Outils de débogage et d'optimisation des calculs flottants dans le contexte HPC

Yohan Chatelain

université PARIS-SACLAY

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

NNT : 2019SACLV096

Thèse de doctorat

Outils de débogage et d'optimisation des calculs flottants dans un contexte HPC

Thèse de doctorat de l'Université Paris-Saclay
préparée à Université de Versailles-Saint-Quentin-en-Yvelines

École doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Versailles, le 12 Décembre 2019, par

Yohan Chatelain

Composition du Jury :

Florent DE DINECHIN Professeur, INSA Lyon	Rapporteur
Stef GRAILLAT Professeur, Sorbonne Université	Rapporteur
Sylvie BOLDO Directrice de Recherche, INRIA	Présidente
Marius CORNEA Ingénieur de Recherche, Intel Corporation	Examinateur
William JALBY Professeur, Université de Versailles-Saint-Quentin-en-Yvelines	Directeur de thèse
Pablo DE OLIVEIRA CASTRO HERRERO Maître de conférences, Université de Versailles-Saint-Quentin-en-Yvelines	Co-encadrant de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Mots clés : HPC, virgule flottante, débugage, optimisation

Résumé : Le Calcul Haute Performance (HPC) est un écosystème dynamique où architectures et codes de calcul scientifiques sont en co-évolution permanente (parallélisme, accélérateurs spécialisés, nouvelles mémoires). Ce dynamisme impose aux développeurs d'adapter leur logiciel régulièrement pour exploiter au mieux tous les nouveaux ressorts technologiques. En cela, les approches de co-design consistant à développer simultanément le logiciel et le matériel sont une voie intéressante. Néanmoins, les efforts de co-design dans le HPC ont surtout été concentrés sur la performance des applications en laissant un peu de côté l'objectif de qualité numérique. Or celle ci est de plus en plus difficile à maintenir d'une génération de supercalculateur à l'autre en raison de la complexité croissante des architectures et des modèles de programmation parallèles. A cela s'ajoute de nouveaux formats de calcul flottants (bfloat16, binary16) qu'il faut réussir à exploiter lors du processus de modernisation.

Ces constats posent deux problématiques : 1) Comment vérifier la qualité numérique des codes lors du processus de modernisation ? Cela nécessite des outils qui permettent, à la fois d'identifier rapidement des sources d'erreurs numériques mais qui doivent également être simple d'utilisation pour des utilisateurs non-experts. 2) Comment tirer parti des nouvelles possibilités qu'offre le matériel ?

Les possibilités d'applications sont nombreuses et amènent donc à un espace considérable de solutions possibles. Les solutions trouvées sont le résultat d'un compromis entre performance de l'application et qualité numérique des calculs mais également reproductibilité des résultats.

Dans cette thèse, nous avons contribué au logiciel Verificarlo qui aide à la détection d'erreurs numériques en injectant divers modèles de bruit dans les calculs flottants. Plus précisément, nous avons développé une approche permettant d'étudier l'évolution des erreurs numérique au cours du temps. Cet outil est basé sur la génération de traces numériques qui permettent de suivre la qualité numérique des variables au cours du temps. Ces traces sont enrichies par des informations de contexte récupérées lors de la compilation puis peuvent être ensuite visualisées de manière élégante.

Nous avons également contribué à VPREC, un modèle de calcul simulant des formats de taille variable. Cet outil a été utilisé pour répondre au problème d'optimisation de formats dans les schémas itératifs. L'optimisation proposée est temporelle puisqu'elle optimise la précision de calcul pour chaque pas de temps. Enfin, une contrainte majeure dans l'élaboration d'outils pour le HPC est la mise à l'échelle. En effet, la taille des codes et la quantité de calcul mis en jeux accroissent drastiquement la complexité des analyses et limitent les approches conventionnelles. Nous avons démontré que les techniques développés dans cette thèse sont applicables sur des codes industriels puisqu'ils ont permis de, premièrement, détecter et corriger une erreur numérique dans le code ABINIT (code ab initio de chimie quantique développé par le CEA et al.). Secondement, ces outils ont permis de réduire la précision de calcul de YALES2 (code de mécanique des fluides développé par le CORIA) et améliorer les performance en réduisant le volumes des communications de 28% et accélérer jusqu'à $1,30$ fois l'exécution.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Keywords : HPC, floating point, debugging, optimization

Abstract : High Performance Computing (HPC) is a dynamic ecosystem where scientific computing architectures and codes are in permanent co-evolution (parallelism, specialized accelerators, new memories). This dynamism requires developers to adapt their software regularly to exploit all the new technological innovations. For this purpose, co-design approaches consisting of simultaneously developing software and hardware are an interesting approach. Nevertheless, co-design efforts have mainly focused on application performance at the cost of numerical quality. However, this is becoming increasingly difficult to maintain from one generation of supercomputer to the next. In addition, there are new computation formats (bfloat16, binary16) performance factor that have to be used during the modernization process.

These findings raise two issues: 1) How to check the digital quality of codes during the modernization process? This requires tools that allow both to quickly identify sources of numerical errors and to be user-friendly for non-expert users. 2) How can we take advantage of the new possibilities offered by the equipment? The applications possibilities are manifold and therefore lead to a considerable space of possible solutions. The solutions found are the result of a compromise between the performance of the application and the numerical quality of the computations, but also the reproducibility of the results.

In this thesis, we contributed to the Verificarlo software that helps to detect numerical errors by injecting various noise models into floating computations. More precisely, we have developed an approach to study the evolution of numerical errors over time. This tool is based on the generation of numerical traces that allow the numerical quality of the variables to be tracked over time. These traces are enriched by context information retrieved during compilation and can then be viewed in an elegant way.

We also contributed to VPREC, a calculation model simulating formats of varying sizes. This tool has been used to address the problem of format optimization in iterative schemes. The proposed optimization is temporal since it optimizes the computation precision for each time step.

Finally, a major constraint in the development of tools for HPC is the scaling up. Indeed, the size of the codes and the number of computations involved drastically increase the complexity of the analyses and limit conventional approaches. We have demonstrated that the techniques developed in this thesis are applicable to industrial codes since they have made it possible, first, to detect and correct a numerical error in the ABINIT code (ab initio code for quantum chemistry developed by the CEA et al.). Secondly, these tools have reduced the computation accuracy of YALES2 (fluid mechanics code developed by CORIA) and improved performance by reducing communication volumes by 28% and accelerating execution up to 1.30 times.

Remerciements

Ce travail de thèse n'aurait pu être accompli sans les personnes suivantes :

Tout d'abord, mon encadrant Pablo de Oliveira Castro qui a su faire preuve de patience et d'abnégation pour m'accompagner jusqu'ici. Ses remarques pointues, ses conseils avisés, son oreille attentive, sa confiance en moi m'ont permis d'arriver à ce manuscrit, fruit de trois années de travail soutenu. Au-delà du travail, sa bonne humeur quotidienne, sa vivacité intellectuelle et sa vision du monde ont rendu ces trois ans un véritable plaisir. Merci.

Je tiens à remercier mon directeur de thèse William Jalby pour m'avoir accueillis dans son laboratoire et de s'être assurer que cette thèse se déroule dans les meilleurs conditions qu'ils soient.

Je souhaiterais remercier Eric Petit, mon encadrant officieux, qui m'a également accompagner durant toute cette thèse. Sa vision à long terme, ses connaissances étendues et tout simplement sa vision du monde ont rendu le travail avec lui agréable et enrichissant.

Je remercie David Defour qui m'a également accompagner durant cette thèse en apportant son expertise et son expérience sur chacun des projets menés au cours de cette thèse mais aussi les discussions moins formelles toujours agréables et enrichissantes.

Je remercie les membres du jury d'avoir assister à ma soutenance. Toute ma reconnaissance à mes rapporteurs Messieurs Florent De Dinechin et Stef Graillat d'avoir accepté de relire ce manuscrit compte tenu du court délai qui leur étaient imparti. Ma reconnaissance également à Sylvie Boldo et Marius Cornea d'avoir accepté d'être examinatrice et examinateur. Je redouble ma reconnaissance à Marius Cornea pour m'avoir accueilli dans son équipe d'Intel à Portland durant 6 mois, riche d'enseignements et de rencontres.

Une thèse c'est de la recherche mais également des rencontres. Ma liste de rencontre est longue et pourrait être agrémentée d'anecdote pour chacune d'entre elle. J'ai opté pour la forme concise : Bruno, Cédric, Clément, Damien, François, Emmanuel, Hugo, John, Kévin, Loïc(s), Matei, Mathieu, Marie, Mihail, Nathalie, Ricardo, Romain, Salah, Sébastien, Soraya, Stéphane, Thomas(s), Youenn et l'équipe des MIHPS.

Derrière un thésard, il y a une famille, qui ne comprend pas toujours ce qu'il fait mais qui est toujours présent dans les bons comme les mauvais moments. Merci à ma (grande) famille, le décompte prendrait quelques pages ... Un grand merci à mes parents sans qui je ne serai pas là tout simplement. Non les amis je ne vous ai pas oublié mais vous faîtes partie de la famille ;). Enfin merci à mon Poupi, toujours là.

Merci à toi lecteur qui t'apprêtes à lire cette thèse ou d'avoir au moins remercier avec moi les personnes sans qui cette thèse n'existerait pas.

"Engagez-vous, rengagez-vous, qu'ils disaient !"
— Les douzes travaux d'Asterix

Table des matières

Chapitre 1

Introduction

Le Calcul Haute Performance ou High Performance Computing (HPC) est un écosystème dynamique ou architectures et codes de calculs scientifiques sont en co-évolution permanente. Ce dynamisme impose aux développeurs d'adapter régulièrement leur logiciel pour exploiter au mieux tous les nouveaux ressorts technologiques.

Il existe cependant deux freins à la portabilité des codes sur de nouvelles architectures. Le premier est que les résultats d'une simulation numérique peuvent varier d'une architecture à une autre. Cette variation est souvent le signe d'une instabilité numérique dans le code qu'il convient alors de trouver. Le second point est que l'essor du *deep learning* a fait émerger de nouveaux formats (bfloat16, binary16) de calcul de plus petite taille ainsi que du matériel spécifique (TPU, Intel Nervana) qui permettent d'améliorer les performances d'un code. Les développeurs doivent donc utiliser ces formats pour tirer parti au mieux des ressources matériels.

Nous nous intéressons donc à la création d'outils de co-conception numérique pour aider le développeur à débuguer et améliorer ses codes HPC. Dans le contexte du HPC, les codes de simulation numérique utilisés sont volumineux et comportent beaucoup de lignes de codes (centaines de milliers). Il est donc difficile d'identifier les calculs qui impact la stabilité numérique des applications sans outils adaptés. La non-reproductibilité des calculs flottants provient du fait que le modèle flottant qui représente les nombres réels n'est pas associatif. Cela signifie que l'ordre dans lequel sont effectués les calculs flottants influe sur le résultat. Or beaucoup de facteurs influent sur l'ordre de opérations flottantes (optimisations du compilateur, vectorisation, ordonnancement des tâches parallèles, ...) Tous ces niveaux de non-reproductibilité font qu'il est difficile de reproduire des résultats. Lorsque la non-reproductibilité amène des erreurs numériques dans le code, il est là aussi difficile d'identifier les zones du programme responsables de l'instabilité.

Le problème d'optimisation de formats de calcul suit les mêmes contraintes que celui du débugage numérique, à savoir des codes avec de nombreuses lignes de codes. De plus, il existe une grande variété de formats à explorer. La taille de l'espace de solutions rend son parcours manuellement impossible et nécessite des outils adaptés qui puissent naviguer facilement dans ces espaces.

Cette thèse s'intéresse donc au développement d'outils de co-conception numérique pour aider le développeur d'applications HPC à pouvoir adapter ses codes sur des nouvelles architectures. Nous nous sommes particulièrement intéressé aux outils de débugage numérique et d'optimisation des formats de calcul flottant.

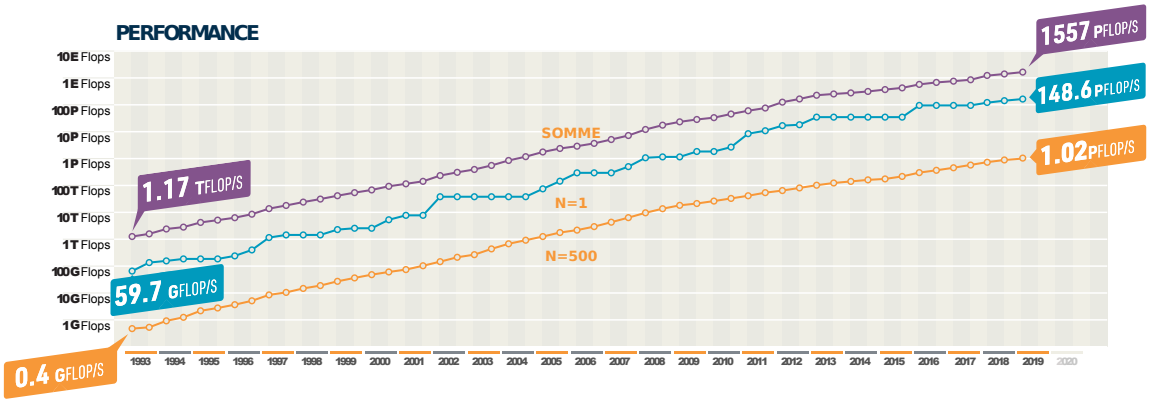


FIGURE 1.1 – Performance des machines du Top500.

