) + \beta \cdot r_{\text{bits faux}}(s)}{\alpha + \beta} \tag{7.2}$$

$$R^2_{\text{p–p}} = \max_{s \in E} | \, r_{\text{bits exacts}}(s) - r_{\text{cycles}}(s) \, | \tag{7.3}$$

où α, $\beta \in \mathbb{N}^*$ sont des constantes. On choisit par défaut $\alpha = \beta$, soit une importance équivalente entre précision et performances.

7.2.2 Espace de recherche

L'espace de recherche est défini de manière statique. C'est un ensemble E de 62 stratégies différentes qui a été créé dans le but de couvrir au mieux les stratégies possibles. Cet ensemble contient 18 stratégies de transformation par répartition simple, 40 stratégies de transformation par répartition intermittente et 4 stratégies de sélection par précision. Elles sont paramétrées par défaut pour couvrir des problèmes de petites tailles (c'est-à-dire une centaine d'opérations flottantes) mais peuvent être paramétrées plus finement selon le nombre v d'opérations élémentaires. La figure 7.1 présente par chacune des stratégies `rs`, `ri` et `sp` l'ensemble des paramétrages possibles à travers des arbres où chaque chemin représente un choix de paramètre.

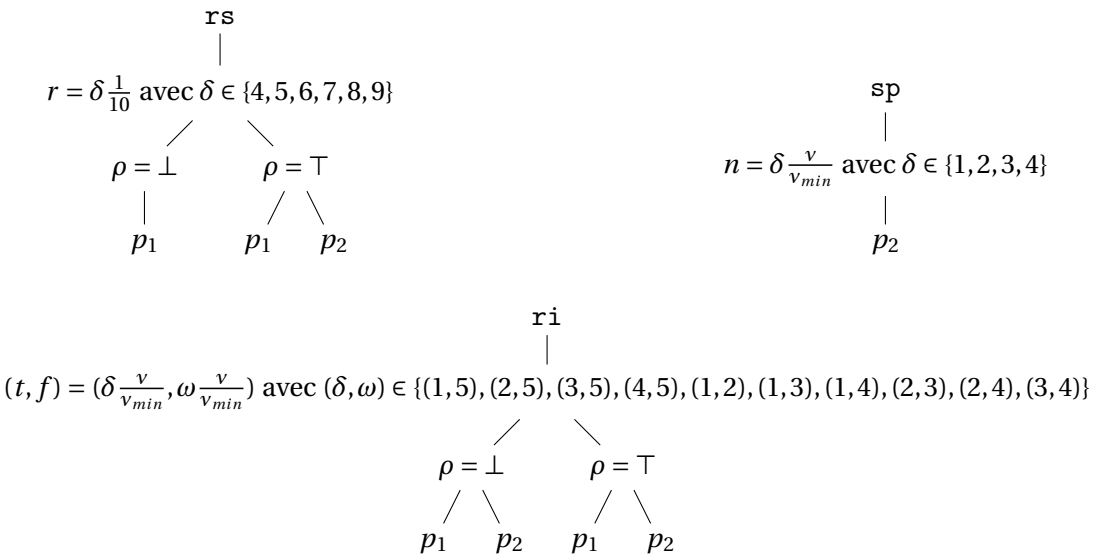


FIGURE 7.1 – Paramétrage de l'ensemble E des stratégies de la synthèse de code en fonction du nombre v d'opérations élémentaires (par défaut, v prend la valeur minimale $v_{min} = 10$).

Par exemple, les stratégies par répartition simple `rs` prennent deux paramètres, r le ratio d'itérations de la boucle à transformer et ρ la position de ces itérations. Un paramètre supplémentaire est p_i (avec $i \in \{1,2\}$) représentant le type de propagation à appliquer (voir chapitre 6.4). L'arbre correspondant de la figure 7.1 montre alors l'ensemble des valeurs que peuvent prendre ces paramètres. Ainsi, les stratégies `rs` compensent respectivement 40, 50, 60, 70, 80 et 90% du corps de la boucle. Elles sont indépendantes de la taille v des problèmes traités. Les stratégies `ri` compensent δ itérations de boucle toutes les ω itérations (voir figure 7.1 pour les différentes valeurs de δ et ω). Ces stratégies sont paramétrables de façon à traiter avec plus de robustesse les problèmes de grande taille. Par exemple, pour une somme d'un million de termes, compenser une itération sur cinq, sur le

million d'itérations est très peu efficace. C'est pourquoi le paramètre v permet d'appliquer un coefficient multiplicateur permettant de compenser un plus grand nombre d'itérations à la fois. En gardant le même exemple que précédemment, avec $v = 1\ 000\ 000$, 100 000 itérations de la boucle seraient alors compensées toutes les 500 000 itérations. Le même principe est appliqué pour les stratégies sp, permettant de sélectionner un nombre d'opérations à transformer qui s'adapte avec plus de cohérence à la taille du problème traité. Enfin, toutes les combinaisons possibles de positions des opérations à transformer et de propagations d'erreurs sont aussi générées.

7.2.3 Technique de recherche

Finalement, une technique de recherche (employant divers outils comme GCC et PAPI), permet de caractériser les bons choix à faire dans le but de synthétiser un programme répondant aux critères souhaités. Elle est présentée à l'algorithme 7.1. Les mesures de performances sont effectuées grâce à l'outil PAPI [MBDH99]. De plus, tous les programmes transformés peuvent être obtenus à l'aide de différents compilateurs et options (voir annexe A pour les configurations utilisées pour nos travaux). Le fonctionnement de cette méthode implique de pouvoir mesurer ou caractériser de façon automatique les performances et la précision (voir chapitre 6).

Par soucis de clarté, la fonction ÉVALUERLAPRÉCISION de l'algorithme 7.1 est simplifiée. Néanmoins, il est possible d'évaluer la précision pour plusieurs résultats au sein d'un même programme. La politique actuelle dans un tel cas est de considérer la moyenne des bits exacts des résultats mesurés.

7.3 Outils de transformation et de synthèse

Cette section présente les deux outils que nous avons développés pour la synthèse et la transformation d'un code source, en un code cible répondant à certaines spécifications. **CoHD** effectue des transformations source à source suivant différents paramètres générés par **SyHD**. SyHD, qui est finalement l'outil qui effectue la synthèse de code dans le but de fournir un programme transformé répondant aux spécifications désirées. Ces outils sont développé en OCAML.

La figure 7.2 décrit les phases et interactions que possèdent ces différents outils. Elles sont détaillées dans les sections 7.3.1 et 7.3.2.

7.3.1 CoHD : transformation de code source à source

CoHD est un outil de transformation de code C source à source. Cet outil implémente les contributions présentées aux chapitres 5 et 6. Il est la base de la synthèse de code mais peut

Entrée(s) : L'ensemble S des stratégies, l'ensemble D des données et le nombre moyen $n_{\text{bits exacts}}$ de bits exacts des résultats obtenus avec le programme en *double-double*.
Sortie(s) : Un programme p_ω automatiquement transformé répondant le mieux aux critères de performances et de précision désirés.

fonction ÉVALUERLESPERFORMANCES(p, d, n_{ref}) ▷ *Évaluer les performances du programme p selon la valeur de référence n_{ref} et des données d.*
  $n \leftarrow$ le nombre de cycles du programme p mesuré par PAPI.
  retourner $r_{\text{cycles}} \leftarrow \frac{n}{n_{\text{ref}}}$ ▷ *voir relation (6.4)*
fin fonction

fonction ÉVALUERLAPRÉCISION(p, d, n_{ref}) ▷ *Évaluer la précision du programme p selon la valeur de référence n_{ref} et des données d.*
  $o \leftarrow$ la valeur numérique du résultat mesuré.
  $n \leftarrow N_{\text{Overton}}(o)$, le nombre de bits exacts du résultat o. ▷ *voir relation (1.2)*
  retourner $r_{\text{bits exacts}} \leftarrow \frac{n}{n_{\text{ref}}}$ ▷ *voir relation (6.4)*
fin fonction

▷ *Initialisation des valeurs de référence.*
$p_{\text{cycles}} \leftarrow$ le programme *double-double* de référence pour la mesure des performances.
$n_{\text{cycles}} \leftarrow$ le nombre de cycles du programme p_{cycles} mesuré par PAPI.
$n_{\text{bits exacts}} \leftarrow 53$ ▷ *Le nombre de bits exacts par défaut des résultats en double-double de référence. Cette valeur peut être paramétrée, mais les mesures doivent être effectuées à part (voir section 6.6.2).*
▷ *Calcul des ratios R_{p-p} performances–précision.*
$\Omega \leftarrow \emptyset$
pour tout $i \in S$ **faire**
  $p^i \leftarrow$ le programme transformé selon la stratégie i.
  $r^i_{\text{cycles}} =$ ÉVALUERLESPERFORMANCES(p^i, `head`(D), n_{cycles})
  pour tout $d \in D$ **faire**
    $r^{i_d}_{\text{bit exacts}} =$ ÉVALUERLAPRÉCISION(p^i, d, $n_{\text{bit exacts}}$)
  fin pour
  $r^i_{\text{bit exacts}} \leftarrow$ moyenne $\left(r^{i_d}_{\text{bit exacts}}\right)$
  $\Omega \leftarrow \Omega \cup \left(i, r^i_{\text{cycles}}, r^i_{\text{bit exacts}}\right)$
fin pour
$s \leftarrow R^j_{\text{p–p}}(\Omega)$ ▷ *Retourne la stratégie gagnante (voir relationss (7.1), (7.2) et (7.3)).*
retourner $p_\omega \leftarrow$ le programme transformé selon la stratégie s.

Algorithme 7.1 – Recherche de programmes optimisé par la synthèse de code.

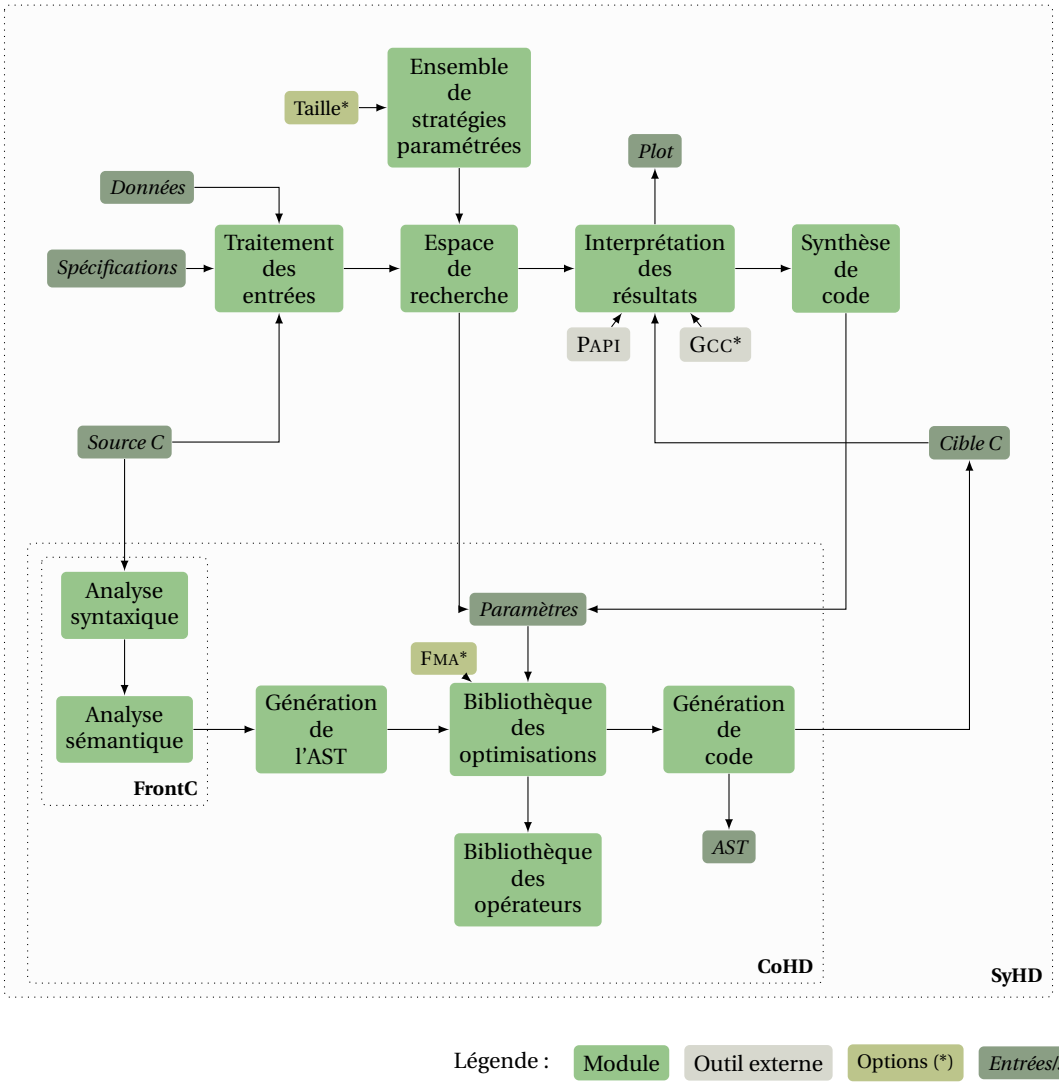


FIGURE 7.2 – Phases et interactions des outils de transformation *CoHD* et de synthèse de programme *SyHD* (modèle simplifié).

être utilisé seul, dans un but de profilage ou d'optimisation manuelle de code. La figure 7.2 représente cet outil à travers les six modules suivants.

• *Analyses syntaxique et sémantique.* Les modules *Analyse syntaxique* et *Analyse sémantique* permettent de transformer un programme C ANSI/C89 [KR88] en une représentation intermédiaire. Cette transformation est assurée par FRONTC [Cas00], un *parser/lexer* C en OCAML.

• *Génération de l'AST (arbre de syntaxe abstrait).* Ce module permet de transformer la représentation intermédiaire générée par FRONTC en un arbre de syntaxe abstrait enrichi de nombreuses informations. Par exemple, l'arbre est enrichi des informations données par les directives de pré-compilation destinées aux diverses optimisations qui devront être effectuées par nos outils.

• *Bibliothèque des optimisations.* Ce module contient toutes les passes d'optimisation disponibles au sein de l'outil de transformation de code. Cette bibliothèque comprend les passes suivantes : transformation en code trois adresses, compensation automatique complète, transformation en *double-double*, transformation suivant les stratégies `rs`, `ri` et `sp`. Ces passes sont sélectionnées suivant les paramètres passés en entrée. De plus, une option permet d'activer le support de l'opération FMA. Ces passes font appels à de nombreux opérateurs appliqués lors des différentes transformations. Ces opérateurs sont réunis au sein du module suivant.

• *Bibliothèque des opérateurs.* Ce module comprend l'ensemble des opérateurs de compensation (algorithmes 5.1 et 5.2), *double-double* (algorithmes 2.11 et 2.12) et de propagation des erreurs (algorithmes 6.1 et 6.2). Ces opérateurs sont appliqués suivant les différentes passes du module *bibliothèque des optimisations* au sein de l'arbre syntaxique.

• *Génération de code.* Enfin, ce dernier module permet de générer le code C représentatif de l'arbre de syntaxe abstrait modifié par l'ensemble des passes subies par la représentation intermédiaire du programme. De plus, il est possible via ce module de sortir l'arbre de syntaxe sous forme de graphe DOT (GRAPHVIZ).

7.3.2 SyHD : synthèse de code

SyHD est un synthétiseur de code C. Cet outil implémente les résultats de ce chapitre. Cet outil trouve, à partir des critères de performances et de précision, un programme optimisé y répondant le mieux. Il est destiné à être utilisé de façon automatique, mais peut être utilisé manuellement toujours dans des buts de profilage ou d'optimisation manuelle. La figure 7.2 représente les cinq modules suivants de cet outil.

• *Parser.* Ce module gère la totalité des entrées de nos outils. Le code source à transformer, les spécifications des compromis performances–précision désirés et l'ensemble des données.

• *Espace de recherche E.* L'espace de recherche permet la gestion de l'*ensemble des stratégies paramétrées*. C'est ce module qui, pour chaque stratégie fait appel à *CoHD* afin de générer le code transformé y correspondant.

• *Ensemble de stratégies paramétrées.* Ce module gère la création et la gestion des programmes transformés selon l'ensemble des stratégies. Une option permet de paramétrer plus finement la création de l'ensemble des stratégies comme évoqué à la section 7.2.2.

• *Interprétation des résultats.* Ce module reprend le fonctionnement présenté à l'algorithme 7.1 de la section 7.2.3. Ce module mesure les performances et la précision de chaque programme transformé. Il utilise des outils externes tels que GCC (par défaut) et PAPI [MBDH99]. Il est de plus possible de générer des graphes à l'aide de GNUPLOT afin de visualiser les résultats en terme de performances et de précision pour chaque programme étudié (de nombreux exemples sont donnés au chapitre 8).

• *Synthèse de code.* Finalement, le module *synthèse de code* permet de faire le choix parmi l'ensemble des programmes étudiés répondant le mieux aux spécifications.

7.4 Conclusion

Nous avons présenté notre approche pour la recherche de programmes transformés ainsi que les outils que nous avons développés afin d'en fournir un prototype. Ce processus automatique offre une solution d'optimisation de code, destinée à un public de développeurs non spécialistes des particularités de l'arithmétique flottante et de l'algorithmique numérique. Des résultats obtenus avec notre prototype sont présentés au chapitre 8.

CHAPITRE 8

RÉSULTATS EXPÉRIMENTAUX

8.1 Introduction

CE CHAPITRE est consacré aux résultats expérimentaux obtenus avec à nos outils de synthèse, implémentant l'ensemble des contributions des chapitres 5, 6 et 7.

La section 8.2 développe les exemples étudiés aux chapitres 5 et 6 : la sommation et les évaluations polynomiales. Les résultats qui y sont présentés démontrent l'intérêt des compromis performances–précision (définis au chapitre 7) dans le cadre de la synthèse de code. Ces cas sont dorénavant étudiés à travers le processus automatique de la synthèse. La section 8.3 est dédiée à un problème différent. Il s'agit du raffinement itératif, utile à la résolution de système linéaire.

8.1.1 Définitions et rappels des notations

Nous rappelons au tableau 8.1 les différentes notations développées dans les chapitres précédents. De plus, nous introduisons la définition 8.1.1 qui pose les conditions du succès de la synthèse. La définition 8.1.2 présente ce que nous considérons comme une stratégie optimale : les stratégies qui seront retenues par la synthèse de code.

Définition 8.1.1 (Succès de la synthèse)**.** *Un programme transformé répond avec succès aux compromis performances–précision si et seulement si pour chaque donnée d'un jeu de données, le programme synthétisé en améliore la précision par rapport au programme en binary64. Il doit de plus présenter un temps de calcul inférieur au programme en double-*

Notation	Description
$\texttt{rs}(\rho, r)$	Stratégie de transformation de boucle par répartition simple. Une proportion r d'itérations de la boucle est compensée d'un bloc. Le paramètre $\rho \in \{\bot, \top\}$ défini si ce sont les premières ou les dernières itérations qui sont transformées (voir section 6.4).
$\texttt{ri}(\rho, t, f)$	Stratégie de transformation de boucle par répartition intermittente. Un nombre t d'itérations est compensé toutes les f itérations. Le paramètre ρ à la même signification que pour la répartition `rs`.
$\texttt{sp}(n)$	Stratégie de transformation par sélection de la précision. Les n opérations causant les plus grandes erreurs sont compensées (voir section 6.5).
$p_1,\ p_2$	Définissent la politique de propagation des erreurs au sein des calculs non compensés par les stratégies `rs`, `ri` et `sp` (voir section 6.3.1).
$r_{\text{bits exacts}}$	Ratio du nombre de bits exacts du résultat, entre le programme transformé et le programme *double-double*, défini par la relation (6.4).
r_{cycles}	Description identique, mais cette fois c'est le nombre de cycles qui est concerné.
$R^i_{\text{p–p}}$	Ratios performances–précision tels que définis pour $i \in \{0,1,2\}$ par les relations 7.1, 7.2 et 7.3 respectives. Ces ratios déterminent l'orientation des compromis performances–précision : $R^0_{\text{p–p}}$ pour l'aspect « précision d'abord » et les ratios $R^1_{\text{p–p}}$ et $R^2_{\text{p–p}}$ pour l'aspect « performances d'abord ».