1.1 Contexte

1.1.1 La simulation numérique

La simulation numérique cherche à modéliser un phénomène physique par la résolution d'équations à l'aide de calculs sur ordinateurs. La simulation numérique et le HPC sont devenus un atout majeur de recherche et développement pour le monde industriel et académique. On pourra citer Total et le CEA qui tiennent le haut du pavé du Top500 côté français. Comme tout bon écrit sur le HPC, nous ne manquons pas le passage obligé par le Top500, le classement bi-annuel qui référence les machines les plus puissantes de la Terre depuis 1993 (cf. figure 1.1). Á l'heure où cette thèse est rédigée, la machine occupant la première place du podium est le *Summit* du *DOE* Oak Ridge National Laboratory aux États-Unis avec 2414592 cœurs physiques pour une puissance crête de calcul de près de 200 PFlops [1], soit 2×10^{17} opérations flottantes par seconde.

Les opérations flottantes sont l'ensemble des opérations effectuées sur les nombres réels par opposition aux opérations sur les nombres entiers. Pour être plus précis, les nombres à virgule flottante, ou plus simplement *flottants*, représentent une approximation des nombres réels. Á ce titre, leur utilisation diffère de celle des réels puisque plusieurs propriétés comme l'associativité sont perdues. Puisque la simulation numérique est essentiellement composée de calculs, il est pertinent de mesurer la performance des architectures à partir du nombre d'opérations flottantes effectuées par seconde. Néanmoins, vitesse n'est pas synonyme de justesse. En plus d'être rapides, il est primordial que ces calculs soient également corrects. Si beaucoup d'efforts ont été menés pour améliorer la performance des applications peu l'ont été pour améliorer la qualité des résultats.

Néanmoins et lorsqu'on retrace la composition des machines supraflopiques du Top 500, ces efforts visant á améliorer la performance ont été nécessaires. On s'aperçoit en effet que les architectures utilisées ont évoluées au cours des années comme le montre la figure 1.2. De plus, la proportion de systèmes intégrant des co-processeurs est en nette croissance depuis 2006 (cf. 1.3), témoignant d'une imprégnation de plus en plus forte de l'hétérogénéité dans les codes de calculs.

1. C'est environ 600000× la puissance de la machine sur laquelle cette thèse est écrite. Bien entendu, ces performances mentionnées dans le Top500 sont à relativiser puisqu'elles sont théoriques ou proviennent du benchmark LINPACK, peu représentatif des applications réellement exécutées. On peut contraster ces résultats avec ceux du HPCG qui donne une performance crête à un peu moins de 3 PFlops.

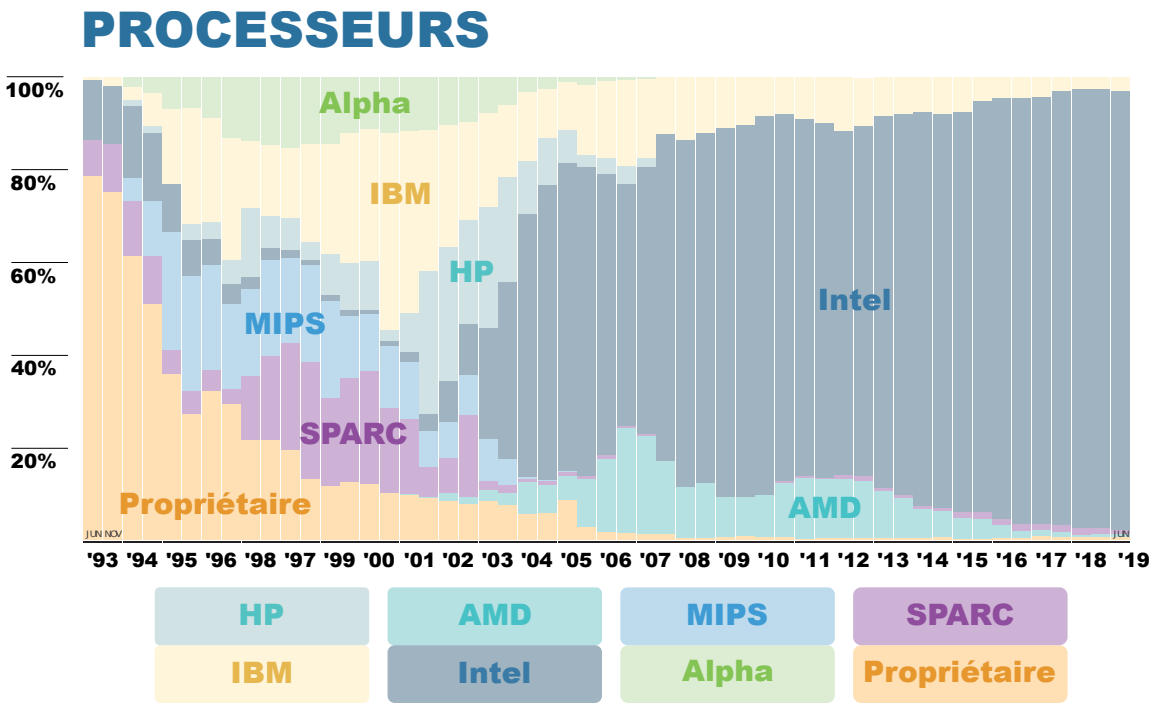


FIGURE 1.2 – Évolution des architectures des supercalculateurs du Top500 par constructeurs au cours des 25 dernières années. Bien qu'Intel domine le marché actuel, il existe 66 processeurs différents sur l'ensemble des 500 machines du Top500 de juin 2019 avec 46 processeurs différents uniquement pour Intel. Graphique tiré du Top500 [60].

L'accroissement de performance est donc le fruit d'une collaboration entre innovations matérielles et adaptations des codes de calculs.

1.1.2 La coconception numérique

La compétitivité entre industriels pousse ces derniers à maintenir un cycle d'innovation technologique soutenue avec, par exemple, un nouveau processeur tous les 2 ans en moyenne pour Intel depuis les années 70. Ces cycles s'accompagnent toujours d'une augmentation de la puissance de calcul. Focalisée sur la puissance mono-cœur par le passé, elle s'oriente dorénavant vers une parallélisation massive avec une densification du nombre de cœurs par chip.

Cette puissance croissante permet aux développeurs de codes de calculs scientifiques d'accroître la taille des problèmes visés mais également d'inclure des modèles physiques de plus en plus sophistiqués, intégrant des modèles multi-physiques et multi-échelles. Par exemple dans la simulation climatique, l'accroissement de puissance permet d'augmenter la résolution des mailles pour simuler des détails plus fins comme les petits nuages [2] mais également de prendre en compte plus d'interactions entre les différents climats tel que ceux de l'atmosphère, de l'océan ou des glaciers.

2. très important pour la qualité des simulations [170].

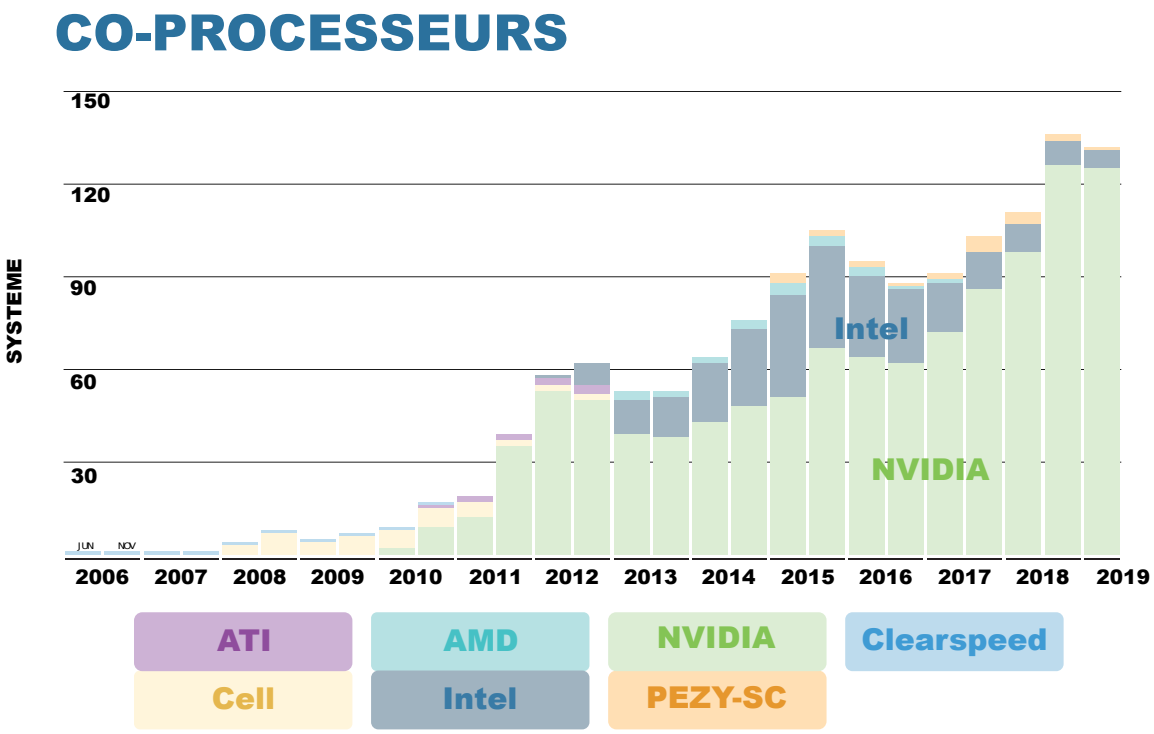


FIGURE 1.3 – Évolution des co-processeurs dans les machines du Top500 par constructeurs depuis 2006. L'introduction des GPGPU comme moyen de calcul à ouvert la voie à l'hétérogénéité dans les clusters HPC. Graphique tiré du Top500 [60].

L'apparition de nouveaux supports de calcul comme le Graphics Processing Unit (GPU) a également amené les développeurs à repenser leurs codes pour qu'ils s'adaptent au paradigme de calcul du GPU. Cependant, l'influence n'est pas à sens unique puisque le logiciel peut également influencer le matériel. On peut par exemple citer la norme IEEE-754 qui sera présentée en section 2.3.1. Cette norme qui régit le fonctionnement des calculs flottants dans la majorité des architectures modernes est née d'une volonté d'harmonisation des résultats entre architectures.

Aujourd'hui, le champ du deep learning bouscule à nouveau les codes du HPC. Le deep learning, qui étudie l'apprentissage par réseaux de neurones profonds, utilise intensivement des calculs de gradients. Or il a été montré [128] que les premières itérations des calculs de gradients pouvaient être faites dans une précision de calcul plus faible, ce qui améliore les performances du calcul. Cette technique dite de *précision mixte* permet en effet de densifier les calculs en intégrant plus d'éléments dans des unités vectorielles ou de réduire le trafic mémoire [3].

Ainsi, en plus des innovations régulières des Central Processing Units (CPUs), les supercalculateurs doivent également intégrer des nouveaux paradigmes de calculs qui émergent au cours des années comme les GPUs ou les FieldProgrammable Gate Array (FPGA). Cette intégration réduit la durée de vie d'un supercalculateur à environ 5 ans [152] pour maintenir sa compétitivité. Néanmoins c'est peu au regard des 20 à 40 ans de durée de vie des codes de calcul [179]. Les architectures évoluant plus rapidement que les codes, il est donc question pour les développeurs de savoir comment réussir la transition entre nouvelles générations de machines en, d'une part, préservant les qualités du logiciel, à savoir la performance et la qualité numérique des résultats, et, d'autre part, en exploitant au mieux les nouvelles ressources mises à disposition. L'objectif étant de réaliser cette transition en un minimum de temps et d'effort.

Dans le HPC, la coconception ou *codesign* promeut l'idée d'un développement simultané du logiciel et du matériel. Ce n'est plus seulement aux logiciels d'exploiter les ressources du matériel mais également aux matériels de s'adapter aux particularités des applications. Cependant, les architectures s'étant considérablement complexifiées et diversifiées au cours du temps, l'automatisation des phases de débogage et d'optimisation est primordiale puisque ces tâches sont difficilement réalisables manuellement.

La conception d'un code calcul nécessite plusieurs étapes de validation dont deux nous semblent cruciales. La première est de s'assurer que les calculs effectués soient valides d'un point de vue numérique, c'est-à-dire qu'ils aient du sens mathématiquement et qu'ils ne soient pas trop éloignés du modèle physique étudié. La deuxième concerne l'optimisation. Une fois la qualité des résultats validée, il est question de savoir si le format de mes calculs, c'est-à-dire l'espace nécessaire en machine pour les représenter, est adapté à mon problème ou s'il est surdimensionné. Dans ce dernier cas, il s'agit de réduire ce format pour qu'il épouse au mieux les besoins du problème.

À l'échelle d'un code de calcul HPC ces problèmes deviennent vite manuellement insolubles. En effet, les codes HPC sont des codes volumineux pouvant contenir jusqu'à plusieurs millions de lignes de codes et qui travaillent sur des données massives. Pour exploiter au mieux la performance des machines, les codes utilisent parfois plusieurs paradigmes de parallélisation qui introduisent chacun un indéterminisme sur les résultats. Une reproductibilité des calculs au bit près permet de soulever cet indéterminisme mais le prix à payer est assez élevé. D'autant plus que cette reproductibilité n'est pas la panacée.

En effet, la limitation de la performance mono-cœurs entraîne une augmentation de la parallélisation. Cette parallélisation massive conduit à des réseaux d'interconnexions entre unités de calculs de plus en plus grands et donc mécaniquement à un allongement du temps de transmission des données. Or la vitesse d'échange sur le réseau croit beaucoup moins vite que la puissance de calculs.

Il est donc nécessaire de réduire le volume de données échangées pour augmenter les performances.

1.2 Problématique

Les sections précédentes ont dressé les principales raisons qui poussent à concevoir des outils de co-développement pour le calcul numérique. Ces outils se doivent d'être extensibles (ou *scalables*) pour pouvoir fonctionner sur des codes HPC mais également être faciles d'utilisation pour les utilisateurs non-experts. Cette problématique de co-développement s'ouvre sur trois axes :

Comment estimer la justesse d'un code de calcul ?

Estimer la justesse d'un code de calcul est difficile car on ne peut pas toujours se comparer à une valeur de référence lorsque la solution du problème que l'on cherche à résoudre n'est pas connue. On doit alors se tourner vers les outils classiques d'analyse d'erreur que sont la forward error et backward error. Ces deux méthodes permettent de mesurer respectivement la distance entre notre solution et la solution exacte et la distance entre le problème résolu et le problème exact. Les deux problèmes, résolus et exacts diffèrent car le problème exact est discrétisé dans les flottants. Or ces méthodes mathématiques sont coûteuses en temps de recherche puisqu'il faut construire la preuve pour chaque schéma numérique. De plus elles s'automatisent difficilement sur des gros codes HPC que l'on cible. Par ailleurs, ces méthodes ne valident pas l'implémentation d'un schéma numérique dans un langage, ce qui ajoute un facteur d'incertitude.

De plus, même lorsque l'on sait que la qualité numérique de notre résultat est mauvaise, il est également difficile de trouver les endroits du code qui sont responsables de l'instabilité numérique détectée. Encore une fois, la taille des codes visés rend les approches manuelles difficiles à utiliser en pratique. Des outils pour aider le développeur dans sa recherche sont donc essentiels.

Comment optimiser les formats de calcul ?

L'émergence de nouveaux formats de calcul dans le monde du deep learning ainsi que des architectures spécifiques pour exploiter ces formats et atteindre de meilleures performances, pousse les développeurs HPC à regarder comment optimiser les formats de calcul de leur codes pour en améliorer les performances. Néanmoins deux problèmes limitent l'exploration de ces nouveaux formats.

Premièrement, il est difficile pour un utilisateur de savoir a priori où utiliser ces formats réduits dans son code. Cela nécessite de connaître finement les propriétés numériques de ces schémas, or il serait souhaitable d'offrir aux développeurs non experts en analyse numérique, un outil permettant d'explorer ces nouveaux formats.

Secondement, le nombre de formats flottants alternatifs à la norme IEEE-754 augmente régulièrement chaque année. Cependant, il est impossible pour un utilisateur de changer manuellement le type de ses calculs afin de tester un nouveau format. On souhaiterai donc avoir un moyen pour tester rapidement un nouveau format sans intervention sur le code.

Comment passer les méthodes d'analyse à l'échelle ?

Les codes HPC sont la cible des deux problèmes précédents. Mais comme nous l'avons montré dans cette introduction, la taille et la complexité de ces codes posent des problèmes pour passer des

méthodes d'analyse classiques à l'échelle. La taille des espaces de recherche induit que la localisation d'erreurs numériques ou la recherche de diminution de précision, nécessitent des méthodes robustes pouvant être utilisées sur des centaines de milliers de lignes de codes. Cette problématique est donc transverse aux deux problèmes précédents puisqu'elle cherche à décrire ces espaces de recherches et proposer des méthodes pour les explorer efficacement.

1.3 État de l'art pour les outils de *co-design* numérique

De nombreux outils existent dans le domaine de l'analyse numérique. On peut catégoriser ces outils suivant trois buts recherchés : la détection d'instabilité numérique, la validation des résultats et l'optimisation des calculs. La détection d'instabilité numérique consiste à détecter des erreurs et à pointer leur origine. L'optimisation des calculs permet de réduire la taille des formats de calcul utilisée pour tirer parti des nouvelles possibilités des architectures comme par exemple le bfloat16 [117] ou les instructions Intel VNNI [4]. Cela vise aussi à diminuer l'empreinte mémoire pour réduire les volumes de communications échangées dans les applications parallèles. Enfin, la validation consiste à s'assurer que les calculs sont corrects, c'est-à-dire que les résultats ne sont pas trop entachés d'erreurs numériques et qu'ils restent dans un intervalle restreint.

Nous les présentons sous deux angles : les méthodes d'analyse statique et d'analyse dynamique. Que ce soit pour la détection, la validation ou l'optimisation, les avantages et inconvénients restent sensiblement les mêmes pour chacune de ces grandes familles.

1.3.1 L'analyse statique

L'analyse statique analyse un programme sans l'exécuter d'où le qualificatif statique. Puisque le programme n'est pas exécuté, les valeurs des variables du programme sont inconnues ce qui oblige à raisonner sur des ensembles de valeurs possibles. Les analyses produites sont ainsi rigoureuses mais raisonnent dans des cadres mathématiques éloignés du code source originel, ce qui demande une traduction du code source vers des formalismes adaptés. Si ces méthodes ont l'avantage d'être rigoureuses, elles souffrent néanmoins d'un très mauvais passage à l'échelle et restent cantonnées à l'étude de programmes de taille raisonnable. Les méthodes d'analyse statique se distinguent en trois groupes :

- Preuve formelle
- Satisfiability Modulo Theory (SMT)
- Interprétation abstraite

Preuve formelle

La preuve formelle fournit une preuve mathématique d'une spécification sous la forme d'un programme informatique. Chaque étape du programme fournit une preuve de la démonstration qui est vérifiée par un assistant de preuve ce qui réduit les erreurs humaines d'une démonstration.

Il existe plusieurs outils pour raisonner formellement sur les calculs flottants [16, 84] dont on peut trouver un détail approfondi dans [13]. Les méthodes formelles pour la vérification de calculs flottants ont été utilisé pour prouver la correction de divers programme comme l'implémentation d'une exponentielle [84] ou la correction d'un code de 30 lignes d'une simulation de propagation d'ondes acoustiques en 1D [14]. La garantie donnée par ces méthodes est du plus haut niveau

puisque c'est une preuve mathématique vérifiée par l'ordinateur. Néanmoins, l'expertise et le temps nécessaires pour fabriquer ces preuves cantonnent leur utilisation à de petits programmes.

Les solveurs SMT

Les solveurs SMT testent la satisfiabilité d'une formule de la logique du premier ordre par rapport à une théorie sous-jacente, d'où le *modulo* une théorie. Les solveurs Z3 [47], MathSAT [31] ou CORAL [168] permettent de raisonner sur les nombres flottants. Ces outils raisonnent sur les flottant au niveau du bit ce qui produit des formules très grandes à analyser. FPhile [143] vérifie la stabilité d'un programme annoté sous forme d'assertions. Il utilise Z3 pour s'assurer que les versions binary32 et binary64 du programme donnent un résultat assez proche. Leeser et al. [115] utilisent des transformations d'arithmétique flottante vers l'arithmétique réelle pour raisonner sur les réels avec des fonctions d'arrondi permettant ainsi d'accélérer la résolution.

Interprétation abstraite

L'interprétation abstraite [39] modélise la sémantique d'un programme par une abstraction plus ou moins éloignée de la sémantique concrète du programme, c'est-à-dire tous les états possibles qui seront réellement atteignables par le programme. L'interprétation abstraite permet notamment de raisonner sur des ensembles de valeurs pour les nombres flottants par exemple. Ceci permet d'obtenir des bornes rigoureuses sur le résultat d'un calcul à priori sans connaître les valeurs prises.

Validation des calculs

Plusieurs outils d'analyse statique existent pour obtenir une borne sur l'erreur de calcul.

Ariadne [6] génère des expressions symboliques qu'il passe au solveur Z3 pour détecter des exceptions flottantes dans les calculs.

CompCert [116] est un compilateur C entièrement vérifié par l'assistant Coq. La vérification assure que la sémantique du code source est la même que celle du code généré. La sémantique sur les opérations flottantes a également été prouvée [15] ainsi que certaines optimisations qui préservent la sémantique de la norme IEEE-754.

Gappa [45] prouve automatiquement la validité d'une proposition logique impliquant des nombres flottants. Il utilise pour cela l'arithmétique d'intervalles ce qui lui permet de donner une borne sur l'erreur du résultat. Gappa génère une preuve qui peut être vérifiée par le prouveur automatique de preuve Coq [91].

Fluctuat [52, 74] utilise une arithmétique affine pour calculer la valeur d'un résultat et une borne sur l'erreur commise. L'arithmétique affine conserve les dépendances linéaires entre variables ce qui permet à Fluctuat de remonter aux variables responsables des erreurs les plus importantes commises sur le résultat.

FPTaylor [167] utilise une expression symbolique des erreurs d'arrondi à base d'un développement de Taylor. FPTaylor se ramène ensuite à un problème d'optimisation pour trouver le point d'erreur maximale ce qui fournit une borne sur l'erreur de l'expression. Il vérifie les inégalités non-linéaires à l'aide de l'assistant de preuve HOL-LIGHT [86].

LifeJacket [140] vérifie les optimisations sur les flottants du compilateurs Low Level Virtual Machine (LLVM). Les spécifications d'une optimisation à vérifier sont traduites par LifeJacket en

un ensemble de contraintes et sont passées au solveur Z3, qui s'assure que les optimisations sont valides.

Icing [8] fournit un compilateur pour le langage du même nom qui étend la vérification des optimisations sur les nombres flottants aux optimisations *fast-math*, c'est-à-dire des optimisations qui ne préservent pas les propriétés de la norme IEEE-754.

PRECiSA [176] utilise une arithmétique d'intervalles combinée á un algorithme de branch and bound pour subdiviser rapidement le domaine d'évaluation. Cela lui permet de calculer les bornes d'erreurs d'une expression symbolique associées à un calcul et de générer un certificat de la preuve des bornes calculées.

Real2Float [121] utilise la programmation Semie-Définie Positive (SPD) qui étend l'optimisation linéaire aux matrices symétriques semie-définies positives. Cela permet à Real2Float d'obtenir des bornes d'erreurs plus précises.

Rosa [43] est un compilateur qui, à partir d'une spécification, génère un programme équivalent en arithmétique flottante. Une spécification est une fonction á laquelle sont ajoutées des pré et post conditions. L'emploie de pré et postconditions permet de spécifier des incertitudes sur des variables. Rosa utilise le démonstrateur de preuve Z3 [47] pour prouver que le programme généré respecte bien les conditions en entrée.

Sardana [96] réécrit des expressions flottantes en utilisant une représentation parcimonieuse des expressions équivalentes. Sardana recherche ensuite l'expression équivalente qui minimise l'erreur sur l'ensemble des valeurs possibles de chacune des variables.

Optimisation des calculs

Les outils utilisés par le calcul de bornes d'erreurs servent également de briques de base pour optimiser les formats de calcul.

Anton [42] génère automatiquement des versions en précision mixe de spécifications dans la même veine que Rosa. Anton propose une réécriture des formules qui permet d'améliorer les performances tout en dégradant la qualité du résultat. Comme Rosa, il génère également des certificats de preuves sur les bornes d'erreurs obtenues à l'aide du solveur Z3.

FPTuner [167] utilise FPTaylor pour générer une version en précision mixe d'une expression flottante. FPTuner transforme l'expression en un problème d'optimisation et cherche à minimiser la précision des variables de binary64 à binary32. De plus, FPTuner cherche à minimiser le nombre de conversions entre formats et á uniformiser la précision des instructions pouvant être vectorisées pour préserver les optimisations.

Salsa [41] transforme un code source à source pour qu'il utilise un format de calcul plus petit tout en préservant une erreur d'arrondi pour certains points du calcul définis par l'utilisateur. Salsa utilise une analyse en avant pour propager les erreurs d'arrondi à partir des entrées et utilise une analyse en arrière pour calculer la précision minimale nécessaire pour assurer une petite erreur. Ces contraintes sont ensuite vérifiées par le solveur Z3.

Conclusion

Ces outils fournissent des bornes précises sur l'erreur d'un calcul ou sur de petits programmes mais ne sont pas conçus pour analyser des codes HPC incluant de nombreuses lignes de code. D'autre part, si les efforts sont orientés pour automatiser le calcul des bornes, un travail conséquent

de la part du programmeur pour traduire son code dans les formalismes proposés est nécessaire, ce qui en limite l'usage sur de très gros codes.

1.3.2 L'analyse dynamique

L'analyse dynamique analyse un programme durant ou après son exécution, contrairement à l'analyse statique. Les méthodes d'analyses dynamiques sont plus efficaces que les méthodes statiques traditionnelles et permettent d'analyser des programmes plus grands. Cependant, elles ne fournissent pas le même niveau d'analyse que les analyses statiques et restent dépendantes du jeu de données sur lequel on les évalue.