TABLE 8.1 – Rappel des notations.

double pour la sélection par le ratio performances-précision $R^0_{p–p}$ ou inférieur au programme en auto-comp pour la sélection par les ratios performances-précision $R^1_{p–p}$ et $R^2_{p–p}$.

Définition 8.1.2 (Stratégie optimale (ou stratégie « gagnante »)**.** *Une stratégie optimale, ou « gagnante », est une stratégie qui remplie des conditions suivantes :*

(i) le programme généré répond avec succès au compromis performances–précision tel que présenté à la définition 8.1.1 ;

(ii) le programme généré présente le meilleur ratio performances–précision $R^i_{p–p}$ pour $i \in \{0,1,2\}$ (voir tableau 8.1).

Si plusieurs stratégies présentent des caractéristiques identiques, le synthétiseur détermine une stratégie optimale arbitrairement parmi celles-ci.

8.2 Synthèse de code avec compromis

Nous considérons les cas déjà étudiés dans les chapitres 5 et 6 : des algorithmes de sommation et d'évaluations polynomiales. Nous avions considéré leurs compensation complète ou partielle pour des stratégies de transformations choisies arbitrairement. Nous proposons et étudions maintenant un ensemble fixe de stratégies permettant, via la synthèse de code, l'émergence d'un programme transformé répondant aux attentes d'un utilisateur (les compromis performances–précision détaillés au chapitre 7). Ces attentes sont définies :

- en terme de précision, c'est-à-dire en nombre de bits exacts des résultats calculés ;
- et en terme de temps de calcul, soit en nombre de cycles mesurés avec l'outil PAPI.

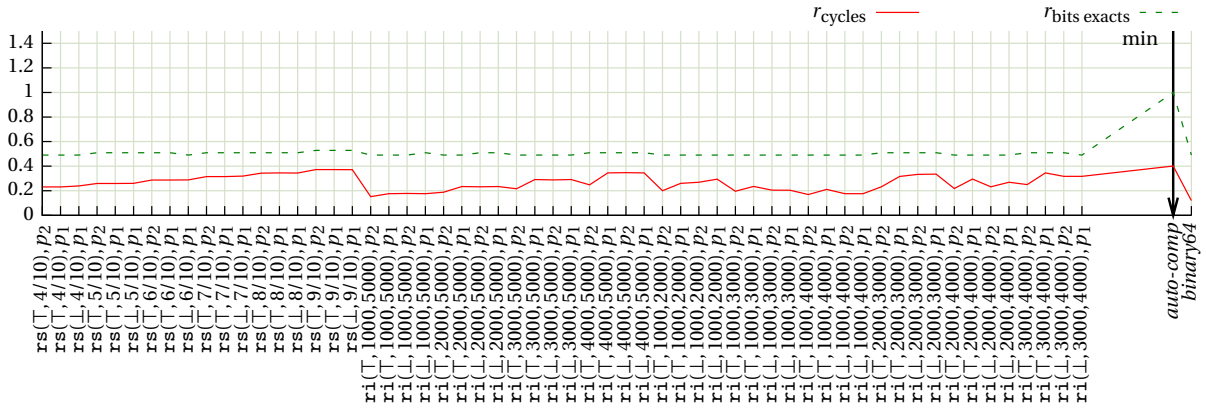
Les résultats de cette section présentent les performances de nos outils dans la recherche d'un tel code. La section 8.2.1 reprend le cas de l'algorithme de sommation 2.7 (SUM [ROO05]), précédemment étudié aux sections 5.3.1, 5.3.3 et 6.6.2. La section 8.2.2 reprend les cas des algorithmes d'évaluation polynomiale. Ces expériences présentent les résultats de la synthèse de code pour les algorithmes 5.3 (HORNER [GLL09]) et 5.4 (CLENSHAWI [JBL+11]) précédemment étudiés aux sections 5.3.2, 5.3.3 et 6.6.2. La section 8.2.3 présente enfin un récapitulatif des résultats obtenus sur l'ensemble des cas étudiés, pour différentes configurations d'environnements et de données (valeurs à sommer pour la somme, et points d'évaluations pour les polynômes). Nous détaillons dans l'annexe A les environnements utilisés : compilateurs, options, architectures, système…

8.2.1 Cas de la somme

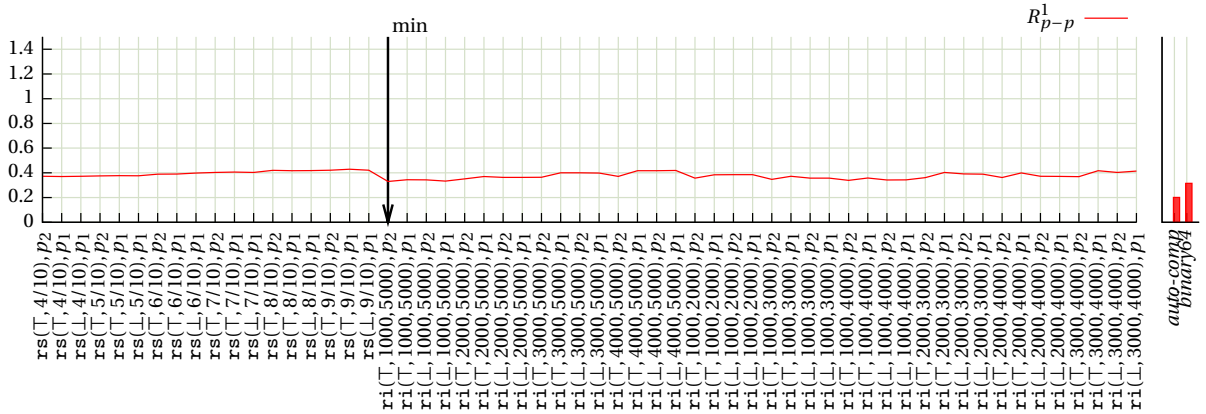
Les résultats concernent l'algorithme de sommation SUM 2.7 implémenté par le programme 6.8. Cet algorithme effectue la somme séquentielle de n termes. Les données utilisées (i.e. les termes à sommer) sont présentées au tableau 8.2 avec les ratios de cycles et de bits exacts entre les programmes en *binary64* et en *double-double*.

Données	Description	r_{cycles}		$r_{\text{bits exacts}}$
		A.1	A.2	
d_1	32 jeux de 10^4 valeurs de conditionnement $\simeq 10^8$	0,12	0,11	0,49
d_2	32 jeux de 10^5 valeurs de conditionnement $\simeq 10^8$		0,16	0,47
d_3	32 jeux de 10^6 valeurs de conditionnement $\simeq 10^8$	0,14		0,43

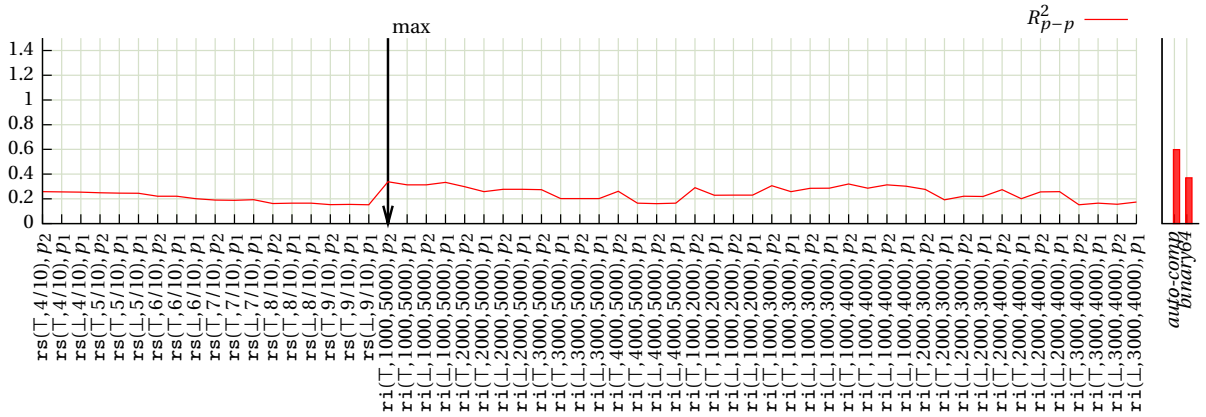
TABLE 8.2 – Présentation des données avec leurs ratios r_{cycles} et $r_{\text{bits exacts}}$ pour le programme 6.8 SUM en *binary64* dans les environnements A.1 et A.2.



(a) Sélection par le ratio $R^0_{\text{p–p}}$.



(b) Sélection par le ratio $R^1_{\text{p–p}}$.



(c) Sélection par le ratio $R^2_{\text{p–p}}$.

FIGURE 8.1 – Détails des ratios performances–précision de l'algorithme SUM et les données d_1 dans l'environnement A.1. Les stratégies gagnantes retenues sont indiquées par une flèche.

Ces ratios sont définis par la relation (6.4). Par exemple, le jeu de données d_1 comprend 32 jeux de 10^4 nombres flottants de conditionnement avoisinant 10^8. Sur l'ensemble de ces données, le programme en *binary64* présente en moyenne 12% du nombre de cycles et 49% de la précision du programme en *double-double* (dans l'environnement A.1).

La figure 8.1 présente les résultats, en terme de ratios de bits exacts et de cycles, pour les stratégies étudiées par la synthèse avec les données d_1 dans l'environnement A.1. Les stratégies gagnantes sont désignées par les sélections des différents ratios performances-précision (voir chapitre 7). Ces sélections permettent de proposer un programme transformé répondant aux critères de précision et de performances spécifiés par un utilisateur. La figure 8.1(a) montre la sélection suivant le ratio orienté précision $R^0_{\text{p–p}}$. La stratégie retenue par cette sélection correspond à la compensation automatique. Les figures 8.1(b) et 8.1(c) montrent respectivement les sélections suivant les ratios orientés performances $R^1_{\text{p–p}}$ et $R^2_{\text{p–p}}$. Les stratégies retenues par ces sélections correspondent, dans ce cas, au même programme transformé suivant la stratégie de compensation partielle $\texttt{ri}(\top, 1\,000, 5\,000)$ et la propagation p_2. L'ensemble des résultats est présenté au tableau 8.3.

Le tableau 8.3 présente les résultats obtenus par la synthèse de code (des résultats complémentaires sont aussi donnés en annexe au tableau D.1). Ces résultats présentent, pour chaque sélection et jeux de données, les stratégies gagnantes et leurs ratios r_{cycles} et $r_{\text{bits exacts}}$. Les tableaux spécifient de plus si la synthèse de code s'est opérée avec succès ou non. Il y a succès si le programme répond à la définition 8.1.1.

La figure 8.2 présente les résultats fournis par le programme généré par la synthèse selon la sélection $R^0_{\text{p–p}}$. Ce ratio performances-précision privilégie la précision du résultat final de l'algorithme. Nous observons à la figure 8.1(a) les résultats de tous les programmes générés par la synthèse de code. Le programme compensé automatiquement (*auto-comp*) est le seul qui permet le maximum de précision. C'est alors celui-ci qui est retenu par la synthèse. La figure 8.2 montre que pour tous les jeux de données, le programme *auto-comp* parvient à obtenir le maximum de précision. Le tableau 8.3 présente de plus que cette précision est assurée pour 40% du coût du programme en *double-double*. Dans ces circonstances, la synthèse est un succès, car elle répond aux contraintes de la définition 8.1.1.

La figure 8.3 présente les résultats fournis par le programme généré par la synthèse selon la sélection $R^1_{\text{p–p}}$ ou $R^2_{\text{p–p}}$. Ces ratios performances-précision privilégient un faible temps de calcul pour le programme. Nous observons aux figures 8.1(b) et 8.1(c) les résultats de tous les programmes générés par la synthèse de code. Comme présenté au tableau 8.3, le programme final fournit par la synthèse de code nécessite 15% du temps de calcul du programme en *double-double*. Il en présente de plus 49% de sa précision. Dans les deux cas la stratégie de transformation retenue ne permet pas d'améliorer la précision pour tous les jeux de données. La figure 8.3 montre que le troisième jeu de données voit

Sélection	Données	Stratégie gagnante	r_{cycles}	$r_{\text{bits exacts}}$	Succès
		Environnement A.1			
$R^0_{\text{P-P}}$	d_1	*auto-comp*	0,4	1	✓
	d_2				
	d_3		0,41		
$R^1_{\text{P-P}}$	d_1	`ri`$(\top, 1\,000, 5\,000), p_2$	0,15	0,49	✗
	d_2	`ri`$(\top, 10\,000, 50\,000), p_2$		0,47	
	d_3	`ri`$(\top, 100\,000, 400\,000), p_2$	0,19	0,45	
$R^2_{\text{P-P}}$	d_1	`ri`$(\top, 1\,000, 5\,000), p_2$	0,15	0,49	✗
	d_2	`ri`$(\top, 10\,000, 50\,000), p_2$		0,47	
	d_3	`ri`$(\top, 100\,000, 400\,000), p_2$	0,19	0,45	
		Environnement A.2			
$R^0_{\text{P-P}}$	d_1	*auto-comp*	0,4	1	✓
	d_2		0,43		
	d_3				
$R^1_{\text{P-P}}$	d_1	`ri`$(\bot, 1\,000, 5\,000), p_1$	0,18	0,5	✗
	d_2	`ri`$(\top, 10\,000, 50\,000), p_2$	0,2	0,47	
	d_3	`ri`$(\top, 100\,000, 500\,000), p_2$		0,43	
$R^2_{\text{P-P}}$	d_1	`ri`$(\bot, 1\,000, 5\,000), p_1$	0,16	0,5	✗
	d_2	`ri`$(\top, 10\,000, 50\,000), p_2$	0,2	0,47	
	d_3	`ri`$(\top, 100\,000, 500\,000), p_2$		0,43	

TABLE 8.3 – Résultats de la synthèse de code de l'algorithme SUM. Présentation des stratégies gagnantes, des ratios r_{cycles} et $r_{\text{bits exacts}}$ pour chaque jeux de données du tableau 8.2 et sélections performances–précision dans les environnements A.1 et A.2.

sa précision passer de 25 bits exacts à 33, alors que la plupart des autres jeux voient leur précision diminuer. Dans ces circonstances, la synthèse est un échec, car elle ne répond pas aux contraintes de la définition 8.1.1.

En effet, en observant la figure 8.1(c), nous remarquons qu'aucune des stratégies étudiées ne parvient réellement à améliorer la précision pour les données d_1. Le tableau 8.3 généralise cette observation pour les jeux de données d_2 et d_3.

La raison la plus satisfaisante pour expliquer cet échec, provient d'un paramétrage inadéquat du synthétiseur. En effet, les stratégies `ri` pourraient être paramétrées de façon à compenser encore plus de calcul. Par exemple, la stratégie `ri`$(\top, 6000, 7000)$ avec la propagation p_2 permet d'atteindre en moyenne 54% de la précision du programme en *double-double* pour 37% de son temps de calcul. Cette stratégie préserve le temps de calcul pour

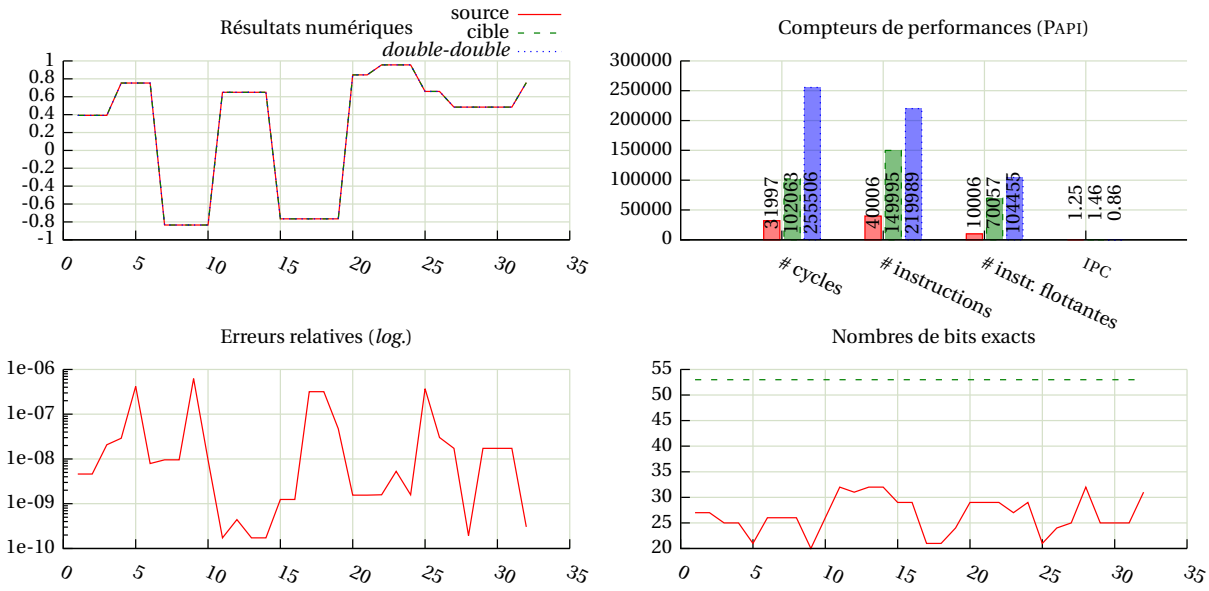


FIGURE 8.2 – Détails des performances et de la précision pour l'algorithme SUM avec la stratégie gagnante sélectionnée par le ratio $R^0_{\text{p–p}}$ pour les données d_1 dans l'environnement A.1 (les abscisses représentent les 32 jeux de données).