Arithmétiques stochastiques

Les arithmétiques stochastiques sont des arithmétiques qui modélisent les erreurs du modèle flottant à l'aide de variables aléatoires. Le but est d'exécuter plusieurs fois les opérations flottantes d'un programme pour estimer la variabilité des résultats. Il existe deux principaux modèles : les méthodes Contrôle et Estimation Stochastique des Arrondis de Calculs (CESTAC) et Monte Carlo Arithmetic (MCA). La méthode CESTAC [181] introduit un arrondi aléatoire sur chacune des opérations flottantes. La méthode MCA ajoute un bruit aléatoire dont la magnitude peut être contrôlée par l'utilisateur. La méthode MCA sera détaillée dans la section 2.6.2. Les outils implémentant des arithmétiques stochastiques sont les plus proches conceptuellement de Verificarlo, l'outil auquel cette thèse a contribué. Nous allons donc présenter en détail ces outils qui lui sont proches.

Control of Accuracy and Debugging for Numerical Applications (CADNA) [97] est la première bibliothèque à avoir implémenté la méthode CESTAC. La bibliothèque CADNA fournit des types flottants spécifiques appelés types stochastiques ainsi que l'ensemble des opérateurs arithmétiques de base $\{+, -, \times, /\}$, les comparaisons et les fonctions élémentaires sur les entrées/-sorties ou les conversions par exemple. La sémantique de ces opérations est définie par la Discrete Stochastic Arithmetic (DSA). Pour chaque opération arithmétique flottante, CADNA exécute trois fois chaque opération de manière synchrone. L'arrondi de l'opération flottante est tiré au sort avec la même probabilité [3] d'être vers le haut que vers le bas. Les trois résultats permettent à CADNA d'estimer le nombre de chiffres significatifs du résultat. La méthode CESTAC introduit également la notion de zéro informatique qui représente une valeur numérique n'ayant aucun chiffre significatif correct. L'exécution synchrone et la notion de zéro informatique permettent à CADNA de détecter les instabilités numériques comme les branchements instables ou les annulations catastrophiques. L'utilisation de types non natifs dans CADNA oblige à réécrire l'ensemble du code à analyser. Cet exercice est coûteux sur des codes de plusieurs milliers de lignes de code bien qu'il existe un outil permettant d'automatiser ce processus (`cadnaizer`). D'autre part, l'utilisation de types composés peut empêcher le déclenchement de certaines optimisations par le compilateur ce qui peut masquer certains problèmes d'instabilités numériques liés aux optimisations du compilateur.

VERROU [4] [63] est un outil qui implémente la méthode CESTAC à l'aide d'une instrumentation binaire par Valgrind [137]. Valgrind permet à Verrou d'instrumenter dynamiquement les opérations flottantes d'un code assembleur sans le recompiler. Verrou remplace les opérations flottantes par des équivalents en arithmétique stochastique et propose plusieurs modes pour introduire le bruit. Contrairement à CADNA, Verrou est un outil asynchrone. Différents flots d'exécution

3. ce tirage au sort est biaisé pour qu'au moins un des tirages soit différent des deux autres.
4. en hommage à CADNA.

peuvent être pris pour l'exécution d'un même programme ce qui oblige à un post-traitement pour détecter les branchements instables. Verrou utilise pour cela l'outil gcov de GNU pour détecter les opérations qui n'ont pas été appelées le même nombre de fois à travers différentes exécutions.

Verificarlo [56] est un compilateur basé sur LLVM qui remplace automatiquement les opérations flottantes par leur équivalent MCA. En remplaçant les opérations après les passes d'optimisation, Verificarlo permet de détecter l'influence des optimisations du compilateur sur la justesse des calculs. Cette thèse ayant contribué à Verificarlo, cet outil fera donc l'objet d'une analyse approfondie dans le chapitre 3.

Promise [75, 76] est un outil basé sur CADNA qui utilise l'algorithme de Δ-debugging pour trouver les opérations flottantes pouvant être exécutées en binary32. Promise cherche à maximiser le nombre d'opérations à changer en binary32 et propose deux méthodes pour vérifier l'exactitude des résultats. Une "full stochastic"compare le nombre de chiffres significatifs (s) du résultat des solutions en précision mixe au s de l'exécution de référence (en binary64). Les solutions sont rejetées si la différence est inférieure à un seuil défini par l'utilisateur. L'autre méthode, moins coûteuse en temps, compare l'erreur relative entre la solution en précision mixe et le résultat de référence calculé avec CADNA.

Les multi-précisions

Les méthodes multi-précisions comparent le résultat d'un programme avec une exécution de celui ci en précision supérieure. Le changement de précision peut se faire opération par opération grâce á une shadow memory afin de traquer les erreurs numériques d'une opération donnée. Ce changement de précision peut également s'effectuer sur un ensemble d'opérations, lorsque l'on souhaite réduire la précision des opérations d'un code afin d'augmenter ses performances (technique de précision mixe). Nous dressons une liste non exhaustive des principaux outils implémentant ce genre de technique.

Détection

Lam [109] utilise DynInst pour instrumenter un programme binaire et détecter dynamiquement les annulations catastrophiques. Les opérations qui perdent un nombre de bits supérieur à un seuil défini par l'utilisateur déclenchent un avertissement. La localisation des annulations est visible à travers une interface graphique. Cette méthode soulève beaucoup d'avertissements sans conséquence sur l'erreur globale.

FPDebug [11] utilise une shadow memory à travers Valgrind pour calculer chaque opération flottante dans une précision étendue. Il garde également une trace du chemin d'exécution pour localiser la source d'erreur. Enfin, FPDebug lève des avertissements chaque fois que l'erreur entre l'opération flottante et sa version en précision étendue est au dessous d'un seuil utilisateur. Comme la méthode précédente, cette méthode soulève des avertissements qui sont sans conséquence sur l'erreur globale.

Herbgrind [156] détecte et localise les erreurs numériques d'un programme en instrumentant le code binaire à l'aide de Valgrind. Herbgrind utilise une shadow memory pour maintenir une version des calculs en précision étendue. Pour chaque opération, Herbgrind déclenche une alerte lorsque l'erreur entre la version flottante et la version en précision étendue est en dessous d'un certain seuil (erreurs locales). Herbgrind utilise une taint analysis pour propager les erreurs locales et mesurer leur influence sur le résultat (erreur globale). Enfin, il garde une expression symbolique des calculs pour donner les formules responsables de l'erreur globale.

S^{3}FP [27] génère des bornes sur l'imprécision des données. À partir d'intervalles sur les entrées données par l'utilisateur, S^3FP échantillonne ces intervalles et évalue, pour chaque

point, l'erreur relative entre le résultat du calcul évalué dans la précision originale et le résultat évalué dans une précision supérieure. Les calculs en précision supérieure sont effectués avec une shadow memory. Une heuristique de recherche aléatoire permet de trouver rapidement des pires cas locaux, où l'erreur relative est la plus grande.

Amélioration

Herbie [144] réécrit des formules pour en améliorer l'exactitude. Herbie propose une réécriture par morceaux avec différentes formule suivant l'intervalle de valeurs considéré. D'autre part, Herbie possède un système de réécriture avec des développements limités qui lui permet de trouver des réécritures avec moins de termes. Pour tester ces solutions, Herbie échantillonne l'espace des flottants en incluant les points spéciaux comme les infinis ou les dénormalisés.

Optimisation

Blame analysis [154] utilise une analyse par shadow memory pour évaluer les variables qui peuvent être passé en binary32. Pour cela, chaque opération est exécuté plusieurs fois avec des précisions différentes pour chaque opérandes. Les solutions dont l'erreur globale est inférieure au seuil défini par l'utilisateur sont sauvegardées et réutilisées pour les opérations suivantes et ce jusqu'à la fin du programme. Cette technique permet de trouver les variables pouvant être remplacé en binary32 avec une seule passe d'analyse.

CRAFT [108] utilise DynInst [19] pour instrumenter un programme binaire et remplacer les opérations flottantes binary64 par des opérations en binary32. CRAFT utilise une approche en largeur pour explorer les sections de code à tester et les teste récursivement à travers les structures du codes en commençant par les modules, puis les fonctions et les basic blocks. L'utilisation de la topologie des codes permet d'élaguer rapidement des grosses sections de code.

FlexFloat [171] définit un type flottant générique en C++ paramétrable par le nombre de bits pour l'exposant et pour la mantisse. Couplé à fpPrecisionTuning, il permet de minimiser la précision et la dynamique des variables. Cependant et comme fpPrecisionTuning, tous les types des variables utilisant FlexFloat doivent être modifiés.

fpPrecisionTuning [90] minimise la taille de la mantisse de variables choisies par l'utilisateur, tout en gardant une erreur globale inférieure à un seuil. Contrairement aux outils précédents, celui ci explore des précisions à un grain fin permettant des analyses plus fines sur les besoins en précision des applications. fpPrecisionTuning calcule les groupes d'influence de chacune des variables à l'aide du graphe de dépendance pour capturer l'impact du changement de précision d'une variable sur les autres variables du programme. Pour chaque groupe d'influence, il cherche ensuite à minimiser la précision d'une variable par dichotomie. fpPrecisionTuning construit ensuite l'union des précisions des groupes d'influences en prenant le maximum à chaque fois. Contrairement aux outils précédents, fpPrecisionTuning explore des précisions à un grain fin permettant des analyses plus fines sur les besoins en précision des applications. Il nécessite cependant de réécrire l'ensemble des opérations avec MPFR et de modifier son code pour qu'il affiche des sorties attendues.

Precimonious [155] utilise l'algorithme du Δ-debugging pour trouver les variables binary64 d'un programme qui peuvent être remplacées par du binary32 afin d'améliorer les performances du programme. Precimonious utilise une passe LLVM pour remplacer automatiquement les opérations flottantes incriminées. Une valeur de référence ainsi qu'un seuil à ne pas dépasser sont fourni par l'utilisateur pour guider la recherche.

STOKE [159] utilise une optimisation stochastique pour trouver une version d'un code as-

sembleur qui soit plus performante et qui respecte des contraintes sur l'erreur globale. STOKE travaille sur des noyaux de fonctions en assembleur x86-64. Il remplace aléatoirement les instructions jusqu'à aboutir à un état minimal stable. STOKE, en échantillonnant les valeurs en entré, teste que l'erreur globale est en dessous d'un seuil. STOKE est limité à de petits noyaux de calculs et ne marche que sur les codes avec des boucles.

TAFFO [26] diminue la précision des opérations flottantes d'un programme spécifiées par l'utilisateur à l'aide d'annotations. Ces annotations permettent de spécifier l'intervalle qui contient une valeur flottante. TAFFO utilise une arithmétique d'intervalles pour calculer la dynamique des variables et les erreurs de troncatures afin de choisir la précision adéquate. TAFFO est implémenté comme une passe d'optimisation LLVM et remplace automatiquement les opérations flottantes par des opérations en précision plus faible ou en précision fixe.

Conclusion

Les méthodes dynamiques, suivant les approches choisies, permettent d'analyser rapidement des codes de tailles raisonnables. Elles sont particulièrement bien adaptées dans le cas de débogage puisqu'elles permettent de trouver rapidement des instabilités durant un exécution. Elles fournissent également des informations utiles sur le comportement d'un programme de manière générale. Le point noir des méthodes dynamiques est le manque de généralisation puisque l'analyse faite est toujours liée au jeu de données en entrée.

Concernant l'optimisation de précision, on note que les outils présentés se focalisent sur la dimension spatiale mais n'abordent pas la dimension temporelle. Néanmoins, cette dimension est importante comme nous le verrons dans le chapitre 4

1.4 Contributions

Les contributions de cette thèse sont triples.

1.4.1 Contributions au projet Verificarlo

Cette thèse a contribué au projet Verificarlo) [56]. Verificarlo est un projet open-source disponible sur github (`github.com/verificarlo/`) développé principalement par l'Université de Versailles Saint Quentin en Yvelines et Intel. Verificarlo est un compilateur basé sur LLVM qui remplace automatiquement les opérations flottantes par des appels génériques. Cette passe dite d'instrumentation est effectuée après l'application des passes d'optimisations ce qui permet de mesurer les effets des optimisations sur la stabilité numérique du code. L'utilisateur choisit ensuite avec quel backends il va lier ces appels génériques durant l'exécution. Plusieurs backends sont implémentés dans Verificarlo, notamment les backends `MPFR` et `QUAD` qui permettent d'utiliser l'arithmétique de Monte Carlo (MCA), une arithmétique stochastique qui modélise les erreurs du modèle flottant par l'introduction de bruit aléatoire. Deux backends ont été implémentés durant cette thèse : le backend `BITMASK` et le backend `VPREC`. En plus d'une participation au design de Verificarlo, les contributions à ce projet sont :

- Développement d'un backend qui implémente un modèle de bruit à base de masques de bits. Le bruit introduit par ce backend est biaisé, ce qui en fait un mauvais candidat pour l'analyse de comportements numériques subtils. Néanmoins, il possède un surcoût très léger ($\times 3$) ce qui en fait un backend de première approximation efficace.

- Optimisation du backend `MPFR` avec une accélération $\times 2,5$ comparé à la version originale. Cette optimisation utilise efficacement les structures de la bibliothèque MPFR.
- Contribution au backend `VPREC` qui permet de simuler n'importe quel format flottant contenu dans un binary64. Le backend `VPREC` offre un surcoût léger et une grande liberté à l'utilisateur puisque celui ci peut définir le format qu'il souhaite simuler au bit près en spécifiant sur combien de bits sont codés l'exposant et la mantisse. Il offre de plus de bonnes performances avec un surcoût de $\times 8$ en moyenne.

1.4.2 Analyse de la qualité numérique au cours du temps

Lorsque l'on s'intéresse à la qualité numérique des calculs flottants, il ne suffit pas de s'intéresser à la qualité du résultat final mais il est également important de vérifier comment évolue le calcul au cours du temps.

Veritracer est un visualiseur de la qualité numérique des calculs flottants au cours du temps. Cet outil est basé sur la génération de traces temporelles qui sont enrichies d'informations de contexte récupérées lors de la compilation. Veritracer fait partie de la suite d'outils de Verificarlo et fournit un ensemble d'outils Python pour compiler, exécuter, analyser et visualiser.

Veritracer a permis de détecter une instabilité numérique sur le code industriel ABINIT, de la corriger et de vérifier la correction proposée. Ceci a démontré le passage à l'échelle de l'outil sur un code HPC. Veritracer a été publié au 25ème *Symposium on Computer Arithmetic* (ARITH) [24].

1.4.3 Optimisation de la précision de calcul au cours du temps

Nous avons proposé un outil d'exploration automatique pour réduire la précision des calculs des schémas itératifs. La contribution majeure est l'optimisation suivant la dimension temporelle. Ainsi, la précision est optimisée pour chaque invocation d'une variable et non pour l'ensemble des invocations comme il est d'usage dans les approches précédentes. De plus, l'utilisation du backend `VPREC` permet une optimisation à grain fin puisque la précision proposée est au bit près.

Nous avons démontré l'efficacité de la méthode sur le code industriel Yales2 avec l'implémentation d'une version en précision mixe à partir des solutions trouvées par notre algorithme. Cette version réduit le volume des communications échangées entre rangs Message Passing Interface (MPI) de 28% et permet d'obtenir jusqu'à $1.30\times$ facteurs d'accélération comparé à la version originale.

Ces travaux ont été présentés à la 25ème International European Conference on Parallel and distributed Computing (Euro-Par) [25].

1.5 Plan de la thèse

Le chapitre 2 introduit le modèle à virgule flottante. Il présentera les différentes notions d'erreurs du modèle flottant et comment les mesurer. Puis il présentera le calcul flottant dans les architectures utilisées pour le HPC. Enfin, il exposera les problèmes liés à la reproductibilité des calculs dans un contexte HPC puis terminera par une présentation de l'arithmétique stochastique et du modèle MCA.

Le chapitre 3 présente en détail l'outil Verificarlo et son fonctionnement. Une présentation détaillée des backends et des optimisations proposées pour en améliorer la performance est donnée. Ce chapitre se termine également par une revue des problèmes qui se posent lors de

l'application de Verificarlo sur des applications HPC et les différentes méthodes existantes et proposées pour rendre possible la mise à l'échelle.

Le chapitre 4 détaille l'outil Veritracer et son fonctionnement dans la chaîne de compilation de Verificarlo. On y explique l'instrumentation automatique du code ainsi que les méthodes utilisées pour récupérer les informations de débogages. Enfin, le chapitre se termine par l'application de Veritracer sur le code industriel ABINIT, de la détection d'une erreur numérique jusqu'à sa réparation.

Le chapitre 5 présente les principaux outils d'optimisation de précision et la méthode que nous avons proposé dans cette thèse. Un exemple détaillé sur la méthode de Newton-Raphson permet de comprendre pas à pas l'algorithme de recherche de précision minimale que nous avons développé. Ce chapitre se termine par le cas d'étude sur YALES2 qui montre la viabilité de la méthode et de son passage à l'échelle ainsi que les performances dégagées grâce à notre analyse.

Le chapitre 6 présente la conclusion de cette thèse, comment elle a répondue aux problématiques posées et les perspectives envisagées autour des travaux menés.

Chapitre 2

L'arithmétique des ordinateurs

L'arithmétique des ordinateurs discrétise les nombres réels en nombres machines, notamment grâce à la représentation en virgule flottante. Longtemps à la charge des constructeurs, la norme IEEE-754 est venue standardiser les calculs flottants et fait maintenant référence au sein des calculateurs. Cependant, la complexification des machines demande à repenser les formats IEEE-754. De plus, la reproductibilité des calculs est un enjeu crucial dans les applications scientifiques. Les arithmétiques stochastiques peuvent offrir une solution élégante et rigoureuse aux vues des dernières contributions théoriques apportées.

Dans ce chapitre, nous présenterons une définition générale du modèle flottant puis les formats utilisés actuellement dans les applications ainsi que les différentes architectures sur lesquels ils reposent. Nous aborderons ensuite les erreurs de calculs que la précision finie et le non-déterminisme introduisent. Enfin nous terminerons par une description des arithmétiques stochastiques qui permettent de détecter la perte d'information durant les calculs flottants.

2.1 Le modèle à virgule flottante

La virgule flottante, par opposition à la virgule fixe, est une représentation des nombres réels où la virgule qui sépare la partie entière de la partie fractionnaire d'un nombre est placée de manière variable. Prenons comme exemple la constante de Planck (notée h) utilisée en mécanique quantique pour représenter la taille d'un quanta et regardons différents moyens de la représenter :

1. $+0,00000000000000000000006626070040$
2. $+0,0006626070040 \times 10^{-30}$
3. $+6,626070040 \times 10^{-34}$
4. $+6626070040 \times 10^{-43}$

Ces quatre notations représentent toutes la valeur de h mais n'ont pas le même format. La première est à virgule fixe et les chiffres sont placés suivant leur positionnement par rapport au premier 0. Si cette représentation est commode dans la vie courante, elle est mal adaptée aux nombres manipulés en science. En effet, elle ne permet pas de représenter de manière concise les valeurs de grande ou de petite magnitude ni de voir rapidement les ordres de grandeurs manipulés.

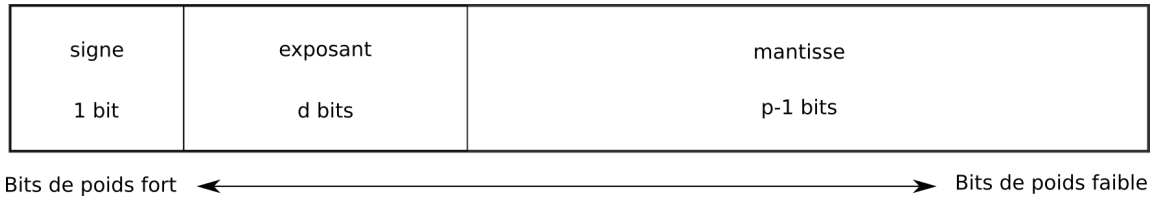


FIGURE 2.1 – Les champs du nombre flottant

Les trois dernières sont des représentations à virgule flottante. On parlera aussi de *notation scientifique*. Chacun des nombres est construit de la manière suivante :

$$\pm m \times \beta^e$$

où β est la base de calcul (ici décimale), m est un nombre réel appelé *mantisse* (ou *significande*) et e est l'exposant qui détermine l'ordre de magnitude. Le nombre de chiffres qui compose la mantisse est appelé la *précision*.

L'avantage de ce format est d'offrir une représentation plus concise que le format à virgule fixe. En effet, moins de bits sont nécessaires pour encoder la magnitude des nombres de grandes amplitudes. À titre d'exemple, un gogol (10^{100}) requiert 8 bits (1 bit de mantisse + 7 d'exposant) en virgule flottante contre 333 bits en notation entière. Cette représentation parcimonieuse est bien adaptée au cas des ordinateurs qui sont des machines disposant d'une mémoire finie.

L'inconvénient est qu'il n'y a pas de représentation unique. En effet, les valeurs 2, 3 et 4 sont chacune une représentation valide de h. Nous verrons dans la section 2.1.1 comment pallier ce problème.

2.1.1 Représentation des nombres réels en machine

Nous avons vu que n'importe quel nombre réel pouvait s'écrire suivant la notation scientifique. Néanmoins, un ordinateur manipule une information de taille finie. L'unité de calcul, pour quelle soit performante, représente les nombres avec un nombre limité de bits. Nous allons maintenant définir la structure d'un nombre flottant et voir les différents champs qui le composent. Si ces champs sont analogues à ceux de la notation scientifique, leur taille diffère d'une norme à l'autre comme nous le verrons à la section 2.3.1. La structure du nombre flottant est résumée dans la figure 2.1.

L'arithmétique flottante représente un nombre flottant x par le triplet (s, m, e) pour un $\beta \geqslant 2$ fixé :

$$x = (-1)^s.m.\beta^e.$$

- Le **signe** : $s \in \{0, 1\}$
  Il permet de coder les nombres négatifs et est représenté sur 1 bit.
- La **mantisse** : $m = m_0.m_1m_2...m_{p-1},\ m_i \in [0, \beta - 1],\ p \geqslant 2$
  avec $p = |m|$ la **précision** du format.
- L'**exposant** : $e \in [e_{min}, e_{max}]$
  Il caractérise la dynamique, i.e. le plus petit (e_{min}) et le plus grand (e_{max}) exposant représentable qui dépend du nombre de bits notée d.

Nous avons vu précédemment qu'une valeur pouvait avoir plusieurs représentations en virgule flottante. Pour pallier ce problème, on introduit la forme *normale*.

Définition 2.1.1 (Forme normale)**.** Un nombre flottant x est dit en forme normale si sa mantisse est de la forme :

$$m = m_0, m_1 m_2 ... m_{p-1}$$
$$m_0 \in [1, \beta - 1],\ \underset{i>0}{m_i} \in [0, \beta - 1].$$

Ceci impose que la virgule soit placée après le premier chiffre de la mantisse et que celui ci soit non nul. Cela implique également que la mantisse $1 \leqslant m < \beta$. Si l'on reprend notre exemple sur la constante de Planck, sa forme normale est donc $+6,626070040 \times 10^{-34}$ avec pour signe $s = 0$, pour mantisse $m = 6,626070040$ et pour exposant $e = -34$.

Un nombre est dit *dénormalisé* s'il n'est pas en forme normale. On parle également de *sous-normaux* pour les nombres dont la mantisse $m < 1$. Dans la suite de cette thèse, nous considérerons les nombres dénormalisés comme sous-normaux.

Définition 2.1.2. (Nombre sous-normaux) Un nombre x est sous-normal si

$$e = e_{min}$$
$$m = 0, m_1 m_2 ... m_{p-1}$$
$$m_i \in [0, \beta - 1].$$

Dans le cas d'une base binaire, la convention du *bit implicite* permet de ne pas coder le premier chiffre de la mantisse. En effet, les nombres normalisés s'écrivent de la forme $1.m_1...m_{p-1}$ et ceux dénormalisés de la forme $0.m_1...m_{p-1}$. Cela permet d'économiser un bit en conservant la précision égale à $p = |m| + 1$. En pratique, le fait que le nombre soit normalisé ou non est encodé dans l'exposant.

Définition 2.1.3 (Bit implicite)**.** Dans le cas $\beta = 2$, si x est un nombre flottant de mantisse $m = m_0, m_1 m_2 ... m_{p-1}$ alors la convention du bit implicite implique de n'encoder que la **fraction** $f = m_1 m_2 ... m_{p-1}$ en machine.

Ces deux définitions nous permettent de définir trois valeurs limites, le plus petit dénormalisé x_{sub}, le plus petit normal x_{min} et le plus grand normal x_{max} représentable dans un format :

- $x_{sub} = \beta^{e_{min}-p+1}$
- $x_{min} = \beta^{e_{min}}$
- $x_{max} = (\beta - \beta^{1-p}).\beta^{e_{max}}$

Enfin, on note $\mathbb{F}_{\beta,d,p}$ l'ensemble des nombres flottants représentables dans une base β sur une dynamique d avec une précision p . Pour la dynamique, on suivra la convention utilisée par la norme IEEE-754, à savoir $e_{max} = 2^{d-1} - 1, e_{min} = 1 - e_{max}$.

2.1.2 Arrondis

Le format flottant a une précision limitée déterminée par la taille de la mantisse choisie. La question des arrondis se pose lorsqu'un format d'origine a une précision plus grande que le format d'arrivée. Par exemple, si l'on souhaite convertir h dans un format à 3 chiffres de précision, deux possibilités s'offrent à nous : soit arrondir vers le bas $6,62 \times 10^{-34}$ soit vers le haut $6,63 \times 10^{-34}$.

Il existe en fait plusieurs façons d'arrondir un nombre flottant mais seulement quatre modes définis dans la norme IEEE-754 ([34] et section 2.3.1) seront présentés. Ces quatre modes sont également illustrés par la figure 2.2.