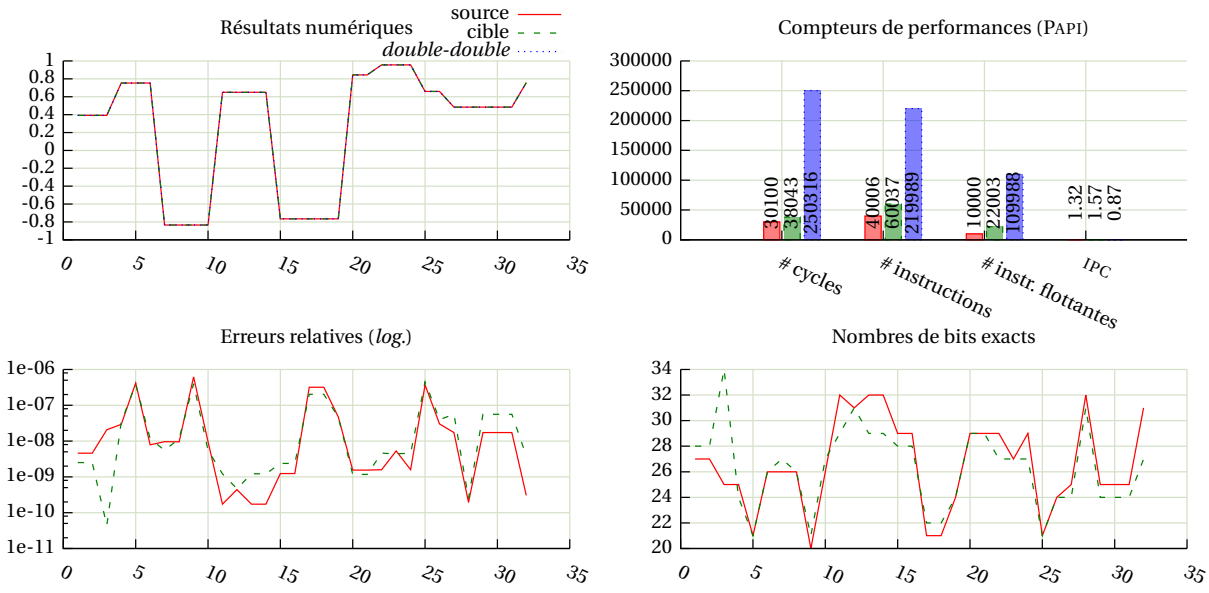


FIGURE 8.3 – Détails des performances et de la précision pour l'algorithme SUM avec la stratégie gagnante sélectionnée par le ratio $R^1_{\text{p–p}}$ ou $R^2_{\text{p–p}}$ pour les données d_1 dans l'environnement A.1 (les abscisses représentent les 32 jeux de données).

tous les jeux de données des données d_1 et en améliore la précision pour 28 d'entre eux. Selon la définition 8.1.1, cette stratégie de transformation ne permet pas encore de remplir les critères de satisfaisabilité pour la synthèse. Néanmoins, nous montrons par ce biais qu'il est possible de rechercher manuellement une stratégie présentant de meilleurs résultats lorsque la synthèse par défaut échoue.

Temps pour la synthèse Le processus complet de la synthèse de code prend environ 20 secondes, 1 minute et 8 minutes pour traiter respectivement les données d_1, d_2 et d_3 dans l'environnement A.1. Dans l'environnement A.2, il faut environ 30 secondes, 2 minutes et 16 minutes pour traiter les données d_1, d_2 et d_3.

8.2.2 Évaluations polynomiales

Cette section présente les résultats pour les algorithmes d'évaluations polynomiales HORNER et CLENSHAWI.

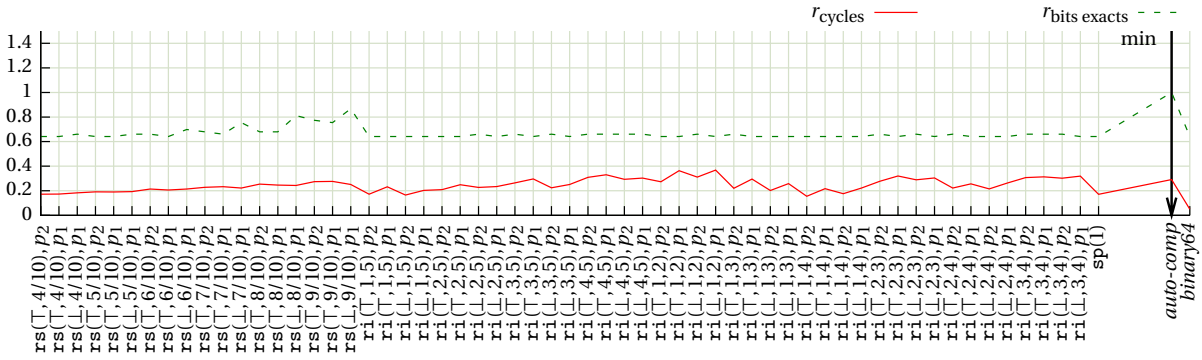
Synthèse de l'évaluation polynomiale avec HORNER

Les résultats suivants concernent l'algorithme 5.3 d'évaluation polynomiale avec HORNER. Les données utilisées sont présentées au tableau 8.4 avec les ratios de cycles et de bits exacts entre les programmes en *binary64* et *double-double*. Un jeu de données représente l'ensemble des points x d'évaluations du polynôme $p_H(x) = (x-0,75)^5(x-1,0)^{11}$. Ces ratios sont définis par l'équation (6.4). Par exemple le jeu de données d_4 comprend 256 valeurs tirées uniformément dans l'intervalle $\{0,35 : 0,45\}$. Sur l'ensemble de ces données, le programme en *binary64* présente en moyenne 5% du nombre de cycles et 64% de la précision du programme en *double-double* (dans l'environnement A.1).

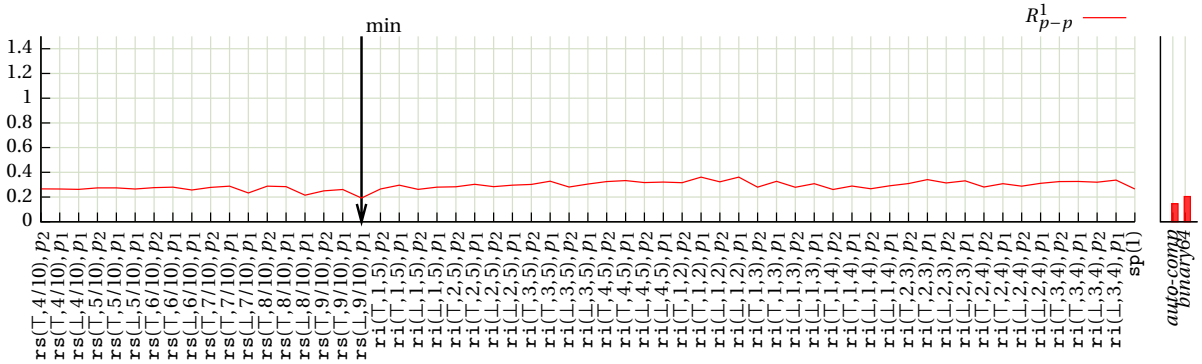
Données	Description	r_{cycles} A.1	r_{cycles} A.2	$r_{\text{bits exacts}}$
d_4	256 valeurs tirées uniformément dans $\{0,35 : 0,45\}$			0,64
d_5	256 valeurs tirées uniformément dans $\{0,6 : 0,7\}$	0,05	0,1	0,22
d_6	256 valeurs tirées uniformément dans $\{1,8 : 1,9\}$			0,58

TABLE 8.4 – Présentation des données avec leurs ratios r_{cycles} et $r_{\text{bits exacts}}$ du programme 5.3 HORNER en *binary64* dans les environnements A.1 et A.2.

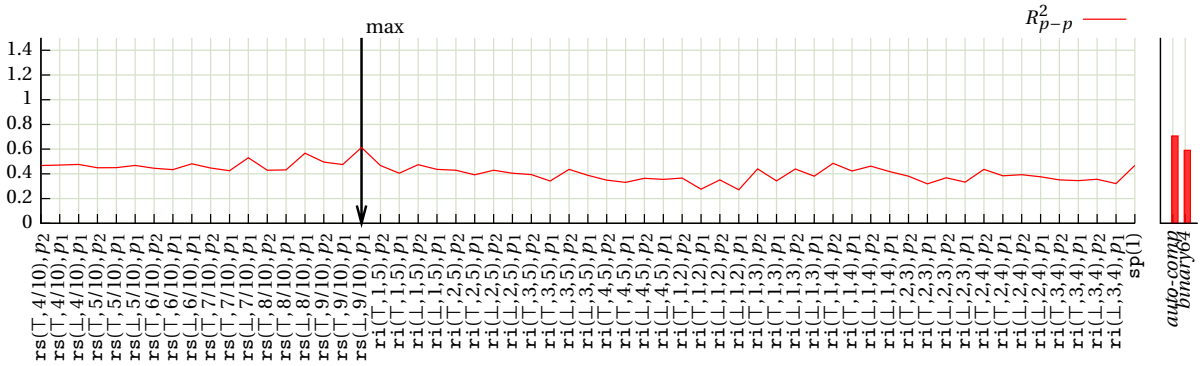
La figure 8.4 présente les résultats de la synthèse de code avec les données d_4 dans l'environnement A.1. Elle présente les stratégies gagnantes désignées par les sélections des différents ratios performances-précision (voir chapitre 7). Ces sélections permettent de proposer un programme transformé répondant aux critères de précision et de performances spécifiés par un utilisateur. La figure 8.4(a) montre la sélection suivant le ratio



(a) Sélection par le ratio $R^0_{\text{p–p}}$.



(b) Sélection par le ratio $R^1_{\text{p–p}}$.



(c) Sélection par le ratio $R^2_{\text{p–p}}$.

FIGURE 8.4 – Détails des ratios performances–précision de l'algorithme HORNER et les données d_4 dans l'environnement A.1. Les stratégies gagnantes retenues sont indiquées par une flèche.

orienté précision $R^0_{\text{p–p}}$. La stratégie retenue par cette sélection correspond à la compensation automatique. Les figures 8.4(b) et 8.4(c) montrent respectivement les sélections suivant les ratios orientés performances $R^1_{\text{p–p}}$ et $R^2_{\text{p–p}}$. Les stratégies retenues par ces sélections correspondent, dans ce cas, au même programme transformé suivant la stratégie de compensation partielle $\texttt{rs}(\bot, 9/10)$ et la propagation p_1. L'ensemble des résultats est présenté entièrement au tableau 8.5.

Le tableau 8.5 présente les résultats obtenus par la synthèse de code (des résultats complémentaires sont aussi donnés en annexe au tableau D.2). Pour chaque sélection et jeux de données, les tableaux en donnent les stratégies gagnantes et leurs ratios r_{cycles} et $r_{\text{bits exacts}}$. Les tableaux indiquent de plus si la synthèse de code s'est opérée avec succès comme spécifiée par la définition 8.1.1.

La figure 8.5 présente les résultats fournis par le programme généré par la synthèse selon la sélection $R^0_{\text{p–p}}$. Ce ratio performances-précision privilégie la précision du résultat final de l'algorithme. Nous observons à la figure 8.4(a) les résultats de tous les programmes générés par la synthèse de code. Le programme compensé automatiquement (*auto-comp*) est naturellement le seul qui permet le maximum de précision. C'est alors celui-ci qui est retenu par la synthèse. La figure 8.5 montre que pour toutes les données de l'intervalle d_4, le programme *auto-comp* parvient à récupérer le maximum de précision. Le tableau 8.5 présente de plus que cette précision est assurée pour 29% du coût du programme en *double-double*. Dans ces circonstances, la synthèse est un succès, car elle répond aux contraintes de la définition 8.1.1.

La figure 8.6 présente les résultats fournis par le programme généré par la synthèse selon la sélection $R^1_{\text{p–p}}$ ou $R^2_{\text{p–p}}$. Ces ratios performances-précision privilégient un faible temps de calcul pour le programme. Nous observons aux figures 8.4(b) et 8.4(c) les résultats de tous les programmes générés par la synthèse de code. Comme présenté au tableau 8.5, le programme final fournis par la synthèse de code nécessite 25% du temps de calcul du programme en *double-double* (alors que le programme *auto-comp* en prend 29%). Il présente de plus 86% de sa précision. Dans les deux cas la stratégie de transformation retenue permet d'améliorer la précision pour toutes les données de l'intervalle d_4. La figure 8.6 montre que la précision des résultats finaux passent d'entre 35 et 40 bits exacts à 45 et 53. Dans ces circonstances, la synthèse est un succès car elle répond aux contraintes de la définition 8.1.1. De plus, nous pourrions aller plus loin dans la recherche d'un code possédant un meilleur nombre de cycle. En effet, nous pouvons remarquer que la stratégie par répartition simple $\texttt{rs}(\bot, 9/10)$ avec la propagation p_1 présente des performances proches du programme entièrement compensé (*auto-comp*). En mettant par exemple de côté cette stratégie, la synthèse donnerait alors gagnant avec succès la stratégie $\texttt{rs}(\bot, 8/10), p_1$ présentant 24% du nombre de cycles du code en *double-double* ainsi que 81% de sa précision. De même, pour aller encore plus loin, la synthèse de code permet de trouver la stratégie

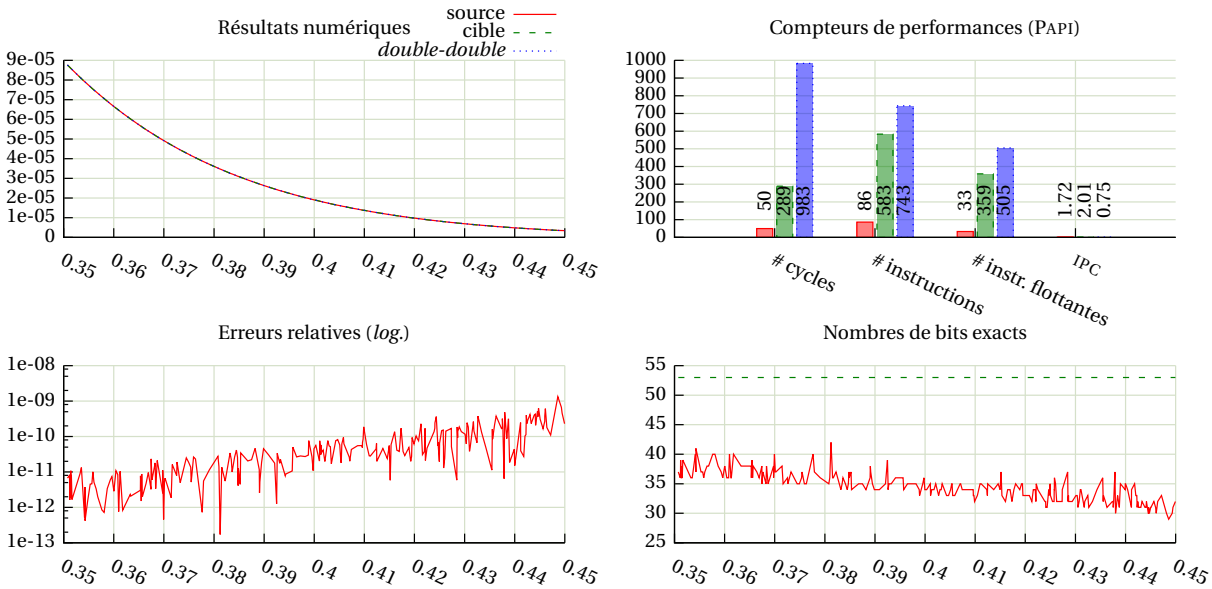


FIGURE 8.5 – Détails des performances et de la précision pour l'algorithme HORNER avec la stratégie gagnante sélectionnée par le ratio $R^0_{\text{p–p}}$ pour les données d_4 dans l'environnement A.1.

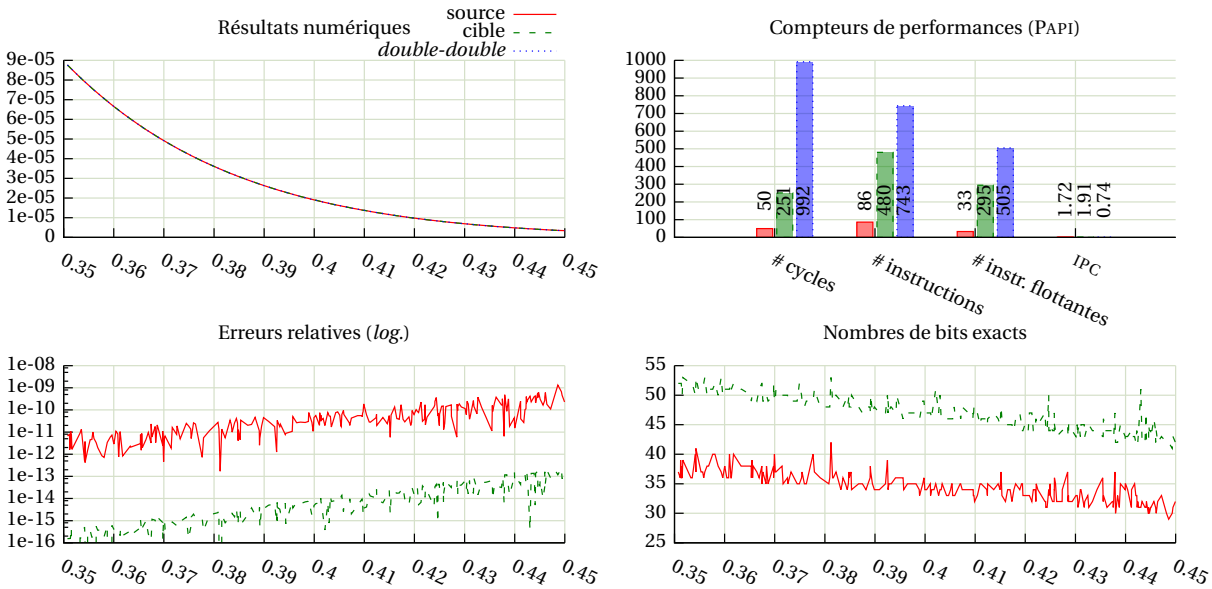


FIGURE 8.6 – Détails des performances et de la précision pour l'algorithme HORNER avec la stratégie gagnante sélectionnée par le ratio $R^1_{\text{p–p}}$ ou $R^2_{\text{p–p}}$ pour les données d_4 dans l'environnement A.1.

Ratio	Données	Stratégie gagnante	r_{cycles}	$r_{\text{bits exacts}}$	Succès
		Environnement A.1			
$R^0_{\text{P-P}}$	d_4 d_5 d_6	*auto-comp*	0,29	1	✓
$R^1_{\text{P-P}}$	d_4	RS($\perp$,9/10), p_1	0,25	0,86	
	d_5	RS($\top$,9/10), p_2	0,27	0,43	✓
	d_6	RS($\top$,9/10), p_1		0,98	
$R^2_{\text{P-P}}$	d_4	RS($\perp$,9/10), p_1	0,25	0,86	
	d_5	RS($\top$,9/10), p_2	0,27	0,43	✓
	d_6			0,98	
		Environnement A.2			
$R^0_{\text{P-P}}$	d_4		0,39		
	d_5	*auto-comp*	0,37	1	✓
	d_6		0,44		
$R^1_{\text{P-P}}$	d_4	RS($\perp$,9/10), p_1	0,32	0,86	
	d_5	RS($\top$,9/10), p_1	0,35	0,41	✓
	d_6			0,98	
$R^2_{\text{P-P}}$	d_4	RS($\perp$,9/10), p_1	0,33	0,86	✓
	d_5	RI($\top$,4,5), p_1	0,43	0,24	✗
	d_6	RS($\top$,9/10), p_1	0,35	0,98	✓

TABLE 8.5 – Résultats de la synthèse de code de l'algorithme HORNER. Présentation des stratégies gagnantes, des ratios r_{cycles} et $r_{\text{bits exacts}}$ pour chaque jeux de données du tableau 8.4 et sélections performances–précision dans les environnements A.1 et A.2.

`rs`($\perp$,7/10), p_1 présentant 22% du nombre de cycles du code en *double-double* ainsi que 75% de sa précision.

Temps pour la synthèse Le processus complet de la synthèse de code prend en moyenne une minute pour les données d_4, d_5 et d_6 dans l'environnement A.1. De plus, il en faut en moyenne 1 minute et 5 secondes dans l'environnement A.2.

Synthèse de l'évaluation polynomiale avec CLENSHAWI

Les résultats suivants concernent l'algorithme 5.4 d'évaluation polynomiale avec CLENSHAWI (implémenté dans le code 6.11).