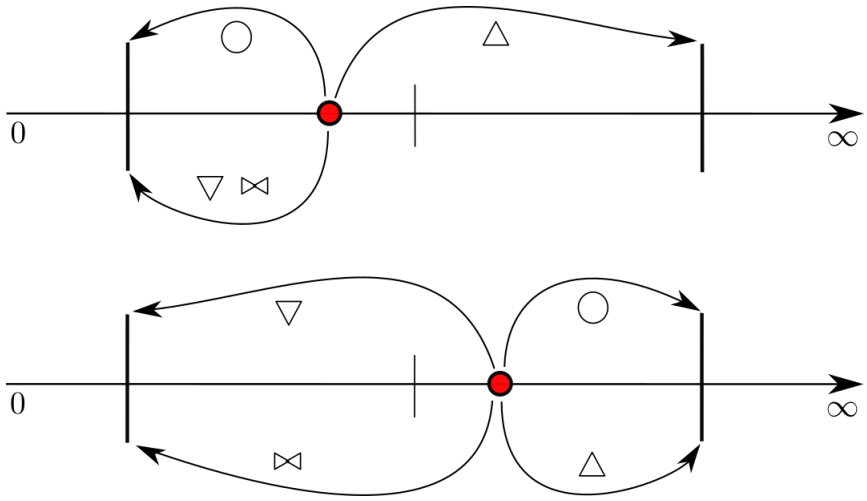


FIGURE 2.2 – Figure des 4 modes d'arrondi d'IEEE-754. Le cercle rouge représente le nombre à arrondir et les grandes barres noires verticales, les deux nombres flottants représentables les plus proches et la petite barre le milieu de l'intervalle qui les sépare.

- ○ **Arrondi au plus près** : Arrondi au nombre flottant représentable le plus proche. Si le nombre est au milieu de nombres flottants, la convention *TiesToEven* choisit celui dont la mantisse est paire.
- △ **Arrondi vers** $+\infty$: Arrondi vers le nombre flottant représentable supérieur ou égale.
- ▽ **Arrondi vers** $-\infty$: Arrondi vers le nombre flottant représentable inférieur ou égale.
- ⋈ **Arrondi vers 0** : Arrondi vers le nombre flottant le plus proche et de valeur absolue inférieure.

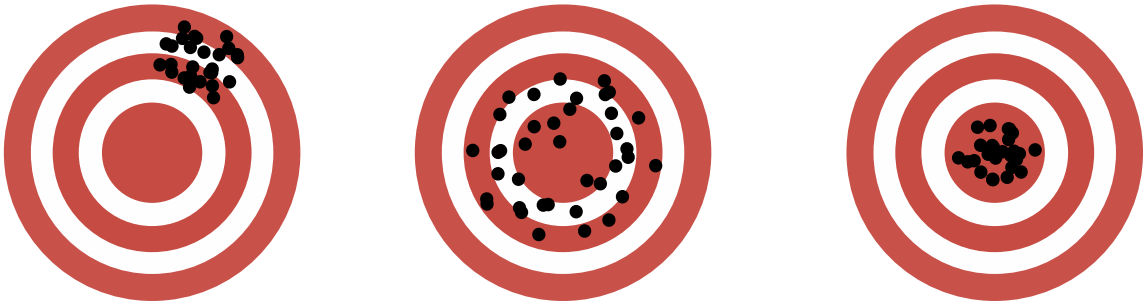
Dans la suite de cette thèse nous utiliserons, sauf indication contraire, l'arrondi au plus près noté $\circ(x)$. Si le contexte ne permet pas de les déduire, nous mentionnerons la précision ou la dynamique dans laquelle il est effectué. On considère pour la suite que la base est binaire $\beta = 2$ et adopterons la notation suivante $\circ_{d,p}(x)$ pour un arrondi en précision p sur une dynamique d.

2.2 Métrologie

Bien qualifier les propriétés d'une mesure est un art difficile. En effet dans la vie courante les notions de précision, de justesse ou d'exactitude sont souvent utilisées de manière interchangeable. Pour nous y aider, nous nous inspirerons de la métrologie qui est la science des mesurages. Nous allons définir dans cette section les différents termes utilisés tout au long de cette thèse pour caractériser les propriétés d'un calcul. Ces définitions sont extraites du *Vocabulaire international de métrologie* [95] et adaptées pour l'usage du calcul numérique.

Définition 2.2.1 (Valeur théorique ou *de référence*)**.** Valeur que l'on souhaite calculer.

Notée $x^\star$, elle fait référence à l'objet mathématique dont on souhaite calculer la valeur numérique comme par exemple la valeur de π .



Fidélité Justesse Exactitude

FIGURE 2.3 – Illustration des notions de fidélité, de justesse et d'exactitude dans le cadre du calcul numérique. Chaque point noir représente le résultat d'une exécution. Ces points se confondent si l'exécution est déterministe. L'exactitude est une propriété forte qui demande à la fois la fidélité – que les résultats empiriques ne soient pas trop éloignés les uns des autres – et la justesse – que la moyenne des résultats empiriques soit centrée sur le résultat théorique.

Définition 2.2.2 (Valeur empirique ou *approchée*)**.** Résultat issu d'une suite de calculs visant à approcher une valeur théorique.

C'est la valeur numérique, notée x, qui approche $x^\star$ par une suite de calculs. Tout l'enjeu du calcul numérique est d'avoir une approximation aussi proche que possible de la valeur théorique. Il existe en générale plusieurs méthodes pour approximer une valeur, chacune ayant des propriétés différentes (vitesse de convergence, stabilité numérique, ...). L'analyse de ces propriétés est du ressort de l'analyse numérique [89]. Par exemple, l'algorithme de Chudnovsky [29] offre une meilleure approximation de π car plus proche du résultat exact que la méthode de Monte-Carlo [130].

Définition 2.2.3 (Exactitude)**.** Étroitesse de l'accord entre une valeur empirique x et une valeur théorique $x^\star$.

L'exactitude mesure l'écart entre la valeur numérique calculée et la valeur théorique. On dira par exemple que $3,14159265359$ est plus exacte que $3,14$ dans l'approximation de π car plus proche.

Définition 2.2.4 (Justesse)**.** Étroitesse de l'accord entre la moyenne d'un nombre infini de valeurs calculées répétées et une valeur de référence.

Définition 2.2.5 (Fidélité)**.** Étroitesse de l'accord entre les valeurs calculées obtenues par des calculs répétés du même objet ou d'objets similaires dans des conditions spécifiées.

Pour bien différencier la justesse de la fidélité, prenons le cas du calcul de π par la méthode de Monte-Carlo (figure 2.4). Dans ce cas ci, le tirage aléatoire rend les résultats différents d'un calcul à l'autre. Deux éléments sont alors intéressant d'un point de vue numérique lorsque l'on fixe le nombre de points à tirer. Premièrement, que les résultats ne soient pas trop éloignés les uns des autres, en s'assurant par exemple que l'écart-type reste faible. Une faible dispersion traduit une bonne **fidélité** des résultats. Deuxièmement, que les résultats ne soient pas trop éloignés du résultat théorique. En effet, la série peut être de haute fidélité mais centrée sur une valeur très éloignée de la valeur de référence. Plus la moyenne est centrée sur le résultat théorique et plus le calcul est **juste**.

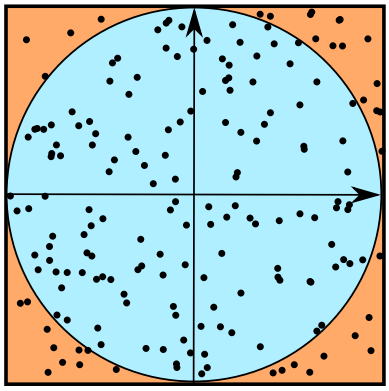


FIGURE 2.4 – Illustration de la méthode de Monte-Carlo pour le calcul de π. Dans le carré unité, la probabilité de tirer un point à l'intérieur du cercle est égale à la surface bleue c'est-à-dire π. En comptant le nombre de points à l'intérieur du cercle et ceux à l'extérieur, on peut estimer l'aire du cercle bleu et donc π.

Un calcul à la fois juste et fidèle a donc une bonne exactitude. La figure 2.4 montre qu'augmenter le nombre de points améliore la justesse (courbe bleu) et la fidélité (courbe rouge).

Définition 2.2.6 (Précision)**.** Nombre de chiffre après la virgule d'une valeur numérique.

Bien que souvent utilisé dans le langage courant, le terme précision ne fait cependant pas partie du langage métrologique. Dans son acception large, il fait référence à la résolution d'un appareil de mesure comme le *millimètre* pour une règle ou le *gramme* pour une balance. Nous utiliserons ce terme pour décrire le nombre de chiffres après la virgule d'une valeur numérique. Ainsi la valeur $x = 3,4324234$ sera considérée comme plus précise que $y = 3,14$ même si x est une approximation de π qui est moins exacte que celle de y.

Définition 2.2.7 (Erreur systématique)**.** Composante de l'erreur de calcul qui, dans des exécutions répétées, demeure constante ou varie de façon prévisible.

C'est l'erreur liée au modèle de représentation des nombres utilisés et à l'ordre dans lequel les calculs sont effectués. Cette erreur caractérise l'algorithme de calcul et est indépendante de l'environnement de calcul.

Définition 2.2.8 (Erreur aléatoire)**.** Composante de l'erreur de calcul qui, dans des exécutions répétées, varie de façon imprévisible.

C'est l'erreur liée à des phénomènes aléatoires, qu'ils soient volontaires (arithmétique stochastique) ou involontaire (exécution non-déterministe). Une erreur aléatoire peut être considérée comme systématique si une partie de l'environnement de calcul responsable de l'indéterminisme est fixée à l'avance.

Définition 2.2.9 (Vérification)**.** Fourniture de preuves tangibles qu'une entité donnée satisfait à des exigences spécifiées.

C'est l'assurance que la méthode numérique calcule de manière correcte les résultats attendus. Ces preuves sont souvent d'ordre mathématique et peuvent être plus ou moins rigoureuses.

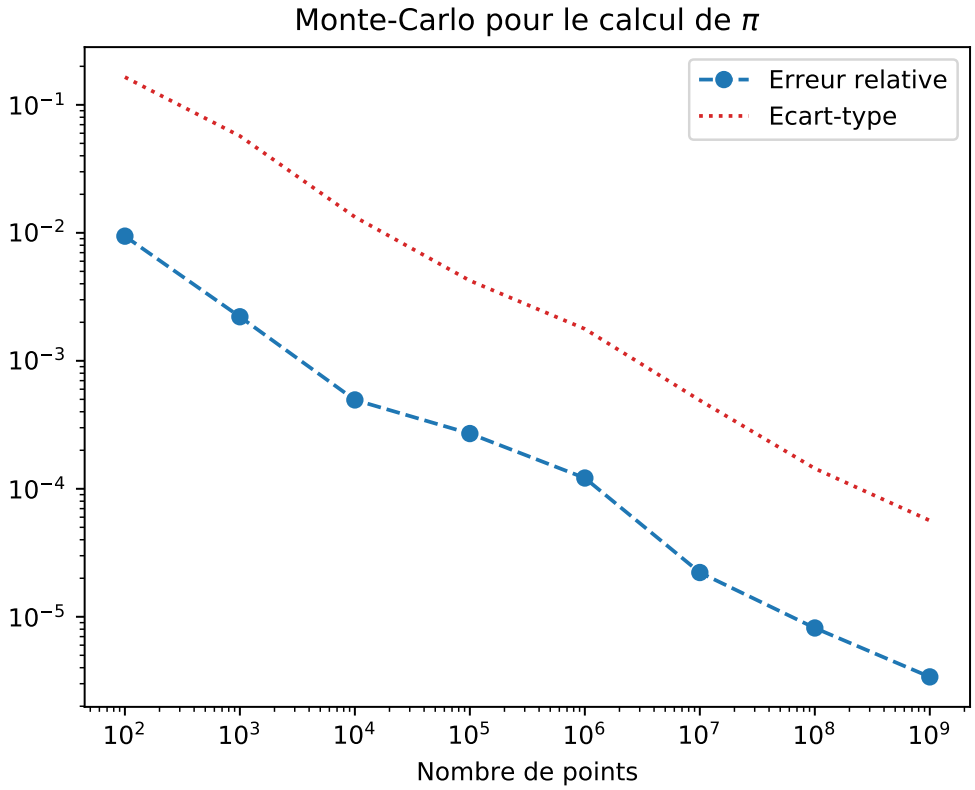


FIGURE 2.5 – Erreur relative de la moyenne par rapport à π et écart-type des résultats sur 30 exécution de Monte-Carlo en fonction du nombre de points tirés (en abscisse). Plus le nombre de point augmente et plus la dispersion des résultats ici représentée par l'écart-type diminue, ce qui traduit une amélioration de la fidélité. D'autre part, l'augmentation du nombre de point améliore également la justesse avec une décroissance de l'erreur relative ce qui signifie que la valeur moyenne empirique se rapproche de la valeur théorique.

Définition 2.2.10 (Validation)**.** Vérification que les exigences spécifiées soient adéquates à l'usage déterminé.

La validation arrive après la vérification pour assurer que la méthode numérique est utilisée dans le bon cadre mathématique. Par exemple, un certain nombre de preuves sur les nombres flottants reposent sur l'absence de dépassements de capacité qui sera présenté en section 2.4.2. La validation consiste ici à s'assurer que c'est bien le cas.

Définition 2.2.11 (Correction)**.** Compensation d'un effet systématique connu.