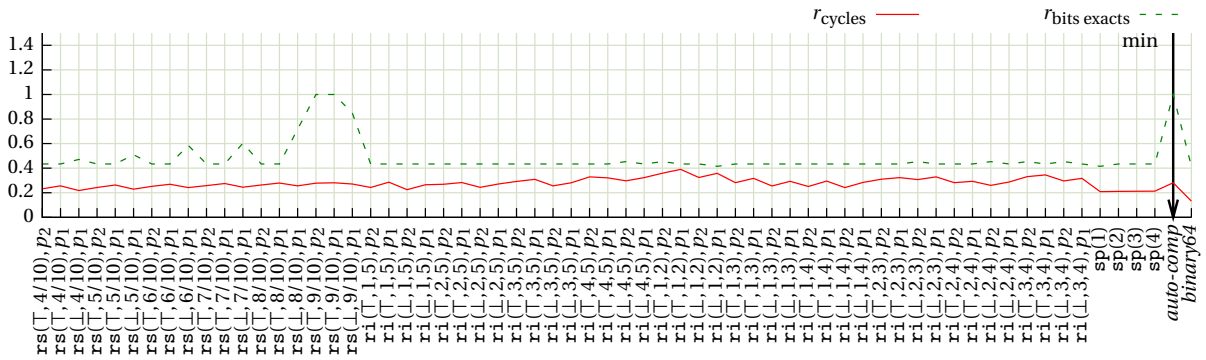
Les données utilisées sont présentées au tableau 8.6 avec les ratios de cycles et de bits exacts entre les programmes en *binary64* et *double-double*. Ces ratios sont définis par l'équation (6.4). Un jeu de données représente l'ensemble des points x d'évaluations du polynôme $p_C(x) = (x-0,75)^7(x-1,0)^{10}$. Par exemple, le jeu de données d_4 comprend 256 valeurs tirées uniformément dans l'intervalle $\{0,35 : 0,45\}$. Sur l'ensemble de ces données, le programme en *binary64* présente en moyenne 13% du nombre de cycles du programme en *double-double* (dans l'environnement A.1) et 41% de sa précision.

Données	Description	r_{cycles} A.1	r_{cycles} A.2	$r_{\text{bits exacts}}$
d_4	256 valeurs tirées uniformément dans $\{0,35 : 0,45\}$			0,41
d_6	256 valeurs tirées uniformément dans $\{1,8 : 1,9\}$	0,13	0,17	0,18
d_7	256 valeurs tirées uniformément dans $\{1,2 : 1,3\}$			0,56

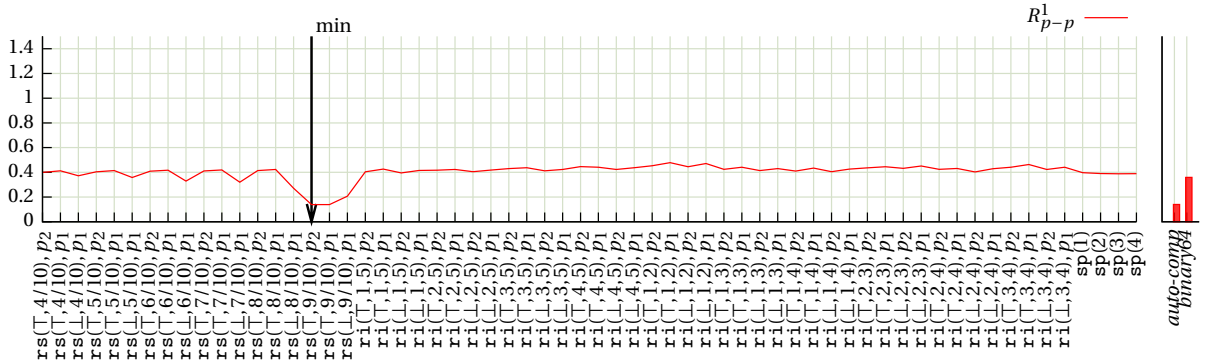
TABLE 8.6 – Présentation des données avec leurs ratios r_{cycles} et $r_{\text{bits exacts}}$ pour le programme 6.11 CLENSHAWI en *binary64* dans les environnements A.1 et A.2.

La figure 8.7 présente les résultats de la synthèse de code avec les données d_4 dans l'environnement A.1. Elle présente les stratégies gagnantes désignées par les sélections des différents ratios performances-précision (voir chapitre 7). La figure 8.7(a) montre la sélection suivant le ratio orienté précision $R^0_{\text{p–p}}$. La stratégie retenue par cette sélection correspond à la compensation automatique. Les figures 8.7(b) et 8.7(c) montrent respectivement les sélections suivant les ratios orientés performances $R^1_{\text{p–p}}$ et $R^2_{\text{p–p}}$. Les stratégies retenues par ces sélections correspondent, dans ce cas, au programme transformé suivant la stratégie de compensation partielle $\texttt{rs}(\bot, 9/10)$ avec la propagation p_2 pour $R^1_{\text{p–p}}$ et $\texttt{rs}(\bot, 9/10)$ avec la propagation p_1 pour $R^2_{\text{p–p}}$. L'ensemble des résultats est présenté entièrement au tableau 8.7.

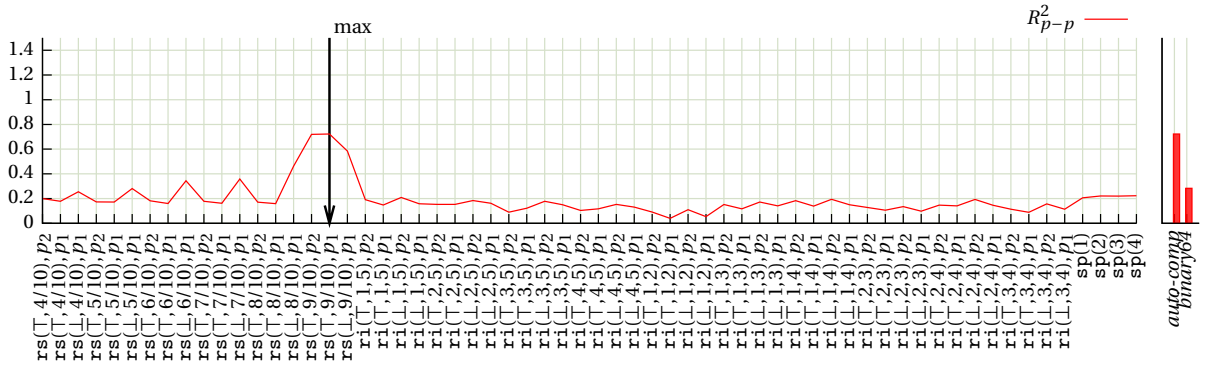
Le tableau 8.7 présente les résultats obtenus par la synthèse de code (des résultats complémentaires sont aussi donnés en annexe au tableau D.3). Pour chaque sélection et jeux de données, les tableaux en donnent les stratégies gagnantes et leurs ratios r_{cycles} et $r_{\text{bits exacts}}$. Les tableaux indiquent de plus si la synthèse de code s'est opérée avec succès comme spécifiée par la définition 8.1.1. Là encore, nous remarquons que la stratégie $\texttt{rs}(\bot, 9/10)$ apporte beaucoup de précision, mais le compromis avec les performances pourrait être plus poussé. Tout comme l'exemple précédent dans le cas de l'évaluation polynomiale avec HORNER, il est possible de choisir des stratégies de transformations partielles apportant moins de précision dans le but de proposer des programmes plus performants.



(a) Sélection par le ratio $R^0_{\text{p–p}}$.



(b) Sélection par le ratio $R^1_{\text{p–p}}$.



(c) Sélection par le ratio $R^2_{\text{p–p}}$.

FIGURE 8.7 – Détails des ratios performances–précision de l'algorithme CLENSHAWI et les données d_4 dans l'environnement A.1. Les stratégies gagnantes retenues sont indiquées par une flèche.

Ratio	Données	Stratégie gagnante	r_{cycles}	$r_{\text{bits exacts}}$	Succès
		Environnement A.1			
$R^0_{\text{P-P}}$	d_4 d_6 d_7	*auto-comp*	0,28 0,27	1	✓
$R^1_{\text{P-P}}$	d_4 d_6 d_7	$\texttt{rs}(\top, 9/10), p_2$ $\texttt{rs}(\top, 9/10), p_1$ $\texttt{rs}(\top, 9/10), p_2$	0,27	1	✓
$R^2_{\text{P-P}}$	d_4 d_6 d_7	$\texttt{rs}(\top, 9/10), p_1$	0,27	1	✓
		Environnement A.2			
$R^0_{\text{P-P}}$	d_4 d_6 d_7	*auto-comp*	0,33	1	✓
$R^1_{\text{P-P}}$	d_4 d_6 d_7	$\texttt{rs}(\top, 9/10), p_2$	0,33	1	✓
$R^2_{\text{P-P}}$	d_4 d_6 d_7	$\texttt{rs}(\top, 9/10), p_2$ $\texttt{rs}(\top, 9/10), p_1$	0,33	1	✓

TABLE 8.7 – Résultats de la synthèse de code de l'algorithme CLENSHAWI. Présentation des stratégies gagnantes, des ratios r_{cycles} et $r_{\text{bits exacts}}$ pour chaque jeux de données du tableau 8.6 et sélections performances–précision dans les environnements A.1 et A.2.

Temps pour la synthèse Le processus complet de la synthèse de code prend en moyenne 1 minute et 12 secondes pour les données d_4, d_6 et d_7 dans l'environnement A.1. De plus, il en faut en moyenne 1 minute et 30 secondes dans l'environnement A.2.

8.2.3 Récapitulatif des résultats et conclusion

Le tableau 8.8 donne le récapitulatif des résultats de la synthèse de code sur l'ensemble des cas étudies dans cette section. Pour chaque cas, nous donnons les résultats concernant deux jeux de données dans deux environnements différents. Les polynômes p_H, p_C, p_D sont définit respectivement aux pages 126, 131 et 83. Les données d_1 à d_6 sont définies dans

les tableaux 8.2, 8.4 et 8.6. Les jeux de données d_8 et d_9 correspondent à deux intervalles de 256 valeurs tirées uniformément dans $\{0,73 : 0,74\}$ pour d_8 et dans $\{0,755 : 0,8\}$ pour d_9.

En conclusion, nous remarquons que les stratégies de transformation par répartition simple `rs` sont plus à mêmes de produire des programmes partiellement transformés répondant aux compromis performances-précision. Plus particulièrement, c'est la transformation compensant partiellement 90% du programme qui est le plus souvent retenue par la synthèse. Dans certains cas (comme pour le cas de l'algorithme CLENSHAW), le compromis avec le temps de calcul n'est pas suffisamment exploité. Néanmoins, nous pouvons nous reporter aux stratégies classées en deuxième, troisième, n-ième position afin d'avoir un programme répondant plus fortement aux compromis temps-précision. Cependant, les transformations par répartition intermittente `ri` ne sont pas inutiles et sont capables dans certains cas d'être meilleures que les transformations `rs`. Par exemple, nous avons vu dans le cas de somme, que les transformations `ri` (correctement paramétrées) parviennent à améliorer la précision plus efficacement que les répartitions `rs`. La seconde raison qui fait que les transformations par répartition simple apparaissent plus souvent, est que dans l'outil de synthèse *SyHD*, ce sont les stratégies par répartition simple qui sont considérées en premier, et donc prioritaires lors de la sélection de la stratégie gagnante. Enfin, un paramétrage différent lors de la génération de l'ensemble de recherche de la synthèse permettrait des meilleurs résultats pour les transformations par répartitions intermittentes `ri`. Nous pourrions par exemple, à la place d'utiliser la transformation `ri`$(\rho,2,3)$, utiliser la transformation `ri`$(\rho,4,6)$. Le nombre d'opérations élémentaires transformées serait exactement le même, mais permettrait, a priori, un meilleur gain en précision. Cet aspect de paramétrage de l'ensemble de recherche pourra être étudié ultérieurement.

Concernant la sélection de la stratégie gagnante, nous remarquons que les ratios performances-précision, donnent de bons résultats. La sélection $R^0_{\text{p–p}}$, permet dans tous les cas de trouver le programme transformé permettant le maximum de précision pour le minimum de temps de calcul. Dans tous les cas étudiés, le programme sélectionné correspond au programme compensé. Il permet donc une précision doublée, à l'image des *double-double*, mais avec de meilleurs temps de calcul. Les ratios orientés performances, $R^1_{\text{p–p}}$ et $R^2_{\text{p–p}}$ donnent dans la majorité des cas les mêmes résultats. Cependant, il arrive quelques fois, que l'un ou l'autre donne un résultat erroné. Néanmoins les faiblesses intrinsèques à l'un sont compensés par l'autre. La comparaison des résultats de ces sélections permet, en cas de conflit, d'alerter l'utilisateur d'erreurs potentielles lors du choix du programme transformé par la synthèse. La comparaison de leurs résultats, facilitée par nos outils, permet néanmoins à l'utilisateur de sélectionner la bonne solution.

Algorithmes et données	Environnement A.1				Environnement A.2			
	Ratio de sélection				Ratio de sélection			
	$R^0_{\text{p–p}}$	✓/✗	$R^1_{\text{p–p}}/R^2_{\text{p–p}}$	✓/✗	$R^0_{\text{p–p}}$	✓/✗	$R^1_{\text{p–p}}/R^2_{\text{p–p}}$	✓/✗
SUM(d_1)	*auto-comp*	✓	ri($\top, 10^3, 5\cdot10^3), p_2$	✗•	*auto-comp*	✓	ri($\bot, 10^3, 5\cdot10^3), p_1$	✗•
SUM(d_2)	*auto-comp*	✓	ri($\top, 10^4, 5\cdot10^4), p_2$	✗•	*auto-comp*	✓	ri($\bot, 10^4, 5\cdot10^4), p_1$	✗•
HORNER(p_H, d_4)	*auto-comp*	✓	rs($\top, 9/10), p_1$	✓	*auto-comp*	✓	rs($\top, 9/10), p_1$	✓
HORNER(p_H, d_5)	*auto-comp*	✓	rs($\top, 9/10), p_2$	✓	*auto-comp*	✓	rs($\top, 9/10), p_1$ †	✓
HORNERDER(p_H, d_4)	*auto-comp*	✓	sp(1)‡	✗•	*auto-comp*	✓	sp(1)‡	✗•
HORNERDER(p_H, d_5)	*auto-comp*	✓	rs($\top, 9/10), p_2$	✓	*auto-comp*	✓	rs($\top, 9/10), p_2$	✓
CLENSHAWI(p_C, d_4)*	*auto-comp*	✓	rs($\bot, 8/10), p_1$	✓	*auto-comp*	✓	rs($\bot, 7/10), p_1$	✓
CLENSHAWI(p_C, d_6)*	*auto-comp*	✓	rs($\top, 8/10), p_2$	✓	*auto-comp*	✓	rs($\top, 8/10), p_1$	✓
CLENSHAWII(p_C, d_4) *	*auto-comp*	✓	rs($\bot, 8/10), p_1$	✓	*auto-comp*	✓	rs($\bot, 8/10), p_1$	✓
CLENSHAWII(p_C, d_6) *	*auto-comp*	✓	rs($\top, 8/10), p_2$	✓	*auto-comp*	✓	rs($\top, 8/10), p_2$	✓
DECASTELJAU(p_D, d_8)	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°
DECASTELJAU(p_D, d_9)	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°
DECASTELJAUDER(p_D, d_8)	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°
DECASTELJAUDER(p_D, d_9)	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°	*auto-comp*	✓	rs($\bot, 9/10), p_1$	✗°

TABLE 8.8 – Récapitulatif des résultats de la synthèse de code sur l'ensemble des cas étudiés à la section 8.2. Détails pour chaque cas selon deux jeux de données et les environnements A.1 et A.2.

• Échec dû au manque de précision (voir définition 8.1.1).

° Échec dû au manque de performances (voir définition 8.1.1).

* Les stratégies rs($\top$,9/10) ne sont pas retenues car elles fournissent autant de précision que le programme en *auto-comp* pour trop peu de gain en temps de calcul.

† Résultat retenu après conflit avec la stratégie ri($\top, 4, 5), p_1$ sélectionnée par $R^2_{\text{p–p}}$.

‡ Il existe une meilleure solution qui répond à la définition 8.1.1 avec succès : la stratégie rs($\bot$,7/10), p_1. Les sélections par les rations $R^1_{\text{p–p}}$ et $R^2_{\text{p–p}}$ ont toutes deux échouées dans ce cas.

8.3 Résolution de systèmes linéaires et raffinement itératif

Nous présentons à présent des exemples d'amélioration automatique de la précision pour des algorithmes basés sur le raffinement itératif. Dans la section 8.3.1, nous nous intéressons plus particulièrement à la résolution de systèmes linéaires.

8.3.1 Résolution de systèmes linéaires

Le raffinement itératif est une méthode classique permettant d'améliorer la précision des résultats d'un algorithme. Cette méthode est souvent utilisée pour l'amélioration de la solution calculée $\hat{x}$ d'un système d'équations linéaires $Ax = b$. Dans notre cas, A est une matrice de taille $n \times n$ et b un vecteur de taille $n \times 1$ [Neu01, Hig02].

Jusqu'à ce que la solution calculée $\hat{x}$ soit suffisamment précise, une itération du raffinement consiste à :

1. calculer le résidu $r = A\hat{x} - b$,
2. résoudre le système $Ad = r$,
3. et mettre à jour la solution $y = \hat{x} - d$ puis répéter si nécessaire depuis l'étape 1 avec $\hat{x} = y$.

S'il n'y avait pas d'erreurs d'arrondis dans le calcul de r, d et y, alors y serait la solution exacte du système. L'idée générale du raffinement itératif est que si r et d sont calculés avec suffisamment de précision alors la solution verra sa précision améliorée. En règle générale, le raffinement itératif est utilisé avec une résolution directe de systèmes linéaires, comme la factorisation de GAUSS. Pour une convergence rapide, le résidu r, est calculé en précision étendue avant d'être arrondi à la précision de travail.

L'algorithme 8.1 proposé par WILKINSON et al. dans les années 1960 [BMPW66] permet d'effectuer la résolution de tels systèmes.

L'analyse faite par WILKINSON montre que si le résidu r est calculé avec le double de la précision de travail u (voir section 1.4.1), et que si A n'est pas trop mal conditionnée, alors la précision de la solution $x^{(i)}$ est équivalente à la précision de travail. La vitesse de convergence de l'algorithme dépend alors du conditionnement de A.

Considérons l'algorithme GEPP (*Gaussian Elimination with Partial Pivoting*) pour la résolution de système linéaire : élimination de GAUSS avec pivot partiel. Nous avons alors étudié différents cas de raffinement itératif selon que le résidu r est calculé avec et sans précision étendue. Nous distinguerons alors trois cas :

1. Le calcul de r est effectué dans la précision de travail à l'aide des BLAS [BDD+02]. Les BLAS sont des routines optimisées de calculs de l'algèbre linéaire. La précision utilisées dans cette expérience est le *binary64*.

Entrée(s) : Une matrice A de taille $n \times n$ et un vecteur b de taille $n \times 1$.
Sortie(s) : Le vecteur des solutions $x^{(i)}$ approchant x dans $Ax = b$.

1. Résoudre $Ax^{(1)} = b$ avec une méthode directe
2. $i = 1$
3. **répéter**
4. Calculer le résidu $r^{(i)} = Ax^{(i)} - b$
5. Résoudre $Ax^{(i+1)} = r^{(i)}$ avec une méthode directe
6. Mettre à jour $x^{(i+1)} = x^{(i)} - x^{(i+1)}$
7. $i = i + 1$
8. **jusqu'à** $x^{(i)}$ soit « suffisamment précis »
9. **retourner** $x^{(i)}$

Algorithme 8.1 – Raffinement itératif pour la résolution de systèmes linéaires (WILKINSON et al. [BMPW66]).