2.3 Formats et architectures

La section 2.1 a présenté le cadre théorique du modèle flottant et ses propriétés. Nous allons maintenant détailler les normes qui régissent le calcul flottant ainsi que les formats émergeant. Nous présenterons ensuite les différents types d'architectures utilisées dans le HPC.

2.3.1 Formats

Les formats pour calculer avec les réels sont nombreux. Du standard IEEE-754 au tout nouveau bfloat16 en passant par la concurrence du posits, l'histoire du nombre flottant montre que les standards et les pratiques évoluent. La période qui à précédé la norme IEEE-754 a vu fleurir bon nombre de formats flottants. En effet, chaque constructeur y allait de sa recette pour calculer avec les réels provoquant des problèmes d'exactitude d'une architecture à l'autre. On voit renaître aujourd'hui cette tendance avec la sortie de formats réduits dédiés aux calculs des réseaux de neurones pour augmenter toujours plus les performances.

Après une présentation de la norme IEEE-754, nous passerons en revu les nouveaux formats à précision réduite pour calculer plus vite, puis les formats inventés pour calculer plus juste.

La norme IEEE-754

La norme IEEE-754 [33] est la première norme de standardisation du calcul flottant sur machine et fait aujourd'hui encore autorité sur l'ensemble des calculateurs utilisés. Elle est née d'une volonté d'unifier les résultats des calculs flottants jusque là dépendants des constructeurs et variants d'une architecture à l'autre. En 2008, une refonte de la norme a été publiée [34] pour introduire de nouveaux formats et résoudre certaines ambiguïtés [134, p.79].

Les objectifs de cette norme IEEE-754 sont de rendre reproductibles et portables les calculs flottants entre architectures. Elle standardise les formats : quatre formats binaires (détaillés dans la table 2.1) et trois décimaux. Chaque format est défini par sa taille en bits ainsi que la taille allouée à l'exposant et la mantisse. La mantisse suit la convention du bit implicite. Deux formats optionnels de précision étendue ont été également introduits lors de la première version de la norme. Le problème est que leur description n'est pas formalisée mais est donnée sous forme de valeurs minimales à respecter (figure 2.2). Nous verrons en section 2.5.2 les problèmes que cela pose pour la reproductibilité des calculs.

La norme définit aussi des valeurs spéciales pour traiter les cas d'overflow ou de non-sens mathématique : deux infinis ($\pm\infty$) lorsque le nombre est trop grand pour être représenté dans le format et le NaN (*Not A Number*) qui traduit le résultat d'une opération illégale dans les réels comme $\sqrt{-1}$ ou $\frac{0}{0}$.

Nom	Exposant (bits)	Mantisse (bits)	e_{min}, e_{max}
binary16	5	10	(-14,+15)
binary32	8	23	(-126,+127)
binary64	11	52	(-1022,+1023)
binary128	15	112	(-16382,+16383)

TABLE 2.1 – Formats binaires de la norme IEEE-754. Le nom du format indique la base et sa taille. Ainsi le *binary16* est un format binaire codé sur 16 bits. Le signe est omis car représenté sur 1 bit quelque soit le format.

Nom	Précision	e_{min}	e_{max}
simple précision étendue	$\geqslant$ 32	$\leqslant$ -1022	$\geqslant$ 1023
double précision étendue	$\geqslant$ 64	$\leqslant$ -16382	$\geqslant$ 16383

TABLE 2.2 – Caractéristiques des formats en précision étendue de la norme IEEE-754.

Afin de rendre les calculs reproductibles, chaque opération élémentaire $\{+,-,\times,/,\sqrt{}\}$ doit retourner un arrondi correct : c'est-à-dire retourner le flottant le plus proche du résultat calculé avec une précision infinie dans le format choisi et arrondi au nombre flottant représentable le plus proche, suivant le mode d'arrondi choisi 2.2. La norme prévoit quatre arrondis [1]. Elle fournit également une liste des fonctions transcendantes [2] devant être correctement arrondies [134, p.103].

Les formats à taille réduite

Les formats de taille réduite ont vu le jour suite à l'utilisation croissante de la précision mixte afin d'augmenter la densité de calculs tout en réduisant le volume de données échangées. Les premiers travaux sur l'utilisation du binary16 par exemple ont connus un intérêt croissant car ce format permet d'accélérer les phases de calculs nécessitant peu de précision [83, 128, 38].

L'utilisation de format réduit est particulièrement étudiée dans l'apprentissage par réseaux de neurones. Les réseaux de neurones reposent sur une utilisation intensive du calcul de gradients qui généralisent la notion de dérivée aux fonctions à plusieurs variables. Or le calcul des gradients s'avère être en pratique dominé par les valeurs de petites magnitudes [128]. Une étape de renormalisation permet alors d'éviter la perte des petites valeurs et la divergence des calculs. Cette étape consiste à remettre les valeurs dans une plage dynamique plus restreinte en les divisant par la magnitude maximale. Même si la renormalisation nécessite un coût supplémentaire en termes de calculs, le gain en performance apporté par le binary16 sur le binary32 reste nettement supérieur. Cependant, cette étape peut être évitée en augmentant la dynamique du format.

C'est ainsi qu'ont émergés des formats alternatifs au binary16, comme le **Bfloat16** *16-bits floating-point* [117] qui est un format de calcul flottant sur 16 bits développé, par Google pour Tensorflow [1], une librairie de calcul dédiée à l'apprentissage par réseaux de neurones. Le bfloat16 avec 8 bits d'exposant dispose d'une dynamique plus grande, la même que celle du binary32. Plus de valeurs sans étapes de renormalisation sont représentables, ce qui diminue le temps de calcul et le trafic mémoire [101].

1. cinq en comptant les deux versions de l'arrondis au plus près.
2. fonction qui n'est pas solution d'une équation polynomiale à coefficients polynomiaux.

On assiste aujourd'hui à une floraison des formats où chaque constructeurs y va de sa recette : IBM avec le **DLFloat** qui est un format 16 bits avec 1 bit de signe, 6 bits d'exposant et 9 bits de mantisse. Selon ses auteurs, le DLFloat est plus efficace que le binary16 en ne gérant pas les nombre dénormalisés et en fusionnant les NaNs et infinis. Microsoft avec le **MSFP8-11** [30] dans le projet Brainwave propose un format sur 8 à 11 bits avec 1 bit de signe, 5 bits d'exposant et 2 à 5 bits de précision suivant les variantes.

Un autre format qui n'est pas basé sur le modèle IEEE-754 développé par Intel le **Flexpoint** [104], propose de ne garder qu'un seul exposant pour toutes les valeurs d'un tenseur qui sont une généralisation des vecteurs dans le cadre de l'algèbre multilinéaire. Dans ce format, les éléments du même tenseur partagent un exposant mais gardent une mantisse propre. Cette mantisse est stockée sous la forme d'un entier et les calculs sont effectués en virgule fixe. La taille du format est quant à elle variable.

Conçu pour des algorithmes itératifs dans le cadre du deep learning, l'exposant est dynamiquement mis-à-jours entre chaque itération. Il est prédit suivant la distribution des mantisses ainsi que la magnitude maximale pour éviter les cas d'overflow. Cette prédiction est précise si les applications testées ont une dynamique qui évolue lentement dans le temps. Elle est en revanche difficilement appréhendable dans le cas d'une dynamique évoluant rapidement.

L'avantage de cette représentation est qu'elle économise le stockage de l'exposant pour le tenseur entier. L'information stockée est uniquement fonction de la précision choisie, permettant d'augmenter la dynamique gratuitement et de réduire le volume de données à échanger. De plus, les calculs entiers sont plus efficaces énergiquement que les calculs flottants, permettant de limiter la consommation électrique. Enfin, contrairement à l'arithmétique en virgule fixe classique, la mise-à-jour de l'exposant est laissée à la machine, ôtant au programmeur la charge de les maintenir.

Les formats alternatifs

Nous avons vu en section 2.4 que le format flottant était sujet à des erreurs parfois difficiles à estimer pour des codes de calculs importants. La *significance arithmetic* [72] propose d'y réponde en embarquant une information sur le nombre de chiffres significatifs du calcul ce qui permet de traquer la significance au cours des calculs. Le format **FP-ANR** *Floating Point Adaptive Noise Reduction* [50] se place dans cette lignée en utilisant la place du dernier 1 dans la mantisse comme dernier bit significatif. L'objectif de FP-ANR est de détecter les annulations catastrophiques lors de l'exécution.

Une autre variante de la *significance arithmetic* est apportée avec les **unum** [82] (*universal numbers*), un format de calcul à taille variable décliné en trois types (Type I, Type II), puis dans une version différente avec le type III avec les *posits* et les *valids*. Si les premiers travaux sur les unums (Type I, Type II) peuvent être vus comme un format de calcul exact intégrant l'arithmétique d'intervalles, les *posits* se rapprochent plus du format IEEE-754 classique avec quatre champs : 1 bit de signe, r bits de régime pouvant être vus comme une base, e_s bits d'exposants et f_s bits de mantisse. La taille du posit et de l'exposant déterminent son format, la norme posit [81] (encore à l'état préliminaire) en définit quatre (8, 16, 32 et 64 bits). On ne détaillera pas comment reconstruire un nombre à partir de son encodage posit mais il est moins aisé qu'en arithmétique flottante IEEE-754.

Si, de l'aveu de son auteur, le format posit va, à terme, remplacer la norme IEEE-754, De Dinechin et al. [44] dressent les avantages et les inconvénients des deux représentations et préconisent une utilisation des deux formats suivant les applications visées. Si les posits s'avèrent mieux disposés

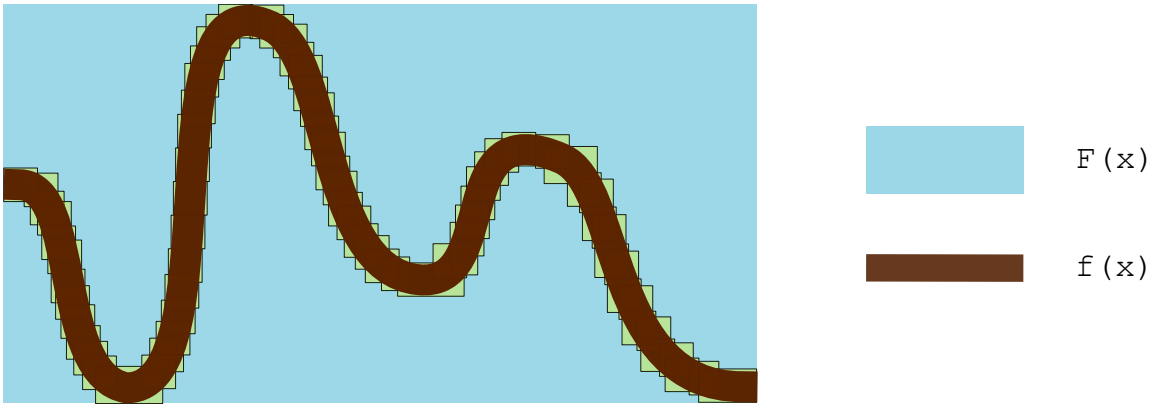


FIGURE 2.6 – Illustration de l'*effet enveloppant* de l'arithmétique d'intervalles. L'enveloppe $F(x)$ est un encadrement très large de $f(x)$ qui peut cependant être raffiné en utilisant des unions d'intervalles (rectangles jaunes).

au *deep learning* grâce à une dynamique plus large, l'erreur de représentation est plus grande comparé à un format flottant de même taille. Enfin l'étape cruciale pour l'essor des posits est le cadre théorique d'analyse d'erreurs, essentiellement construit autour des flottants IEEE-754.

L'**arithmétique d'intervalles** [132] utilise des intervalles pour encadrer le résultat des calculs flottants en assurant que le résultat théorique se situe toujours à l'intérieure de l'intervalle. Cela permet de manipuler des valeurs non représentables en machine comme π ou d'encadrer des incertitudes.

L'arithmétique d'intervalles étend les opérations arithmétiques usuelles $\{+,-,\times,/\}$ aux intervalles ainsi que les opérations algébriques telles que $\sqrt{}$, ln ou exp. Par exemple l'addition devient pour $x \in [\underline{x}, \overline{x}], y \in [\underline{y}, \overline{y}], x + y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. Pour plus de détails, voir [153].

L'arithmétique par intervalles est un outil puissant pour garantir un résultat comprenant la solution exacte. Cependant, son application est coûteuse en pratique puisqu'il faut maintenir deux valeurs pour chaque nombre flottant ce qui, en plus d'augmenter la charge de calcul, peut détruire les optimisations faites par le compilateur. Enfin, d'un point de vue plus théorique, le problème d'*effet enveloppant* conduit certains calculs par intervalles à des surestimations. De nombreuse extensions ont été proposées au modèle original avec notamment l'arithmétique affine [46] ou les modèles de Taylor [136].

2.3.2 Les architectures et le calcul flottant

Le calcul hétérogène [102], consistant à tirer parti des forces et des faiblesses des différentes architectures pour améliorer les performances générales d'une application (temps, mémoire, énergie), est présent dans le HPC depuis quelques années [102]. Si celui-ci a été dominé par l'utilisation intensive du CPU, il n'est cependant pas la solution optimale pour tous les types de problèmes [129]. L'utilisation de coprocesseurs venant appuyer le CPU classique pour des taches spécifiques est un modèle qui connaît un succès grandissant.