2. Le calcul de r est effectué avec le double de précision, c'est-à-dire en *double-double* à l'aide des routines XBLAS [LDB+02] (les XBLAS sont un sous-ensemble des BLAS travaillant en précision étendue). C'est une méthode couramment employée pour la résolution de systèmes linéaires [DHK+06].
3. Le calcul de r est effectué avec du code ayant été automatiquement compensé par notre méthode d'amélioration de la précision.

Nous faisons référence à ces cas en les nommant respectivement BLAS, XBLAS et AC. Ces expériences permettent de montrer comment notre méthode AC se compare à la technique utilisant les XBLAS. Nous ferons ces comparaisons en terme de précision et de temps de calcul. L'algorithme étant itératif, la fin du processus de raffinement se produit, soit lorsque la solution calculée converge avec une précision de l'ordre de u, ou soit lorsque un nombre d'itérations maximum défini arbitrairement est atteint.

Formellement, le raffinement itératif est stoppé lorsque l'une des deux conditions suivantes est remplie :

1.
   - Dans le cas où le résidu r est calculé en précision fixe (i.e. avec la précision de travail), alors le raffinement itératif s'arrête lorsque l'erreur relative inverse $\omega_{|A|,|b|}(\hat{y})$ est de l'ordre de u [Hig02], avec
   $$\omega_{|A|,|b|}(\hat{y}) = \max_j \frac{r_j}{(A|\hat{y}| + b)_j}, \tag{8.1}$$
   et où $\hat{y}$ est la solution corrigée et $\xi/0$ est interprété comme zéro si $\xi = 0$ ou l'infini sinon.
   - Dans le cas où le résidu r est calculé en précision étendu (i.e. avec le double de précision), alors le raffinement itératif s'arrête lorsque l'évolution des itérés

successifs e définie par l'équation (8.2) est de l'ordre de u.

$$e = \frac{\|x^{i+1} - x^i\|_\infty}{\|x^{i+1}\|_\infty} \tag{8.2}$$

2. Le nombre d'itération dépasse $i_{\texttt{max}} = 25$ itérations.

Nous utilisons ces conditions d'arrêt à la ligne 8 de l'algorithme 8.1 en utilisant les relations (8.1) et (8.2) selon la précision utilisée pour le calcul de r ; à savoir la précision fixe pour la première et la précision étendue pour la seconde. Les données utilisées dans ces expériences proviennent de la *boîte à outils Matlab, The Matrix Computation Toolbox* créée par HIGHAM [Hig]. Cette boîte à outils : permet de générer des systèmes linéaires. Nous avons généré différents systèmes $Ax = b$ où :

- $A = \texttt{orthog}(25)$, est une matrice carrée orthogonale de dimension $n = 25$;
- $A = \texttt{clement}(50)$, est une matrice carrée tridiagonale de dimension $n = 50$ où la diagonale principale est composée de zéros ;
- $A = \texttt{gfpp}(50)$, est une matrice carrée de dimension $n = 50$ donnant un facteur de croissance maximal lors de l'élimination gaussienne avec pivot partiel ;

et où dans chaque cas, b est un vecteur dont les valeurs sont choisies aléatoirement selon une distribution uniforme dans l'intervalle $[0, 1]$.

La figure 8.8 montre 50 exécutions du raffinement itératif pour $Ax = b$ quand A est une matrice $\texttt{orthog}(25)$, $\texttt{clement}(50)$ ou $\texttt{gfpp}(50)$ et où b est distinct à chaque exécution. L'axe des abscisses représente 50 exécutions. L'axe des ordonnées de gauche représente, le nombre d'itérations requis pour satisfaire que les conditions de précision des relations (8.1) et (8.2) soit inférieures ou égales à $4u$. Cette valeur a été retenue car prendre une condition plus ou moins forte ne permet pas d'observer correctement les résultats de l'expérience. Avec une condition trop faible, $10u$ par exemple, le raffinement itératif s'arrête au bout d'une ou deux itérations dans tous les cas. Au contraire, dans beaucoup de cas, prendre une valeur trop forte, u par exemple, fait que le raffinement itératif s'arrête seulement lorsque le nombre d'itération maximal est atteint. L'axe des ordonnées de droite, donne le nombre de chiffres perdus dans la solution x du système (en *double-double* le nombre maximal de chiffres perdus est 16). Comme prévu, la résolution du système avec un calcul du résidu en précision double nécessite moins d'itérations qu'en précision fixe. La figure illustre de plus que le calcul du résidu en précision étendue à l'aide de notre approche permet d'obtenir la même précision que lorsque le résidu est calculée à l'aide des XBLAS. Cette précision est fournie de plus avec un nombre d'itérations identique que ce soit avec l'utilisation de la solution utilisant les XBLAS ou le code automatiquement compensé AC. Des mesures de nombre de cycles sont présentées au tableau 8.9

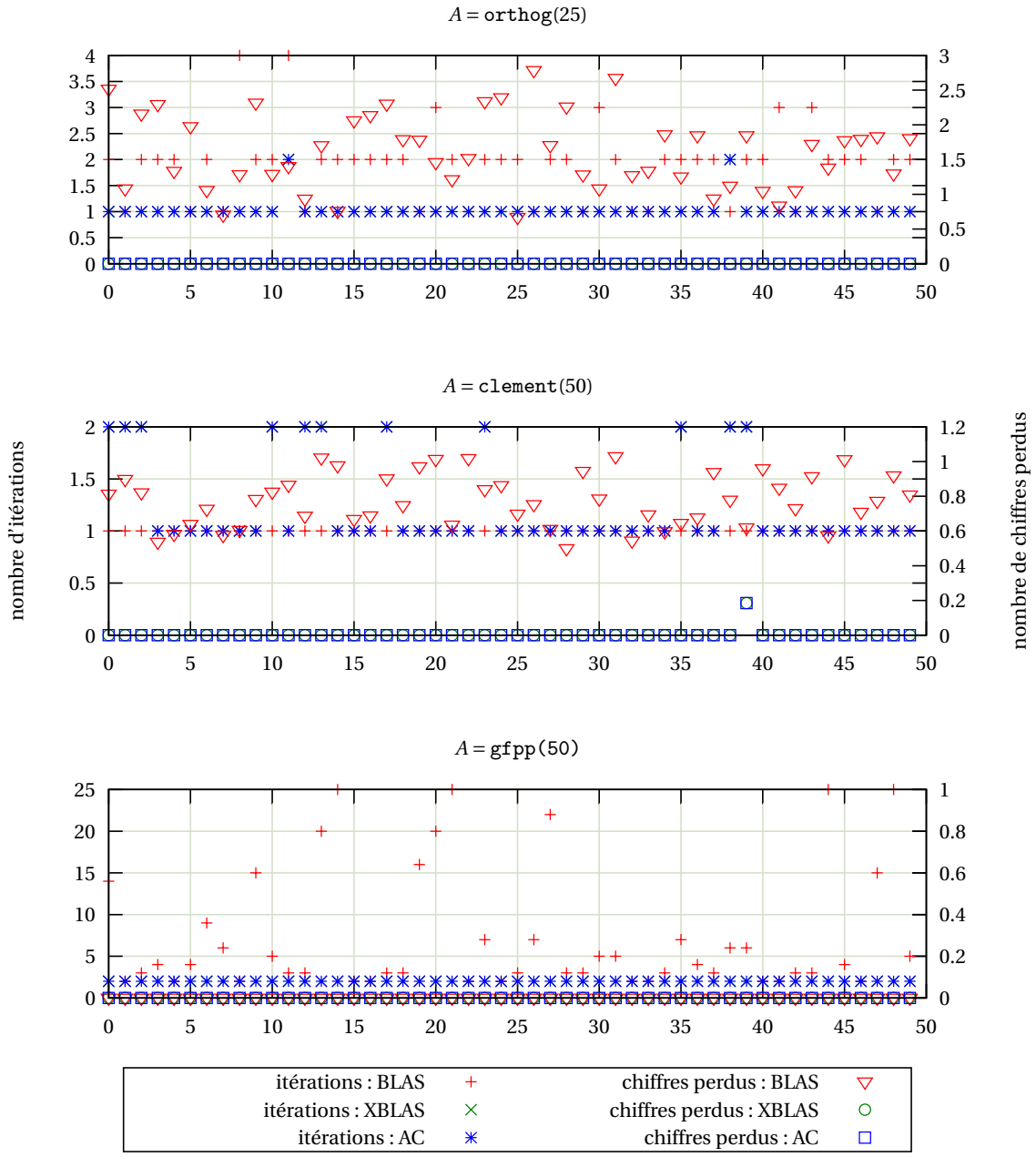


FIGURE 8.8 – 50 exécutions du raffinement itératif de $Ax = b$ quand A est la matrice `orthog`(25), `clement`(50) et `gfpp`(50).

La figure 8.9 donne un exemple détaillé de la précision de la solution calculée en fonction des itérations de l'algorithme pour le système $Ax = b$ donné à la figure 8.8 où A est la matrice `gfpp` de l'exécution 6 présente sur l'axe des abscisses. Le critère d'arrêt est atteint lorsque l'erreur calculée suivant les relations (8.1) et (8.2) est inférieure à $4u$. Nous observons que les erreurs des algorithmes utilisant la précision étendue s'annulent dès la seconde itération. Le raffinement est donc stoppé après la deuxième itération. En revanche, l'algorithme utilisant la précision fixe a besoin de 9 itérations pour que l'erreur entachant la solution soit inférieure à $4u$.

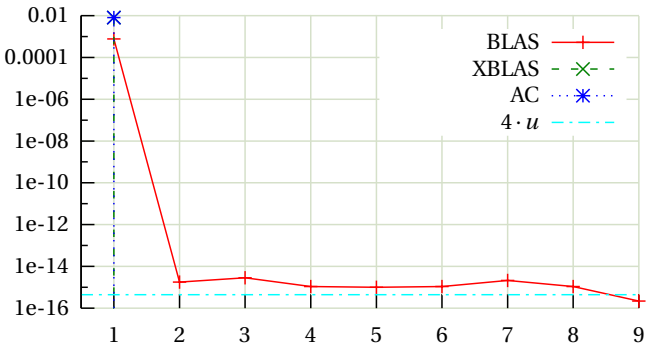


FIGURE 8.9 – Détails de la précision de la solution calculée en fonction du nombre d'itérations de l'algorithme 8.1 pour le système $Ax = b$ donné à la figure 8.8 où A est la matrice `gfpp` (voir exécution numéro 6).

Le tableau 8.9 donne les temps d'exécutions mesurés dans l'environnement A.1 avec PAPI pour la résolution des systèmes linéaires. Les temps sont donnés en kilocycles. Sur l'ensemble des données étudiés nous observons que le code automatiquement compensé AC présente des performances proches à la version de l'algorithme utilisant les XBLAS. Ceci pourrait s'expliquer par le fait que les XBLAS exploitent au mieux les optimisations de compilation. Néanmoins, de façon complètement automatique, notre approche est capable de donner les mêmes résultats pour un coût relativement proche. On observe de plus que lorsque les systèmes sont mal conditionnées, les versions de l'algorithme utilisant la précision étendue AC et XBLAS présentent de meilleures performances que la version utilisant la précision fixe. Ceci est dû au nombre d'itérations moindres nécessaires à la résolution du système lorsque le résidu est calculé en précision étendue.

8.4 Conclusion

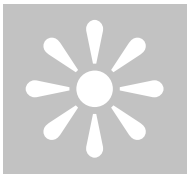
Avec ce chapitre, nous avons illustré les performances de notre approche dans la transformation automatique de programme permettant de répondre aux attentes d'utilisateurs en terme de temps de calcul et de précision. Ces transformations génèrent des programmes

temps mesuré	min			moyen			max		
	BLAS	XBLAS	AC	BLAS	XBLAS	AC	BLAS	XBLAS	AC
`orthog`(25)	388	483	496	393	552	568	418	790	816
`clement`(50)	560	746	773	1575	751	778	5025	759	790
`gfpp`(50)	86	109	112	127	114	116	222	180	186

TABLE 8.9 – Temps d'exécution (minimal, moyen et maximal en kilocycles) pour la résolution des 50 systèmes linéaires mesurés avec PAPI (chaque mesure est une moyenne sur 10 000 exécutions).

partiellement compensés permettant d'améliorer la précision tout en minimisant l'impact sur les performances. Ces programmes, en terme de précision et de temps de calcul, sont situés entre le programme original (en *binary64*) et les programmes en précision étendue (*double-double*). Notre méthode permet alors d'exploiter cet espace de façon automatique pour générer des programmes ayant des caractéristiques de temps et de précision voulues par un utilisateur et réalisant un meilleur compromis performances–précision que les solutions *double-double* ou *binary64*.

Les résultats montrent en outre que notre prototype ne permet pas toujours de trouver automatiquement un programme répondant au compromis désiré. Néanmoins, notre prototype permet dans ces cas là une recherche manuelle pour trouver une solution avec de bonnes caractéristiques, si elle existe. Enfin, pour la totalité des expériences menées dans ce chapitre, il a toujours été possible d'améliorer automatiquement, ou dans le pire des cas égaler, la précision et les performances comparativement aux méthodes actuelles d'amélioration de la précision.



CONCLUSION ET PERSPECTIVES

Résumé du travail de thèse

Temps de calcul vs. précision : le cas de somme comme point de réflexion

Nous avons étudié au chapitre 4 les relations existantes entre performances et précision de l'évaluation d'une somme de n termes, au travers d'une étude exhaustive des réécritures possibles. Bien que la grande combinatoire des réécritures nous limite à l'étude des sommes de 10 termes au maximum, nous avons montré qu'un schéma entre performances et précision se répétait. Ce schéma caractérise la répartition des différentes réécritures en fonction de leur temps de calcul (caractérisé par la longueur du chemin critique de la réécriture), et de leur précision, pour un ensemble de jeux de données représentatif. Nous avons remarqué que cette répartition semble suivre une répartition gaussienne (qu'il serait intéressant de prouver) centrée ou légèrement décalée vers les valeurs d'erreurs les plus faibles. Cela signifie que pour les réécritures d'une somme de n termes, le plus grand nombre d'entre elles présente une erreur située entre les erreurs minimales et maximales. En comparaison, le nombre de réécritures présentant une petite (ou grande) erreur est faible. Nous avons ensuite étudié les performances de ces réécritures, c'est-à-dire la longueur du chemin critique de l'expression. Nous avons montré que plus les performances augmentent, plus le nombre de réécritures présentant les plus petites erreurs diminuait. Elles tendent même à disparaître. De plus, nous constatons qu'en pratique, les architectures actuelles limitent le niveau de parallélisme d'instruction. De ce fait, il existe des réécritures présentant un niveau parallélisme non optimal, disposant d'un maximum de précision, qui s'exécutent à la même vitesse que des réécritures complètement parallèles (mais présentant moins de précision). Ce résultat nous a permis de mettre en évidence les

relations entre performances et précision. Il a de plus justifié la suite de notre travail sur l'obtention automatique d'un compromis performances–précision.

De la compensation automatique de programme en arithmétique flottante

Nous avons développé une méthode pour améliorer la précision numérique des programmes utilisant l'arithmétique flottante en automatisant la compensation (voir chapitre 5). Cette approche présente en effet un fort potentiel de parallélisation comparativement aux expansions (i.e. les *double-double*). Dans l'optique de contrôler au mieux l'effet de l'amélioration de la précision sur les performances, notre choix s'est alors naturellement dirigé vers cette solution. Nous avons alors développé des algorithmes permettant de compenser automatiquement les opérations d'additions et de multiplication. Les programmes générés sont alors plus performants que les programmes utilisant les expansions à précision équivalente. Nous avons employé notre approche sur un ensemble de problèmes présents dans la bibliographie [ROO05, GLL09, JBL+11]. Elle comporte un algorithme de sommation classique et des algorithmes d'évaluations polynomiales pour lesquels la communauté scientifique a développé des algorithmes compensés. Nous avons alors comparé les résultats obtenus grâce à notre méthode aux algorithmes trouvés dans la littérature et aux mêmes algorithmes en *double-double*. Ces mesures ont permis de montrer que la compensation automatique de ces programmes donne les mêmes résultats en terme de précision que leurs versions compensées manuellement et *double-double*. Ces mesures ont montré que les performances des algorithmes compensés automatiquement et manuellement étaient équivalentes. De plus, nous constatons que les algorithmes compensés étudiés sont en règle générale deux à trois fois plus rapide en pratique que les algorithmes en *double-double*. Nous avons ensuite étudié d'autres cas pour lesquels la précision des calculs est cruciale et affecte directement les performances tel que le raffinement itératif pour la résolution de système linéaire. Là encore, la compensation automatique permet de retrouver la précision des algorithmes existants.

Des transformations source à source pour des compromis performances–précision

Pour aller plus loin, nous avons développé trois méthodes de transformation de code pour appliquer partiellement la compensation. Différentes stratégies permettent de générer des programmes suivant des compromis performances–précision (voir chapitre 6). Deux d'entre elles opèrent des transformations au sein des boucles de calcul. Elles se basent sur des techniques de déroulage et de fission. Grâce à ces stratégies, nous pouvons générer un grand nombre de programmes équivalents permettant des compensations partielles différentes. La troisième stratégie, opère quant à elle, directement aux endroits suscep-

tibles de provoquer les plus grosses erreurs de calculs pour les compenser. Ces stratégies permettent donc la génération de nombreux programmes transformés présentant des caractéristiques distinctes de performances et précision. Nous avons ensuite vu que le choix et le paramétrage de ces stratégies est a priori difficile, car non intuitif et dépendant de divers facteurs tels que l'environnement utilisé ou encore les données. Nous avons alors utilisé la synthèse de code pour effectuer de façon automatique le choix correspondant le mieux à des spécifications de performances et de précision (voir chapitre 7). Celle-ci nécessite qu'un ensemble de paramètres soient fixés afin de pouvoir proposer des programmes transformés. Certains de ces paramètres doivent être fournis par l'utilisateur, tels que les attentes en termes de précision et de performances ou un jeu de données représentatif des données utilisées par le programme à améliorer. De plus, nous avons développé un espace de recherche, c'est-à-dire un ensemble de programmes transformés suivant différents paramétrages des stratégies d'application partielle de la compensation. Enfin, nous proposons plusieurs techniques de recherche permettant de fournir un programme (parmi l'ensemble des programmes de l'espace de recherche) répondant au mieux aux critères de performances et de précision.

Application à des problèmes classiques de précision numérique

Durant cette thèse nous avons cherché à comparer nos résultats vis-à-vis de la précision et des performances d'autres méthodes d'amélioration de la précision, et en particulier, les plus performantes : les compensations, et les plus accessibles : les expansions. Ces méthodes permettent de doubler la précision numérique des résultats. D'un côté les compensations, efficaces mais destinées à un public de spécialistes, et de l'autre, les *double-double* facilement accessibles et automatiques. Nous avons répertorié les exemples présents dans la littérature : algorithmes compensés pour la somme [ROO05] et certaines évaluations polynomiales [GLL09, JGH+13, JLCS10, JBL+11]. En procédant ainsi, nous avons pu montrer que nos programmes automatiquement compensés présentaient la même précision que les programmes compensés de la littérature. Les performances sont quant à elles, en pratique, équivalentes à quelques cycles près (voir chapitre 5).

Nous avons ensuite appliqué nos techniques d'amélioration partielle de la précision pour générer des programmes encore plus performants. Nous avons montré qu'il est possible d'exploiter automatiquement l'espace des programmes présentant une précision et des performances comprises entre celles du programme original et celles du programme entièrement compensé (disposant du double de précision). Ceci permet d'obtenir des programmes améliorant la précision tout en contrôlant l'impact sur les performances. Enfin, nous avons appliqué notre approche sur un problème qui fait l'objet de travaux concernant sa qualité numérique, mais pour lequel une solution compensée n'existe pas encore : la résolution de systèmes linéaires par raffinement itératif [DHK+06]. Les résultats obtenus sur le cas étudié montrent que nous sommes capables de retrouver la même précision que son équivalent utilisant les expansions (voir chapitres 6 et 8).

CoHD et SyHD, des outils automatiques pour la synthèse de code.

Enfin, pour automatiser cette approche, nous avons développé en OCAML des outils permettant la synthèse de code (voir chapitre 7). Ils effectuent automatiquement la transformation d'un programme écrit en C. Le programme automatiquement généré dépend de paramètres tel que : l'architecture, le compilateur, de données représentatives ou de critères de précision et de performances voulus. Ces outils permettent aussi une étude des programmes, dans des buts d'optimisation, de profilage, ou simplement d'observation des différentes stratégies d'optimisations. Ce type d'étude est facilité par différentes sorties telles que des graphes de résultats générées automatiquement avec GNUPLOT ou par l'exportation de l'arbre de syntaxe abstrait en DOT (GRAPHVIZ).

Perspectives

Les travaux effectués dans le cadre de cette thèse ouvrent la voix à de nombreuses perspectives complétant et améliorant notre approche. Ces différentes perspectives sont principalement dirigées (i) vers une intégration plus large des constructions du langage C, (ii) vers une amélioration plus poussée de la précision, et (iii) vers des processus d'optimisation, de transformations et de synthèse plus variés et plu efficaces.

Vers l'amélioration automatique d'un ensemble plus large d'opérations.

Lors de ces travaux nous nous sommes intéressés à la compensation des opérations élémentaires de la somme et du produit. Nous avons vu que des méthodes d'approximations permettent d'évaluer les erreurs des opérations de division [PV93] ou de racine carré [Mar90]. Il existe de plus d'autres méthodes, comme par exemple la méthode de NEWTON pour la division, qui permettent d'effectuer des opérations élémentaires avec plus de précision. Naturellement, une perspective serait d'intégrer le support de ces opérations, à travers les compensations ou autres méthodes, afin d'améliorer la précision de toutes les opérations élémentaires. L'ajout du support de ces opérations permettra alors de développer de nouvelles stratégies d'amélioration avec compromis performances–précision tenant en compte les coûts relativement faibles de la compensation de la somme ou encore du produit par rapport aux transformations de la division et de la racine carrée.

La seconde perspective serait de compléter l'intégration complète du langage de programmation source. Le langage supporté actuellement est le C, et certaines structures de contrôles ne sont pas gérées par nos prototypes CoHD et SyHD. Par exemple, l'intégration des structures comme les boucles *faire tant que* ou encore les appels de fonctions pourrait s'avérer utile.

Vers une amélioration automatique de la précision non bornée.

Notre approche permet d'améliorer automatiquement la précision des programmes utilisant l'arithmétique flottante en employant des compensations. Plus précisément, le programme est compensé une fois. Les algorithmes compensés peuvent cependant, à l'instar des algorithmes utilisant les expansions, être compensés n fois. C'est-à-dire que la précision est $k = n+1$ fois supérieure à la précision de calcul. Par exemple, l'algorithme de sommation SUM2 est dérivé d'un algorithme plus général SUMK [ROO05] permettant de compenser $n = k-1$ fois l'algorithme de sommation original. Cette démarche pourrait présenter des similarités au fonctionnement des expansions et nous pourrions dans ce cas obtenir les mêmes conclusions faites dans le cadre de nos travaux. Par exemple, un algorithme utilisant les expansions et quadruplant la précision (i.e. les *quad-double* [HLB01]) pourrait présenter le même niveau de précision qu'un algorithme compensé trois fois ($k = 4$) avec cependant un meilleur niveau de parallélisme.

De ce point de vue, l'amélioration automatique de la précision, via la transformation complète d'un programme pourrait se faire autant de fois que nécessaire jusqu'à l'obtention d'une précision suffisante, et ce pour de meilleures performances que les techniques automatiques actuelles. Malgré ce premier compromis performances–précision, nous pourrions vouloir exercer un contrôle plus fin sur les performances en appliquant les n « étapes » de compensations suivant de nouvelles stratégies. Deux premières possibilités sont :

(i) L'application des compensations par couches successives, les $n-1$ premières couches sont appliquées sur l'ensemble des opérations élémentaires concernées, puis la synthèse de code applique les stratégies développées sur la couche n pour générer des programmes partiellement compensés n fois.

(ii) L'application des compensations se fait partiellement par couches. De ce fait les couches de compensations supérieures s'appliqueraient uniquement si la couche inférieure est transformée. Cette approche nécessite le développement de nouvelles stratégies de transformation de programme. Dans ce cas, le nombre de couches pourrait dépasser n tant que les performances seraient inférieures à l'algorithme n fois compensé entièrement.

Vers de nouvelles stratégies de transformation de programme.

Dans le cadre de l'optimisation multi-critères performances–précision, nous avons développé trois stratégies de transformation de programme permettant un contrôle fin sur les performances. Ces stratégies permettent, à l'aide de différents paramétrages, de générer des programmes ayant des propriétés de précision différentes. De ce fait, les programmes ainsi transformés possèdent des caractéristiques de précision et de performances présentant des compromis différents. Ces stratégies nécessitent de plus une gestion particulière

de la propagation des erreurs. Ces points pourraient faire l'objet de nombreuses améliorations. Par exemple, nous pourrions mixer les deux stratégies actuelles de transformation de boucle pour créer une nouvelle stratégie. Nous pourrions aussi améliorer la gestion des boucles imbriquées en développant des stratégies différentes selon les niveaux d'imbrication des boucles. De plus, la stratégie de sélection par précision peut prendre diverses formes. En effet, cette stratégie qui consiste à transformer les opérations causant les erreurs les plus importantes, implique de définir des notions a priori difficiles à déterminer. Comment déterminer les erreurs : selon l'erreur absolue ou l'erreur relative ? De plus, dans le cas des structures de contrôle (les boucles ou encore les blocs si-alors-sinon) il est possible de considérer les erreurs de différentes façons. Par exemple, dans une boucle de i itérations, chaque opération sera effectuée i fois. Dans ce cas, quel choix serait judicieux afin de déterminer l'opération causant le plus d'erreurs, et donc l'opération de la boucle à transformer. Ce choix peut se porter sur le maximum des erreurs, la moyenne, ou encore un intervalle de valeurs...

Outre ces améliorations, les perspectives les plus importantes seraient de développer de nouvelles stratégies de transformation et de propagation des erreurs. Nous pourrions par exemple, en plus des transformations dans la structure du programme, nous permettre certaines réécritures (d'expressions arithmétiques par exemple) permettant soit, une amélioration des performances, soit de la précision, ou idéalement des deux.

Vers l'intégration de nouveaux critères d'optimisation.

L'optimisation multi-critères ouvre de plus la voix à de nombreuses perspectives d'intégration de nouveaux critères. Cette intégration pourrait se faire soit via l'ajout d'un ou de plusieurs critères supplémentaires, tels que la consommation mémoire ou la taille du code généré ; soit par la possibilité de seulement sélectionner certains des critères disponibles. Nos travaux tiennent comptent des critères de précision et de performances en terme de nombre de cycles. Un critère supplémentaire pourrait être une gestion spécifique de certaines opérations. En effet, certaines architectures sont conçues pour exploiter au mieux certaines opérations (ou suite d'opérations). C'est pourquoi, dans certains cas comme celui des processeurs VLIW [JJLMR10] par exemple, l'optimisation du nombre de multiplications ou de carrés, est un critère important. Néanmoins, indépendemment de cela, il serait possible et intéressant d'intégrer de nouveaux critères, plus éloignés de la précision et des performances, tels que la consommation en ressources physiques (mémoire, bande passante, caches), énergétiques ou même la taille du code.

Vers une amélioration de la synthèse de code.

Enfin, nous terminerons par les perspectives concernant la synthèse de code et son amélioration. La synthèse de code, de par son action, utilise les différentes notions d'amélioration de la précision et de stratégies de transformations afin de générer des programmes ré-

pondant à des critères de performances et de précision. Les perspectives à ce sujet peuvent être séparées en deux groupes. D'abord (i) les perspectives d'amélioration, complétant les travaux effectuées ; et (ii), des perspectives à plus long terme.

(i) Nous limitons actuellement l'espace de recherche utilisé pour la synthèse à un ensemble de 62 programmes transformés suivant un paramétrage fixé. Une perspective serait d'améliorer cet espace de recherche, par un paramétrage plus fin des stratégies générant les codes transformés, voir par exemple, d'une génération dynamique à base d'heuristiques, reposant par exemple sur des algorithmes génétiques [Gol89].
Les spécifications données par l'utilisateur peuvent aussi être raffinées afin de mieux caractériser les programmes générés et déterminer le programme répondant au mieux aux critères imposés.
De plus, les perspectives précédemment évoquées vont demander l'amélioration des mesures faites automatiquement lors de la synthèse. Ces mesures concernent, d'un côté la précision, qui pourraient être effectuées à l'aide de MPFR [FHL+07] afin de baser le calcul des erreurs relativement aux calculs exacts. Par ailleurs, le calcul des performances d'un programme pourra être amélioré, ou effectué à l'aide d'autres méthodes et outils afin d'avoir des mesures en pratique toujours plus précises. De plus, à terme, l'utilisation de l'outil PERPI [GLPP12], pourrait de façon automatique, donner des indications sur les performances idéales d'un programme. Un nouveau paramètre de la synthèse serait alors le choix des méthodes de mesure.

(ii) Finalement, nous pourrions inclure de nouvelles techniques permettant une caractérisation différente du comportement des programmes étudiés. Par exemple, l'analyse statique de programme permettrait de mieux cerner le comportement vis-à-vis des données numériques d'un programme. Ceci serait utile dans un cadre de vérification et de certification.
Nous pourrions aussi utiliser notre méthode conjointement à d'autres, toujours dans le but d'enrichir l'espace et les techniques de recherche de la synthèse de code. Des travaux récents effectués au sein de notre équipe de recherche seraient à même de réaliser de telles perspectives [IM12]. Ces travaux concernent l'amélioration de la précision à base de réécritures d'expressions arithmétiques. Ces travaux, combinés aux nôtres, nous permettrait de développer de nouvelles stratégies de génération de code.

Troisième partie

Annexes

ANNEXE A

ENVIRONNEMENTS ET RESSOURCES DE TRAVAIL

Dans ce manuscrit, toutes les expérimentations dépendantes du matériel sur lequel elles sont effectuées font référence à un environnement. Ces environnements décrivent l'architecture et le système d'exploitation (table A.1) ou les logiciels utilisés précisant leur version et leurs options (table A.2).

Environnement A.1. *Cet environnement est composé des ressources (i) de la table A.1 et (i), (iv), (vi) de la table A.2.*

Environnement A.2. *Cet environnement est composé des ressources (ii) de la table A.1 et (ii), (v), (vi) de la table A.2.*

Environnement A.3. *Cet environnement est composé des ressources (i) de la table A.1 et (iii), (iv), (vi) de la table A.2.*

Environnement A.4. *Cet environnement est composé des ressources (ii) de la table A.1 et (iii), (v), (vi) de la table A.2.*

	Architecture et système d'exploitation
(i)	`Intel(R) Core(TM) i5 CPU M540 @ 2.53GHz` Linux 3.2.0.51-generic-pae i686 i386
(ii)	`AMD Athlon(tm) 64 X2 Dual Core Processor 4000+` Linux 3.2.0.51-generic-pae i686 athlon i386

TABLE A.1 – Architecture et systèmes d'exploitation.

	Logiciels	Version	Options
(i)	`gcc`	4.6.3	-O2 -mfpmath=sse -msse4
(ii)	`gcc`	4.6.3	-O2 -mfpmath=sse -msse2
(iii)	`clang`	3.4	-O1 -mfpmath=sse -msse4
(iv)	`papi`	5.1.0.2	$\emptyset$
(v)	`papi`	5.1.1	$\emptyset$
(vi)	`perpi`	$pilp5$	$\emptyset$

TABLE A.2 – Version et options des logiciels utilisés.

ANNEXE B

DÉTAILS DES MESURES DE PERFORMANCES RELATIVES AU CHAPITRE 3

	Environnement A.1			Environnement A.2		
	Instr. (dont *flop*)	Cycles	IPC	Instr. (dont *flop*)	Cycles	IPC
			$n = 10^2$			
SUM	519(99)	314	1,65	519(103)	422	1,23
SUM2	1 609(694)	726	2,22	1 609(1 194)	1 593	1,01
SUMDD	2 105(1 002)	2 368	0,89	2 105(1 688)	3 339	0,63
			$n = 10^3$			
SUM	5 019(999)	3 020	1,66	5 019(1 003)	4 022	1,23
SUM2	16 009(6 994)	7 026	2,28	16 009(11 994)	15 684	1,02
SUMDD	21 005(9 994)	23 992	0,88	21 005(16 988)	33 450	0,63
			$n = 10^4$			
SUM	50 019(9 999)	30 179	1,66	50 019(10 003)	40 076	1,25
SUM2	160 009(69 995)	70 561	2,27	160 009(119 994)	149 672	1,07
SUMDD	210 005(100 036)	241 369	0,87	210 005(169 988)	335 776	0,63
			$n = 10^5$			
SUM	500 019(99 999)	301 108	1,66	500 019(100 003)	657 650	0,76
SUM2	1 600 009(700 003)	708 910	2,26	1 600 009(1 199 994)	1 523 056	1,05
SUMDD	2 100 005(1 000 293)	2 402 912	0,87	2 100 005(1 699 988)	3 402 040	0,62

TABLE B.1 – *Suite à la page suivante*

Suite de la page précédente

	Environnement A.1			Environnement A.2		
	Instr. (dont *flop*)	Cycles	IPC	Instr. (dont *flop*)	Cycles	IPC
	$n = 10^6$					
SUM	5 000 019(1 000 296)	3 956 615	1,26	5 000 019(1 000 003)	6 679 371	0,75
SUM2	16 000 009(7 004 593)	7 817 610	2,05	16 000 010(11 999 994)	15 405 860	1,04
SUMDD	21 000 007(10 006 852)	24 760 352	0,85	21 000 009(16 999 988)	34 042 424	0,62

TABLE B.1 – Mesures de performances (instructions et cycles) des algorithmes SUM, SUM2 et SUMDD effectuées avec PAPI (moyenne sur 10^4 exécutions).

	Environnement A.1			Environnement A.2		
	Instructions	Cycles	IPC	Instructions	Cycles	IPC
	$n = 10^2$					
SUM	509	104	4,89	509	104	4,89
SUM2	1 599	210	7,61	1 599	210	7,61
SUMDD	2 095	897	2,33	2 095	897	2,33
	$n = 10^3$					
SUM	5 009	1 004	4,98	5 009	1 004	4,98
SUM2	15 999	2 010	7,95	15 999	2 010	7,95
SUMDD	20 995	8 997	2,33	20 995	8 997	2,33
	$n = 10^4$					
SUM	50 009	10 004	4,99	50 009	10 004	4,99
SUM2	159 999	20 010	7,99	159 999	20 010	7,99
SUMDD	209 995	89 997	2,33	209 995	89 997	2,33
	$n = 10^5$					
SUM	500 009	100 004	4,99	500 009	100 004	4,99
SUM2	1 599 999	200 010	7,99	1 599 999	200 010	7,99
SUMDD	2 099 995	899 997	2,33	2 099 995	899 997	2,33
	$n = 10^6$					
SUM	5 000 019	1 000 004	4,99	5 000 019	1 000 004	4,99
SUM2	15 999 999	2 000 010	7,99	15 999 999	2 000 010	7,99
SUMDD	20 999 995	8 999 997	2,33	20 999 995	8 999 997	2,33

TABLE B.2 – Mesures de performances (instructions et cycles) des algorithmes SUM, SUM2 et SUMDD effectuées avec PERPI.

ANNEXE C

CODES SOURCES ET MESURES RELATIFS AU CHAPITRE 5

Cette annexe présente les codes sources relatifs à l'évaluation polynomiale compensée suivant le schéma de HORNER étudiés au chapitre 5. De plus, cette annexe présente en détails les mesures de performances et de précision effectuées pour l'ensemble des algorithmes étudiés à la section 5.3.3.

Le code C.1 implémente l'algorithme compensé COMPHORNER d'après les travaux de S. GRAILLAT, P. LANGLOIS et N. LOUVET [GLL09] (le code ci-dessous est tiré de [Lou07, p. 63]).

Code C.1 : COMPHORNER : compensation de l'algorithme 5.3. ◁

```
Ligne 1  #define DBL_SPLITTER 134217729.0
-
-
-  double
5  CompHorner(double *P, unsigned int n, double x)
-  {
-    double p, r, c, pi, sig, x_hi, x_lo, hi, lo, t;
-    int i;
-
10   /* Split(x_hi, x_lo, x) */
-    t = x * DBL_SPLITTER;
-    x_hi = t - (t - x);
-    x_lo = x - x_hi;
-    r = P[n];
15   c = 0.0;
-
-    for(i=n-1; i>=0; i--) {
-      /* TwoProd(p, pi, s, x); */
-      p = r * x;
20     t = r * DBL_SPLITTER;
-      hi = t - (t - r);
-      lo = r - hi;
-      pi = (((hi*x_hi - p) + hi*x_lo) + lo*x_hi) + lo*x_lo;
-
25     /* TwoSum(s, sigma, p, P[i]); */
-      r = p + P[i];
-      t = r - p;
-      sig = (p - (r - t)) + (P[i] - t);
-      /* Accumulation des termes d'erreurs */
30     c = c * x + (pi+sig);
-    }
-    return(r+c);
-  }
```

Le code C.2 implémente l'algorithme ACHORNER compensé d'après la méthode de compensation automatique présentée au chapitre 5. Ce code obtenu automatiquement est très similaire au code compensé C.1 (optimisé à la main).

Code C.2 : ACHORNER : compensation automatique de l'algorithme 5.3 (le code est légèrement mis en page et commenté pour une meilleure lisibilité). ◁

```
Ligne 1  double
-        ACHorner(double *P, unsigned int n, double x)
-        {
-          int i;
5         double r, delta_r, t, delta_t, delta_plus, delta_times;
-         double c, xh, xl, rh, rl, u;
-
-         r = P[n];
-         delta_r = 0.0;
10
-         for(i = n-1; i>=0; i--)
-           {
-             /* [t, delta_t] = AutoComp_TwoProduct(r, delta_r, x) */
-             /* [t, delta_times] = TwoProduct(r, x) */
15           t = r * x;
-            c = r * 134217729; /* [rh, rl] = Split(r) */
-            rh = c - (c - r);
-            rl = r - rh;
-            c = x * 134217729; /* [xh, xl] = Split(x) */
20           xh = c - (c - x);
-            xl = x - xh;
-            delta_times = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl);
-            /* propagation des erreurs */
-            delta_t = delta_times + delta_r * x;
25
-            /* [r, delta_r] = AutoComp_TwoSum(t, delta_t, P[i]) */
-            /* [r, delta_plus] = TwoSum(t, P[i]) */
-            r = t + P[i];
-            u = r - t;
30           delta_plus = (t - (r - u)) + (P[i] - u);
-            /* propagation des erreurs */
-            delta_r = delta_plus + delta_t;
-          }
-
35        /* fermeture */
-         r = r + delta_r ;
-
-         return r;
-       }
```

Le tableau C.1 présente les mesures de performances étudiés au chapitre 5. Les mesures ont été effectuées dans l'environnement A.1. De plus les performances évaluées avec l'outil PAPI [MBDH99] sont les moyennes de 10^6 exécutions. Les données utilisées pour ces mesures sont décrites dans le tableau 5.5.

	PAPI			PERPI		
	Instructions	Cycles	IPC	Instructions	Cycles	IPC
	HORNER (p_1, x)					
binary64	42	57	0,73	76	37	2,05
COMP	532	277	1,99	566	62	9,12
DD	658	920	0,72	676	325	2,08
AC	553	303	1,82	581	77	7,54
AC/COMP	1,04	1,09	0,95	1,02	1,24	0,82
AC/DD	0,84	0,33	2,55	0,85	0,23	3,62
	HORNERDER (p_1, x)					
binary64	981	416	2,35	1 212	95	12,7
COMP	2 958	1 672	1,77	3 118	107	29,1
DD	4 348	3 464	1,26	4 500	459	9,80
AC	2 960	1 720	1,72	2 976	118	25,2
AC/COMP	1,00	1,03	0,97	0,95	1,10	0,86
AC/DD	0,68	0,50	1,37	0,66	0,25	2,57
	CLENSHAWI (p_2, x)					
binary64	177	179	0,98	259	93	2,78
COMP	1 110	658	1,69	1 241	145	8,55
DD	1 552	1 955	0,79	1 697	672	2,52
AC	1 150	679	1,69	1 282	161	7,96
AC/COMP	1,04	1,03	1,00	1,03	1,11	0,93
AC/DD	0,74	0,35	2,13	0,75	0,23	3,15
	CLENSHAWII (p_2, x)					
binary64	178	175	1,01	257	92	2,79
COMP	1 099	660	1,66	1 243	145	8,57
DD	1 544	1 959	0,79	1 700	672	2,52
AC	1 149	675	1,70	1 333	161	8,27
AC/COMP	1,05	1,02	1,02	1,07	1,11	0,96
AC/DD	0,74	0,34	2,16	0,78	0,23	3,27

TABLE C.1 – *Suite à la page suivante*

Suite de la page précédente

	PAPI			PERPI		
	Instructions	Cycles	IPC	Instructions	Cycles	IPC
	DECASTELJAU (p_3, x)					
binary64	599	261	2,29	1 156	54	21,4
COMP	2 858	1 405	1,97	3 871	71	54,5
DD	3 843	2 607	1,47	4 588	240	19,1
AC	2 713	1 272	2,13	3 416	78	43,7
AC/COMP	0,95	0,88	1,08	0,88	1,09	0,80
AC/DD	0,71	0,49	1,45	0,74	0,32	2,28
	DECASTELJAUDER (p_3, x)					
binary64	515	233	2,21	1 027	80	12,8
COMP	2 390	1 205	1,98	3 298	68	48.5
DD	3 159	2 104	1,50	3 874	219	17,6
AC	2 435	1 195	2,04	2 986	75	39,8
AC/COMP	1,02	0,99	1,03	0,90	1,10	0,82
AC/DD	0,77	0,57	1,36	0,77	0,34	2,25
	SUM (d_7)					
binary64	500 019	302 051	1,66	500 009	100 004	4,99
COMP	1 600 009	711 290	2,25	1 599 999	200 010	7,99
DD	2 100 005	2 406 581	0,87	2 099 995	899 997	2,33
AC	1 600 009	710 006	2,25	1 599 999	200 010	7,99
AC/COMP	1,00	1,00	1,00	1,00	1,00	1,00
AC/DD	0,76	0,30	2,58	0,76	0,22	3,42
	SUM (d_8)					
binary64	5 000 019	3 882 790	1,29	5 000 019	1 000 004	4,99
COMP	16 000 009	7 816 373	2,05	15 999 999	2 000 010	7,99
DD	21 000 007	24 775 068	0,85	20 999 995	8 999 997	2,33
AC	16 000 009	7 819 774	2,05	15 999 999	2 000 010	7,99
AC/COMP	1,00	1,00	1,00	1,00	1,00	1,00
AC/DD	0,76	0,32	2,41	0,76	0,22	3,42

TABLE C.1 – Mesures des performances des algorithmes étudiés au chapitre 5, section 5.3.3.

Le tableau C.2 présente les mesures de précision étudiées au chapitre 5. Les mesures sont exprimées sous la forme de moyenne de bits exacts du résultat de l'évaluation des différents algorithmes étudiés. Le nombre d'évaluations ainsi que les données utilisées, sont décrites dans le tableau 5.5.

Algorithmes et données	*binary64*	COMP	DD	AC	AC-COMP	AC-DD
SUM (d_1)	$26,6^{32}_{20}$	53	53	53	0	0
SUM (d_2)	$25,2^{31}_{20}$	53	53	53	0	0
SUM (d_3)	$23,7^{26}_{19}$	53	53	53	0	0
SUM (d_4)	$0,84^{5}_{0}$	50^{53}_{43}	53	50^{53}_{43}	0	−3
SUM (d_5)	$0,56^{4}_{0}$	$48,2^{53}_{42}$	53	$48,2^{53}_{42}$	0	−4,8
SUM (d_6)	$0,28^{8}_{0}$	43^{49}_{39}	53	43^{49}_{39}	0	−10
HORNER (p_1, x_1)	$0,02^{5}_{0}$	45^{53}_{35}	46^{53}_{35}	45^{53}_{35}	0	−1
HORNER (p_1, x_2)	$0,53^{6}_{0}$	$47,9^{53}_{37}$	$48,4^{53}_{37}$	$47,9^{53}_{37}$	0	−0,5
HORNERDER (p_1, x_1)	$0,98^{12}_{0}$	$51,9^{53}_{47}$	$52,3^{53}_{48}$	52^{53}_{47}	0,1	−0,3
HORNERDER (p_1, x_2)	$7,38^{15}_{0}$	$52,6^{53}_{51}$	$52,8^{53}_{51}$	$52,9^{53}_{51}$	0,3	0,1
CLENSHAWI (p_2, x_1)	0	$40,5^{50}_{32}$	$41,8^{50}_{34}$	$40,5^{50}_{32}$	0	−1,3
CLENSHAWI (p_2, x_2)	$0,3^{7}_{0}$	$46,9^{53}_{36}$	$48,1^{53}_{37}$	$46,6^{53}_{36}$	−0,3	−1,5
CLENSHAWII (p_2, x_1)	0	$41,1^{53}_{32}$	$42,5^{53}_{33}$	$40,8^{50}_{32}$	−0,3	−1,7
CLENSHAWII (p_2, x_2)	$0,32^{8}_{0}$	$46,8^{53}_{36}$	48^{53}_{38}	$46,8^{53}_{36}$	0	−1,2
DECASTELJAU (p_3, x_1)	49^{53}_{43}	53	53	53	0	0
DECASTELJAU (p_3, x_2)	$52,4^{53}_{51}$	53	53	53	0	0
DECASTELJAUDER (p_3, x_1)	$48,8^{53}_{44}$	53	53	53	0	0
DECASTELJAUDER (p_3, x_2)	$52,5^{53}_{51}$	53	53	53	0	0

TABLE C.2 – Mesures de la précision (nombre de bits exacts) des algorithmes étudiés au chapitre 5 (53 bits de précision au maximum en *binary64*). Les colonnes *binary64*, COMP, DD et AC donnent la moyenne du nombre de bits exacts du résultat des différents algorithmes (voir section 5.3.3), avec en exposant le maximum et en indice le minimum mesurés (si différent de la moyenne). Les valeurs exposées dans les colonnes AC-COMP et AC-DD représentent les différences (soustraction) des moyennes correspondantes.

ANNEXE D

MESURES COMPLÉMENTAIRES RELATIVES AU CHAPITRE 8

Cette annexe présente des mesures relatives aux expériences réalisées au chapitre 8.

Sélection	Données	Stratégie gagnante	r_{cycles}	$r_{\text{bits exacts}}$	Succès
		Environnement A.3			
$R^0_{\text{P-P}}$	d_1	*auto-comp*	0,39	1	✓
	d_2		0,4		
	d_3		0,41		
$R^1_{\text{P-P}}$	d_1	$\texttt{ri}(\bot, 1\,000, 5\,000), p_1$	0,17	0,5	✗
	d_2	$\texttt{ri}(\top, 10\,000, 50\,000), p_1$		0,47	
	d_3	$\texttt{ri}(\bot, 100\,000, 400\,000), p_1$	0,2	0,45	
$R^2_{\text{P-P}}$	d_1	$\texttt{ri}(\bot, 1\,000, 5\,000), p_1$	0,17	0,5	✗
	d_2	$\texttt{ri}(\bot, 10\,000, 40\,000), p_2$		0,47	
	d_3	$\texttt{ri}(\bot, 100\,000, 500\,000), p_1$	0,2	0,45	
		Environnement A.4			
$R^0_{\text{P-P}}$	d_1	*auto-comp*	0,45	1	✓
	d_2		0,49		
	d_3				
$R^1_{\text{P-P}}$	d_1	$\texttt{ri}(\bot, 1\,000, 5\,000), p_1$	0,2	0,5	✗
	d_2	$\texttt{ri}(\top, 10\,000, 50\,000), p_2$	0,23	0,47	
	d_3	$\texttt{ri}(\top, 100\,000, 500\,000), p_1$	0,24	0,45	
$R^2_{\text{P-P}}$	d_1	$\texttt{ri}(\bot, 1\,000, 4\,000), p_1$	0,18	0,49	✗
	d_2	$\texttt{ri}(\bot, 10\,000, 40\,000), p_2$	0,25	0,47	
	d_3	$\texttt{ri}(\bot, 100\,000, 500\,000), p_1$	0,24	0,45	

TABLE D.1 – Résultats de la synthèse de code de l'algorithme SUM. Présentation des stratégies gagnantes, des ratios r_{cycles} et $r_{\text{bits exacts}}$ pour chaque jeux de données du tableau 8.2 et sélections performances–précision dans les environnements A.3 et A.4.

Dans le cas de la somme, la synthèse de code prend respectivement 24 secondes, 1 minute 7 secondes et 8 minutes 27 secondes pour traiter les données d_1, d_2 et d_3 dans l'environnement A.3. Dans l'environnement A.4, il faut respectivement 28 secondes, 2 minutes 9 secondes et 16 minutes 56 secondes pour traiter les données d_1, d_2 et d_3.

Sélection	Données	Stratégie gagnante	r_{cycles}	$r_{\text{bits exacts}}$	Succès
		Environnement A.3			
$R^0_{\text{P-P}}$	d_4	*auto-comp*	0,33	1	✓
	d_5				
	d_6				
$R^1_{\text{P-P}}$	d_4	$\text{RS}(\bot, 9/10), p_1$	0,27	0,86	✓
	d_5	$\text{RS}(\top, 9/10), p_2$	0,30	0,43	
	d_6	$\text{RS}(\top, 9/10), p_1$		0,98	
$R^2_{\text{P-P}}$	d_4	$\text{RS}(\bot, 9/10), p_1$	0,28	0,86	✓
	d_5	$\text{RI}(\top, 2, 3), p_1$	0,39	0,22	✗
	d_6	$\text{RS}(\bot, 9/10), p_1$	0,29	0,98	✓
		Environnement A.4			
$R^0_{\text{P-P}}$	d_4	*auto-comp*	0,43	1	✓
	d_5		0,41		
	d_6		0,43		
$R^1_{\text{P-P}}$	d_4	$\text{RS}(\bot, 9/10), p_1$	0,36	0,86	✓
	d_5	$\text{RI}(\bot, 1, 5), p_2$	0,16	0,22	✗
	d_6	$\text{RS}(\top, 9/10), p_1$	0,39	0,98	✓
$R^2_{\text{P-P}}$	d_4	$\text{RS}(\bot, 9/10), p_1$	0,36	0,86	✓
	d_5	$\text{RI}(\top, 1, 2), p_1$	0,4	0,22	✗
	d_6	$\text{RS}(\top, 9/10), p_1$	0,41	0,98	✓

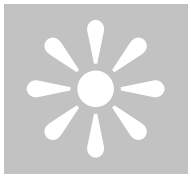
TABLE D.2 – Résultats de la synthèse de code de l'algorithme HORNER. Présentation des stratégies gagnantes, des ratios r_{cycles} et $r_{\text{bits exacts}}$ pour chaque jeux de données du tableau 8.4 et sélections performances–précision dans les environnements A.3 et A.4.

Dans le cas de l'évaluation polynomiale par l'algorithme de HORNER, la synthèse de code prend en moyenne 1 minute et 4 secondes pour traiter les données d_4, d_5 ou d_6 dans l'environnement A.3. Dans l'environnement A.4, il faut en moyenne 1 minute 12 secondes pour traiter les données d_4, d_5 ou d_6.

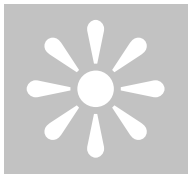
Sélection	Données	Stratégie gagnante	r_{cycles}	$r_{\text{bits exacts}}$	Succès
		Environnement A.3			
$R^0_{\text{P-P}}$	d_4	*auto-comp*	0,28	1	✓
	d_6		0,27		
	d_7				
$R^1_{\text{P-P}}$	d_4	$\texttt{rs}(\top, 9/10), p_2$	0,28	1	✓
	d_6	$\texttt{rs}(\top, 9/10), p_1$			
	d_7	$\texttt{rs}(\top, 9/10), p_2$			
$R^2_{\text{P-P}}$	d_4	$\texttt{rs}(\top, 9/10), p_1$	0,28	1	✓
	d_6				
	d_7				
		Environnement A.4			
$R^0_{\text{P-P}}$	d_4	*auto-comp*	0,36	1	✓
	d_6				
	d_7		0,34		
$R^1_{\text{P-P}}$	d_4	$\texttt{rs}(\top, 9/10), p_1$	0,34	1	✓
	d_6				
	d_7				
$R^2_{\text{P-P}}$	d_4	$\texttt{rs}(\top, 9/10), p_2$	0,34	1	✓
	d_6				
	d_7	$\texttt{rs}(\top, 9/10), p_1$	0,35		

TABLE D.3 – Résultats de la synthèse de code de l'algorithme CLENSHAWI. Présentation des stratégies gagnantes, des ratios r_{cycles} et $r_{\text{bits exacts}}$ pour chaque jeux de données du tableau 8.6 et sélections performances–précision dans les environnements A.3 et A.4.

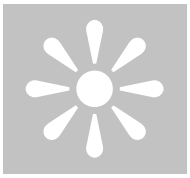
Dans le cas de l'évaluation polynomiale par l'algorithme de CLENSHAWI, la synthèse de code prend en moyenne 1 minute et 15 secondes pour traiter les données d_4, d_6 ou d_7 dans l'environnement A.3. Dans l'environnement A.4, il faut en moyenne 1 minute 30 secondes pour traiter les données d_4, d_6 ou d_7.



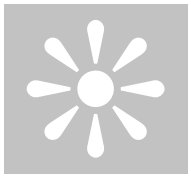
LISTE DES FIGURES



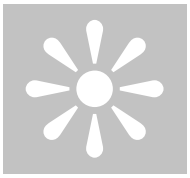
LISTE DES TABLEAUX



LISTE DES ALGORITHMES



LISTE DES CODES



BIBLIOGRAPHIE

[10994] ISO-IEC 10967 : *Information Technology, Language Independent Arithmetic, Part 1 : Integer and Floating Point Arithmetic*, 1994. *Cité page 11.*

[75408] IEEE 754 : IEEE Standard for Floating-Point Arithmetic. Rapport technique, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, août 2008. *Cité pages 6, 11, et 12.*

[ALSU06] Alfred V. AHO, Monica S. LAM, Ravi SETHI et Jeffrey D. ULLMAN : *Compilers : Principles, Techniques, and Tools*. Addison Wesley, seconde édition, août 2006. *Cité pages 1, 6, 38, et 39.*

[App98] Andrew W. APPEL : *Modern Compiler Implementation : In ML*. Cambridge University Press, New York, NY, USA, 1998. *Cité page 38.*

[Bai91] David H. BAILEY : Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. *Supercomputing Review*, pages 54–55, août 1991. *Cité pages 1 et 6.*

[BBDN10] Matthew BADIN, Lubomir BIC, Michael DILLENCOURT et Alexandru NICOLAU : Improving Accuracy through Selective Doubly Compensated Summation, 2010. Workshop on Language, Compiler, and Architecture Support for GPGPU. *Cité pages 2, 51, et 85.*

[BBL+02] Christian H. BISCHOF, H. Martin BÜCKER, Bruno LANG, Arno RASCH et Andre VEHRESCHILD : Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs. *In SCAM*, pages 65–72. IEEE Computer Society, octobre 2002. *Cité page 4.*

[BDD+02] Susan L. BLACKFORD, James DEMMEL, Jack DONGARRA, Iain DUFF, Sven HAMMARLING, Greg HENRY, Michael HEROUX, Linda KAUFMAN, Andrew LUMSDAINE, Antoine PETITET, Roldan POZO, Karin REMINGTON et Clint R. WHALEY : An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, juin 2002. *Cité page 136.*

[BGS94] David F. BACON, Susan L. GRAHAM et Oliver J. SHARP : Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, décembre 1994. *Cité pages 1 et 39.*

[Bir04] Tom R. BIRD : Methods to Improve Bootup Time on Linux. *In Ottawa Linux Symposium*, volume 1, pages 79–88, 2004. *Cité page 1.*

[BL02] Thierry BRACONNIER et Phillippe LANGLOIS : From Rouding Error Estimation to Automatic Correction with Automatic Differentiation. *In* George CORLISS, Christèle FAURE, Andreas GRIEWANK, Lauren HASCOËT et Uwe NAUMANN, éditeurs : *Automatic Differentiation of Algorithms*, pages 351–357. Springer-Verlag New York, Inc., New York, NY, USA, 2002. *Cité pages 4, 22, et 65.*

[BLW11] David BAILEY, Robert LUCAS et Samuel WILLIAMS : *Performance Tuning of Scientific Applications*. CRC Press, Boca Raton, novembre 2011. *Cité page 42.*

[BMPW66] H.J. BOWDLER, R.S. MARTIN, G. PETERS et J.H. WILKINSON : Solution of Real and Complex Systems of Linear Equations. *Numerische Mathematik*, 8(3):217–234, 1966. *Cité pages 136, 137, et 176.*

[BOB92] Michael BLAIR, Sally OBENSKI et Paula BRIDICKAS : GAO/IMTEC-92-26 Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia. Rapport technique, GAO/IMTEC, février 1992. *Cité pages 2 et 15.*

[Cas00] Hughes CASSE : frontc 3.4 : an OCAML C Parser and Pretty-Printer. *TRACES Research Group, Institut de recherche en informatique de Toulouse*, 2000. *Cité pages 5 et 115.*

[Dau99] Marc DAUMAS : Multiplications of Floating Point Expansions. *In Proceedings of the 14th Symposium on Computer Arithmetic*, pages 250–257, 1999. *Cité page 28.*

[Dek71] Theodorus J. DEKKER : A Floating-Point Technique for Extending the Available Precision. *Numerische Mathematik*, 18:224–242, 1971. *Cité pages 6, 23, 26, et 29.*

[Dem92] James W. DEMMEL : Trading Off Parallelism and Numerical Stability. Rapport technique CRPC–TR92422, Center for Researsh on Parallel Computation, Rice University, 1992. *Cité pages 51 et 85.*

[DGP+09] David DELMAS, Eric GOUBAULT, Sylvie PUTOT, Jean SOUYRIS, Karim TEKKAL et Franck VÉDRINE : Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. *In Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag. *Cité pages 2 et 22.*

[DHK+06] James W. DEMMEL, Yozo HIDA, William KAHAN, Xiaoye S. LI, Soni MUKHERJEE et E. Jason RIEDY : Error Bounds From Extra-Precise Iterative Refinement. *ACM Transactions on Mathematical Software*, 32(2):325–351, juin 2006. *Cité pages 6, 137, et 145.*

[DRV00] Alain DARTE, Yves ROBERT et Frederic VIVIEN : *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000. *Cité pages 1, 39, et 90.*

[FHL+07] Laurent FOUSSE, Guillaume HANROT, Vincent LEFÈVRE, Patrick PÉLISSIER et Paul ZIMMERMANN : MPFR : A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw*, 33:818, 2007. *Cité pages 22 et 149.*

[FM01] Claire FINOT-MOREAU : *Preuve et algorithmes utilisant l'arithmétique flottante normalisée IEEE*. Thèse de doctorat, Lyon, ENSL, Grenoble, 2001. *Cité pages 28 et 85.*

[Fou14] Free S. FOUNDATION : GNU GCC Manual, 2014. *Cité page 85.*

[GJ08] Dick GRUNE et Ceriel J.H. JACOBS : *Parsing Techniques : A Practical Guide*. Springer-Verlag New York Inc, seconde édition, 2008. *Cité page 39.*

[GLL09] Stef GRAILLAT, Philippe LANGLOIS et Nicolas LOUVET : Algorithms for Accurate, Validated and Fast Polynomial Evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3):191–214, 2009. *Cité pages 2, 6, 28, 71, 74, 76, 85, 121, 144, 145, et 158.*

[GLPP12] Bernard GOOSSENS, Philippe LANGLOIS, David PARELLO et Eric PETIT : PerPI : A Tool to Measure Instruction Level Parallelism. *In* Kristján JÓNASSON, éditeur : *Applied Parallel and Scientific Computing*, volume 7133 de *Lecture Notes in Computer Science*, pages 270–281. Springer Berlin Heidelberg, 2012. *Cité pages 46 et 149.*

[Gol89] David E. GOLDBERG : *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. *Cité pages 41 et 149.*

[Gol91] David E. GOLDBERG : What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23:5–48, 1991. *Cité pages 2 et 12.*

[Gul10] Sumit GULWANI : Dimensions in Program Synthesis. *In Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 13–24, New York, NY, USA, 2010. ACM. *Cité pages 3, 6, 40, 109, et 110.*

[Hig] Nicholas J. HIGHAM : The Matrix Computation Toolbox. `http://www.ma.man.ac.uk/~higham/mctoolbox`. *Cité page 138.*

[Hig93] Nicholas J. HIGHAM : The Accuracy Of Floating Point Summation. *SIAM J. Sci. Comput*, 14:783–799, 1993. *Cité pages 2 et 22.*

[Hig02] Nicholas J. HIGHAM : *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, seconde édition, 2002. *Cité pages 2, 12, 19, 55, 136, et 137.*

[Hil87] Begnaud F. HILDEBRAND : *Introduction to Numerical Analysis*. Dover Publications, Inc., New York, NY, USA, seconde édition, 1987. *Cité page 20.*

[HLB01] Yozo HIDA, Xiaoye S. LI et David H. BAILEY : Algorithms for Quad-Double Precision Floating Point Arithmetic. *In Proceedings of the 15th Symposium on Computer Arithmetic*, pages 155–162. IEEE Computer Society Press, 2001. *Cité pages 2, 6, 29, et 147.*

[Hop69] Frank R. A. HOPGOOD : *Compiling Techniques*. MacDonald Computer Monographs. MacDonald New York, London, 1969. *Cité pages 1 et 38.*

[HP12a] Laurent HASCOËT et Valérie PASCUAL : The Tapenade Automatic Differentiation Tool : Principles, Model, and Specification. Rapport de recherche RR-7957, INRIA, mai 2012. *Cité page 4.*

[HP12b] John L. HENNESSY et David A. PATTERSON : *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, cinquième édition, 2012. *Cité pages 1 et 42.*

[IM12] Arnault IOUALALEN et Matthieu MARTEL : Sardana : an Automatic Tool for Numerical Accuracy Optimization. *In 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics, SCAN*, pages 64–65, septembre 2012. *Cité pages 2, 7, 22, 57, et 149.*

[Iou13] Arnault IOUALALEN : *Transformation de programmes synchrones pour l'optimisation de la précision numérique*. Thèse de doctorat, Université de Perpignan Via Domitia, Perpignan, décembre 2013. *Cité page 7.*

[JBL+11] Hao JIANG, Roberto BARRIO, Housen LI, Xiangke LIAO, Lizhi CHENG et Fang SU : Accurate Evaluation of a Polynomial in Chebyshev Form. *Applied Mathematics and Computation*, 217(23):9702 – 9716, 2011. *Cité pages 6, 28, 71, 74, 79, 121, 144, et 145.*

[JC08] Fabienne JÉZÉQUEL et Jean-Marie CHESNEAUX : CADNA : A Library for Estimating Round-Off Error Propagation. *Computer Physics Communications*, 178(12):933 – 955, 2008. *Cité page 22.*

[JGH+13] Hao JIANG, Stef GRAILLAT, Canbin HU, Shengguo LI, Xiangke LIAO, Lizhi CHANG et Fang SU : Accurate Evaluation of the k-th Derivative of a Polynomial and its Application. *Journal of Computational and Applied Mathematics*, 243(0):28–47, 2013. *Cité pages 6, 28, 71, et 145.*

[JJLMR10] Claude-Pierre JEANNEROD, Jingyan JOURDAN-LU, Christophe MONAT et Guillaume REVY : How to Square Floats Accurately and Efficiently on the ST231 Integer Processor. 2010. *Cité page 148.*

[JLCS10] Hao JIANG, Shengguo LI, Lizhi CHENG et Fang SU : Accurate Evaluation of a Polynomial and its Derivative in Bernstein Form. *Journal of Computational and Applied Mathematics*, 60(3):744–755, août 2010. *Cité pages 6, 28, 71, et 145.*

[KLLM12] Peter KORNERUP, Vincent LEFÈVRE, Nicolas LOUVET et Jean-Michel MULLER : On the Computation of Correctly Rounded Sums. *Computers, IEEE Transactions on*, 61(3):289–298, 2012. *Cité page 24.*

[Knu97] Donald E. KNUTH : *The Art of Computer Programming : Seminumerical Algorithms*, volume 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, troisième édition, 1997. *Cité pages 6 et 24.*

[KR88] Brian W. KERNIGHAN et Dennis M. RITCHIE : *The C Programming Language*. Prentice Hall Professional Technical Reference, seconde édition, 1988. *Cité pages 110 et 115.*

[Lan00] Philippe LANGLOIS : A Revised Presentation of the CENA Method. Rapport technique RR-4025, ARENAIRE, Inria Grenoble Rhône-Alpes, LIP Laboratoire de l'Informatique du Parallélisme, octobre 2000. *Cité page 65.*

[Lau05] Christoph Quirin LAUTER : Basic Building Blocks for a Triple-Double Intermediate Format. Rapport technique RR-5702, INRIA, septembre 2005. *Cité pages 6 et 29.*

[Lay05] David C. LAY : *Linear Algebra and Its Applications with CD-ROM, Update : United States Edition*. Pearson, troisième édition, août 2005. *Cité page 15.*

[LCM+05] Chi-Keung LUK, Robert COHN, Robert MUTH, Harish PATIL, Artur KLAUSER, Geoff LOWNEY, Steven WALLACE, Vijay Janapa REDDI et Kim HAZELWOOD : Pin : Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, juin 2005. *Cité page 46.*

[LDB+02] Xiaoye S. LI, James W. DEMMEL, David H. BAILEY, Greg HENRY, Yozo HIDA, Jimmy ISKANDAR, William KAHAN, Suh Y. KANG, Anil KAPUR, Michael C. MARTIN, Brandon J. THOMPSON, Teresa TUNG et Daniel J. YOO : Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, juin 2002. *Cité page 137.*

[LHHL96] Jacques-Louis LIONS, Remy HERGOTT, Bernard HUMBERT et Eric LEFORT : Ariane 5 Flight 501 Failure, Report by the Inquiry Board. Rapport technique, European Space Agency, 1996. *Cité pages 2 et 15.*

[LL07] Philippe LANGLOIS et Nicolas LOUVET : More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms. Laboratoire de Physique Appliquée et d'Automatique - LP2A, 2007. *Cité pages 32 et 34.*

[LMT10a] Philippe LANGLOIS, Matthieu MARTEL et Laurent THÉVENOUX : Trade-off Between Accuracy and Time for Automatically Generated Summation Algorithms. *In SCAN 2010 – 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, pages 85–86, juillet 2010. *Cité pages 3, 7, 8, et 51.*

[LMT10b] Philippe LANGLOIS, Matthieu MARTEL et Laurent THÉVENOUX : Accuracy Versus Time : A Case Study with Summation Algorithms. *In PASCO '10 : Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 121–130, New York, NY, USA, 2010. ACM. *Cité pages 3, 7, 8, 22, 51, 54, et 85.*

[LMT12] Philippe LANGLOIS, Matthieu MARTEL et Laurent THÉVENOUX : Automatic Code Transformation to Optimize Accuracy and Speed in Floating-Point Arithmetic. *In SCAN 2012 – 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, septembre 2012. *Cité pages 3, 7, 8, et 64.*

[Lou07] Nicolas LOUVET : *Algorithmes compensés en arithmétique flottante : Précision, validation, performances.* Thèse de doctorat, Université de Perpignan Via Domitia, Perpignan, novembre 2007. *Cité pages 32, 76, 79, et 158.*

[LPGP12] Philippe LANGLOIS, David PARELLO, Bernard GOOSSENS et Kathy PORADA : Less Hazardous and More Scientific Research for Summation Algorithm Computing Times. Rapport technique, Université de Perpignan Via Domitia -

UPVD , Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier - LIRMM, septembre 2012. *Cité pages 1, 6, 42, et 74.*

[LSCK09] Antonio Roldao LOPES, Amir SHAHZAD, George A. CONSTANTINIDES et Eric C. KERRIGAN : More Flops or More Precision ? Accuracy Parameterizable Linear Equation Solvers for Model Predictive Control. *In* Kenneth L. POCEK et Duncan A. BUELL, éditeurs : *FCCM*, pages 209–216. IEEE Computer Society, 2009. *Cité page 85.*

[Mar90] Peter W. MARKSTEIN : Computation of Elementary Functions on the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34:111–119, janvier 1990. *Cité pages 25 et 146.*

[Mar09] Matthieu MARTEL : Program Transformation for Numerical Precision. *In Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 101–110, New York, NY, USA, 2009. ACM. *Cité page 22.*

[MBdD+10] Jean-Michel MULLER, Nicolas BRISEBARRE, Florent de DINECHIN, Claude-Pierre JEANNEROD, Vincent LEFÈVRE, Guillaume MELQUIOND, Nathalie REVOL, Damien STEHLÉ et Serge TORRES : *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. *Cité pages 2, 6, 12, et 65.*

[MBDH99] Philip J. MUCCI, Shirley BROWNE, Christine DEANE et George HO : PAPI : A Portable Interface to Hardware Performance Counters. *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999. *Cité pages 6, 44, 113, 117, et 160.*

[McN04] John Michael MCNAMEE : A Comparison of Methods for Accurate Summation. *SIGSAM Bull.*, 38(1):1–7, 2004. *Cité page 22.*

[Møl65] Ole MØLLER : Quasi Double-Precision in Floating-Point Addition. *BIT Numerical Mathematics*, 5:37–50, 1965. *Cité page 24.*

[MR11] Christophe MOUILLERON et Guillaume REVY : Automatic Generation of Fast and Certified Code for Polynomial Evaluation. *In* E. ANTELO, D. HOUGH et P. IENNE, éditeurs : *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, pages 233–242, Tuebingen, Germany, juillet 2011. IEEE Computer Society. *Cité pages 2, 41, et 63.*

[Mul05] Jean-Michel MULLER : On the Definition of ulp(x). Rapport technique RR-5504, Arenaire, Inria Rhône-Alpes, Laboratoire de l'Informatique du Parallélisme, CNRS : UMR5668, Université Claude Bernard, Lyon I, École normale supérieure de Lyon, février 2005. *Cité page 18.*

[Nat98] Fabrice NATIVEL : *Fiabilité numérique et précision finie : une méthode automatique de correction linéaire de l'erreur d'arrondi.* Thèse de doctorat, 1998. Thèse de doctorat dirigée par P. LANGLOIS. Mathématiques appliquées et calcul scientifique, La Réunion. *Cité page 65.*

[Neu01] Arnold NEUMAIER : *Introduction to Numerical Analysis.* Cambridge University Press, 2001. *Cité page 136.*

[Ove01] Michael L. OVERTON : *Numerical Computing with IEEE Floating-Point Arithmetic - Including One Theorem, One Rule Of Thumb, and One Hundred and One Exercices.* SIAM, 2001. *Cité pages 12 et 17.*

[PV93] Michèle PICHAT et Jean VIGNES : *Ingénierie du contrôle de la précision des calculs sur ordinateur.* Technip, France, 1993. *Cité pages 6, 25, 70, et 146.*

[RG81] B. Ramakrishna RAU et Christopher D. GLAESER : Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *SIGMICRO Newsl.*, 12(4):183–198, décembre 1981. *Cité page 39.*

[RGNN+13] Cindy RUBIO-GONZÁLEZ, Cuong NGUYEN, Diep NGUYEN, James DEMMEL, William KAHAN, Koushik SEN, David H. BAILEY, Costin IANCU et David HOUGH : Precimonious : Tuning Assistant for Floating-Point Precision. *In Proceedings of the SC13's International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, USA, novembre 2013. *Cité page 64.*

[ROO05] Siegfried M. RUMP, Takeshi OGITA et Shin'ichi OISHI : Accurate Sum and Dot Product. *SIAM J. Sci. Comput*, 26:2005, 2005. *Cité pages 6, 27, 71, 72, 85, 88, 121, 144, 145, et 147.*

[ROO08] Siegfried M. RUMP, Takeshi OGITA et Shin'ichi OISHI : Accurate Floating-Point Summation Part I : Faithful Rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, octobre 2008. *Cité pages 18 et 22.*

[RS88] Thomas G. ROBERTAZZI et Stuart C. SCHWARTZ : Best Ordering for Floating-Point Addition. *ACM Transactions on Mathematical Software*, 14:101–110, mars 1988. *Cité page 22.*

[Rum09] Siegfried M. RUMP : Ultimately Fast Accurate Summation. *SIAM J. Sci. Comput.*, 31(5):3466–3502, septembre 2009. *Cité pages 2 et 42.*

[Sar98] Vivek SARKAR : Loop Transformations for Hierarchical Parallelism and Locality. *In* DavidR. O'HALLARON, éditeur : *Languages, Compilers, and Run-Time*

Systems for Scalable Computers, volume 1511 de *Lecture Notes in Computer Science*, pages 57–74. Springer Berlin Heidelberg, 1998. *Cité page 39.*

[SGF10] Saurabh SRIVASTAVA, Sumit GULWANI et Jeffrey S. FOSTER : From Program Verification to Program Synthesis. *SIGPLAN Not.*, 45(1):313–326, janvier 2010. *Cité pages 3, 40, et 109.*

[She97] Jonathan Richard SHEWCHUK : Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, octobre 1997. *Cité pages 6, 28, et 85.*

[SIH10] Tsubasa SAITO, Emiko ISHIWATA et Hidehiko HASEGAWA : Development of Quadruple Precision Arithmetic Toolbox QuPAT on Scilab. *In Computational Science and Its Applications – ICCSA 2010*, volume 6017 de *Lecture Notes in Computer Science*, pages 60–70. Springer Berlin Heidelberg, 2010. *Cité page 29.*

[The04] THE COQ DEVELOPMENT TEAM : *The Coq Proof Assistant Reference Manual*. LogiCal Project, 2004. Version 8.0. *Cité page 2.*

[Vel68] G. W. VELTKAMP : ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Rapport technique 22, RC-Informatie, Technishe Hogeschool Eindhoven, 1968. *Cité page 25.*

[Vel69] G. W. VELTKAMP : ALGOL procedures voor het rekenen in dubbele lengte. Rapport technique 21, RC-Informatie, Technishe Hogeschool Eindhoven, 1969. *Cité page 25.*

[WD10] Vincent M WEAVER et Jack DONGARRA : Can Hardware Performance Counters Produce Expected, Deterministic Results ? *3rd Workshop on Functionality of Hardware Performance Monitoring (FHPM)*, décembre 2010. *Cité pages 1, 6, et 44.*

[Wil63] James H. WILKINSON : *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1963. *Cité page 20.*

[WTM13] Vince WEAVER, Dan TERPSTRA et Shirley MOORE : Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2013. *Cité page 44.*

[ZH10] Yong-Kang ZHU et Wayne B. HAYES : Algorithm 908 : Online Exact Summation of Floating-Point Streams. *ACM Trans. Math. Softw.*, 37(3):37 :1–37 :13, septembre 2010. *Cité page 22.*

[ZL10] Thomáš ZAHRADNICKÝ et Róbert LÓRENCZ : FPU Supported Running Error Analysis. *Acta Polytechnica*, 50(2):30–36, 2010. *Cité page 19.*

Abstract

Numerical accuracy and execution time of programs using the floating-point arithmetic are major challenges in many computer science applications. The improvement of these criteria is the subject of many research works. However we notice that the accuracy improvement decrease the performances and conversely. Indeed, improvement techniques of numerical accuracy, such as expansions or compensations, increase the number of computations that a program will have to execute. The more the number of computations added is, the more the performances decrease. This thesis work presents a method of accuracy improvement which take into account the negative effect on the performances. So we automatize the error-free transformations of elementary floating-point operations because they present a high potential of parallelism. Moreover we propose some transformation strategies allowing partial improvement of programs to control more precisely the impact on execution time. Then, tradeoffs between accuracy and performances are assured by code synthesis. We present also several experimental results with the help of tools implementing all the contributions of our works.

Keywords: *Floating-point arithmetic, program transformations, optimization, numerical accuracy, error-free transformations, performances.*

Résumé

La précision numérique et le temps d'exécution des programmes utilisant l'arithmétique flottante sont des enjeux majeurs dans de nombreuses applications de l'informatique. L'amélioration de ces critères fait l'objet de nombreux travaux de recherche. Cependant, nous constatons que l'amélioration de la précision diminue les performances et inversement. En effet, les techniques d'amélioration de la précision, telles que les expansions ou les compensations, augmentent le nombre de calculs que devra exécuter un programme. Plus ce coût est élevé, plus les performances sont diminuées. Ce travail de thèse présente une méthode d'amélioration automatique de la précision prenant en compte l'effet négatif sur les performances. Pour cela nous automatisons les transformations sans erreur des opérations élémentaires car cette technique présente un fort potentiel de parallélisme. Nous proposons de plus des stratégies d'optimisation permettant une amélioration partielle des programmes afin de contrôler plus finement son impact sur les performances. Des compromis entre performances et précision sont alors assurés par la synthèse de code. Nous présentons de plus, à l'aide d'outils implantant toutes les contributions de ce travail, de nombreux résultats expérimentaux.

Mots clefs : *Arithmétique flottante, transformation de programmes, optimisation, précision numérique, transformations sans erreur, performances.*

UPVD — 52 avenue Paul Alduy — 66860 Perpignan Cedex 9 — France