CPU

Le Central Processing Unit (CPU) est l'unité de calcul la plus répandue dans les machines. C'est une architecture généraliste visant à exécuter diverses taches séquentielles. Il reste l'unité principale pour développer des applications car il est très bien adapté à débiter des instructions diverses à la suite grâce au pipeline superscalaire. De plus, il gère efficacement les instructions logiques de branchements par exemple. Depuis la norme IEEE-754 la plupart des CPU implémentent les formats binary32 et binary64.

GPU

Le Graphics Processing Unit (GPU) est une unité de calcul massivement parallèle reposant sur le paradigme Single Instruction Multiple Data (SIMD). Au départ utilisé pour le rendu graphique, son utilisation a été détourné au profit du calcul pur. On parle aussi de General Purpose Graphics Processing Unit (GPGPU). On le retrouve aujourd'hui dans la plupart des infrastructures HPC en tant que coprocesseur.

En effet, la structure du GPU se prête particulièrement bien au calcul matriciel : elle est composée d'une grille de micro-unités de calcul réalisant chacune une même tache simple en parallèle (comme une addition) mais sur des données différentes.

L'arithmétique flottante utilisée sur les premiers GPUs n'était pas conforme aux standard IEEE-754 [182, 49]. Dans un souci de reproductibilité, les constructeurs ont renforcé leur compatibilité avec la norme IEEE-754 en intégrant des formats standards (binary16, binary32, binary64), plusieurs modes d'arrondi (seul le mode d'arrondi au plus près était disponible auparavant) et la gestion correcte des dénormalisés.

ASIC

Les Application-Specific Integrated Circuits (ASICs) [126] sont des unités de calcul dédiés au traitement d'une application en particulier comme le traitement du signal (*Digitial Signal Processor* [165]), contrairement aux processeurs plus généralistes de type CPU.

Pour une même application, les ASICs sont plus performants et ont une consommation énergétique plus faible que les processeurs généralistes. Leur conception privilégie des unités de calculs simples organisées de manière à optimiser la surface de transistors. L'intégration des composantes de calcul, de mémoire et des interconnections directement sur la puce rend ce type d'architecture efficace. L'inconvénient majeur est le manque de souplesse et le coût de conception puisqu'elles sont par définition conçues pour la résolution d'une tâche en particulier.

Les calculs flottants au sein des ASICs sont marqués par une forte tendance aux formats sur mesure, notamment dans le domaine du *deep learning*. On observe que chaque constructeur calibre son format pour répondre à ses besoins applicatifs. L'usage de formats à précision réduite décuple les performances en densifiant les calculs et réduisant le trafic mémoire. Parmi ces nouveaux coprocesseurs, on pourra citer le Tensor Processor Unit (TPU) [98], conçu par Google pour accélérer la phase d'inférence, qui fait un usage intensif du bfloat16 comme le processeur Nervana d'Intel [**?**].

Pour bien comprendre pourquoi le bfloat16 accroît les performances, regardons l'architecture d'un TPU de plus près. Le TPU repose sur une architecture systolique [107] de Multiply ACcumulators (MACs) bfloat16 qui découple l'accès aux données des calculs (*Decoupled Access-Execute - DEA* [164]). Une architecture systolique est un réseau de petites unités de calcul interconnectées qui s'échangent de l'information localement. La topologie des réseaux est alors construite en fonction

du type de problème à résoudre. Le premier avantage est d'augmenter la densité de calculs (plus de calculs sur moins de silicium). Le deuxième est de réduire l'empreinte énergétique en utilisant des unités de calculs plus simples. Dans le cas du TPU, les unités sont des MACs qui réalisent l'opération $x = x + y.z$. Enfin, le DEA permet de charger les données dont auront besoins les MACs pendant qu'ils calculent. En réduisant la taille du format flottant, on accroît de fait le débit de calculs.

FPGA

Les FPGAs sont des circuits intégrés reprogrammables. Contrairement aux processeurs classiques, les FPGAs sont programmés pour effectuer un calcul et non exécuter une suite d'instructions. Cette reconfigurabilité permet de prototyper rapidement des circuits de calcul. Les FPGAs reposent sur l'assemblage de blocs logiques sur une matrice de routage. L'assemblage des blocs logiques se fait par l'intermédiaire de langages de programmations, dont VHDL [135] et Verilog [173] sont les plus répandus. Chacun des blocs logiques est indépendant des autres et ils sont tous cadencés par un tic d'horloge en entrée. Cette indépendance offre un potentiel de parallélisation important et fait du FPGA une architecture privilégiée pour la résolution de tâches spécifiques [7, 88] à l'instar de l'ASIC.

Si le FPGA partage des similarités avec l'ASIC, il n'a cependant pas les mêmes performances. De plus, étant donné qu'il n'utilise que des éléments logiques simples (portes logiques, multiplexeurs, ...) et n'a pas d'unité dédiées au calcul flottant par exemple, la conception des calculs sur FPGA est plus difficile [57, 2, 177, 174] que sur d'autres architectures type CPU. Le FPGA présente alors des avantages sur l'implémentation de fonctions plus complexes telles que les fonctions trigonométriques [58]. Le FPGA permet également de travailler sur des précisions variables [172] à un grain très fin ce qui laisse un large boulevard d'optimisation.

2.4 Notions d'erreurs

Le modèle flottant est une discrétisation des nombres réels. Par conséquent, des erreurs sont commises lors de la traduction en machine mais aussi lors des calculs. Nous allons présenter différentes manières de quantifier l'erreur puis nous verrons plusieurs typologies d'erreurs liées au modèle flottant.

2.4.1 Quantification de l'erreur

Nous présentons ici trois mesures d'erreurs utiles pour quantifier des erreurs d'exactitude.

Définition 2.4.1 (Erreur absolue)**.** L'erreur absolue est définie comme la distance entre une valeur de référence x^* et une valeur x.

$$Erreur_{absolue} = x^* - x.$$

Définition 2.4.2 (Erreur relative)**.** L'erreur relative est définie comme la distance entre une valeur de référence $x^* \neq 0$ et une valeur x, normalisée par la valeur de référence.

$$Erreur_{relative} = \frac{x^* - x}{x^*}.$$

Exemple : Soit $\hat{h} = \circ_{p=3}(h) = 6,62 \times 10^{-34}$ une approximation de h. L'$Erreur_{absolue} = |h - \hat{h}| = 6.07... \times 10^{-37}$ mesure la distance qui sépare h de son approximation flottante. L'$Erreur_{relative} = |h - \hat{h}|/|h| = 9.16... \times 10^{-4}$ nous indique que les valeurs ont à peu près 3 chiffres en commun. De manière plus rigoureuse, si l'on prend l'opposé du $\log_{10}$ de l'erreur relative, on peut dire que ces deux valeurs ont 3,0380 chiffres communs. Ces notions de chiffres communs seront abordées plus longuement en section 2.6.

L'Unit in the Last Place (ulp) définit l'unité d'erreur minimale d'un format flottant donné. Plusieurs définitions existent dans la littérature [100, 71, 85, 37]. Nous retenons la définition 2.4.3 extraite de [134, p.33].

Définition 2.4.3 (Définition de Goldberg étendue aux nombres réels)**.** Si x est un nombre flottant tel que $x \in [\beta^e, \beta^{e+1}[$ alors

$$ulp(x) = \beta^{max(e,e_{min})-p+1}.$$

L'ulp permet de quantifier la distance minimale entre deux nombres représentables successifs dans un format flottant donné. Par exemple, soit un format avec $p = 3$, $\beta = 10$ et $e_{min} = -10$. Soit $x = 3,14$ un nombre flottant du format. Alors $ulp(x) = 10^{max(0,-10)-3+1} = 10^{-2}$. Le nombre représentable suivant x est $x + ulp(x) = 3,14 + 10^{-2} = 3,15$.

Cette opération permettant de trouver le nombre représentable suivant n'est cependant plus valable lorsque x est une puissance de la base. En effet, la distance qui sépare x de son prédécesseur n'est pas la même que celle qui le sépare de son successeur.

Enfin une métrique utile dans la mesure de l'exactitude est celle du nombre de chiffres significatifs. L'idée est de compter le nombre de chiffres que deux valeurs ont en commun. Parker propose deux définitions [145, p.19] du nombre de chiffres significatifs, dont une liée à l'erreur relative et l'autre à l'erreur absolue.

Définition 2.4.4 (Nombre de chiffres significatifs relatifs)**.** Le nombre de chiffres s en base β qui diffère entre une valeur réelle x et son approximation $\tilde{x}$:

$$s = -\log_\beta \left| \frac{\tilde{x} - x}{x} \right|.$$

Définition 2.4.5 (Nombre de chiffres significatifs absolus)**.** Le nombre de chiffres s en base β qui diffère entre une valeur réelle x et son approximation $\tilde{x}$:

$$s = -\log_\beta \left(2 \left| \frac{\tilde{x} - x}{\beta^{\lfloor \log_\beta |x| \rfloor + 1}} \right| \right).$$

Ces deux définitions donnent des s différents mais qui diffèrent au plus de 1. La démonstration est donnée [145, p.20].

2.4.2 Les erreurs du modèle flottant

Le modèle flottant, s'il est une approximation des nombres réels, n'en garde néanmoins pas les mêmes propriétés. Nous allons passer en revue les différentes erreurs qu'il introduit que nous illustrerons d'exemples.

Erreur de représentation

C'est l'erreur liée à la représentation d'un nombre réel dans un format flottant. Puisque la machine dispose d'une mémoire finie, la représentation numérique des nombres irrationnels est par conséquent impossible. Pour exemple π dont la précision nous est connue avec 31415926535897 chiffres. Ce dernier record établi par Google [151] a nécessité 121 jours de calculs sur plusieurs serveurs. Si une telle précision est un exploit elle est inutile pour les calculs courants. Cependant, le développement décimal de π n'en reste pas moins infini et nécessite un arrondi quelque soit la précision finie voulue.

D'autre part, les constantes physiques sont connues avec une certaine précision. Si la précision dont dispose le format flottant est inférieure, un arrondi est nécessaire pour représenter ce nombre.

Enfin, un nombre peut ne pas avoir un développement fini dans une base. Par exemple, $0,1$ représentable en base décimale ne l'est pas en base binaire :

$$0,1_{10} = 0.0001100110011001100 1...{}_2.$$

Erreur d'absorption

Le résultat d'un calcul flottant est lui aussi sujet à des erreurs. Pour l'addition flottante, le phénomène d'absorption est lié à un grand écart d'amplitude. En effet, si les deux opérandes n'ont pas le même exposant, une partie des chiffres du nombre avec plus petite magnitude sera perdue. Ce problème se comprend mieux visuellement. Prenons la somme de $3,14 \oplus 3,14 \times 10^{-2}$ avec $\circ_{p=3}$:

$$\begin{array}{rr} & 3,14 \\ + & 0,0314 \\ \hline \circ_3 & 3,1714 \\ \hline & 3,17 \end{array}$$

L'erreur relative entre le résultat en précision infinie et celui après arrondi est au maximum de $\sim 9.9 \times 10^{-3}$, ce qui correspond à l'écart entre les magnitudes des nombres. Du fait de cette erreur, la somme flottante n'est pas associative. Ainsi, l'ordre dans lequel sont sommés les éléments d'un tableau peut changer le résultat. On verra en section 4.10.3 qu'il existe des algorithmes pour récupérer une partie de l'information perdue, notamment via les méthodes de compensation.

Erreur d'annulation

Les erreurs d'annulation ou *cancellation* apparaissent lors de la soustraction de nombres proches. Ce phénomène provoque une disparition de l'information contenue dans la mantisse. Les chiffres de poids faible sujets aux erreurs d'arrondi sont alors remontés au niveau des chiffres de poids fort. Prenons le cas du calcul de $b^2 - 4ac$ avec $a = 1,14$, $b = 3,99$ et $c = 3,48$ dans un format $p = 3$.

$$\begin{array}{lll} \circ(b^2) & = \circ(1,5920999... \times 10^1) & = 1,59 \\ \circ(4ac) & = \circ(1,5868799... \times 10^1) & = 1,59 \\ & \circ(b^2) - \circ(4ac) & = 0 \end{array}$$

Or la valeur exacte est $5,1299... \times 10^{-2}$. On voit donc que toute l'information pertinente a été perdue et que le résultat calculé est faux.

Erreur de dépassement de capacité

Les erreurs de dépassement surviennent lorsque le nombre réel à représenter est plus petit ou plus grand que les valeurs limites du format. Le dépassement de capacité est dirigé soit vers $\pm\infty$, soit vers 0 :

- **Vers** $\pm\infty$: Le valeur absolue du résultat est supérieure au plus grand flottant représentable. On parle également d'*overflow*.
- **Vers** 0 : Le résultat du calcul n'est pas représentable par un nombre normalisé ou dénormalisé. On parle également d'*underflow*.

2.5 La reproductibilité des calculs

Les efforts de la norme IEEE-754 pour standardiser les formats et uniformiser les calculs à travers les architectures ont été une avancée décisive pour la reproductibilité. Cependant, faire tourner une application sur un calculateur HPC nécessite d'empiler plusieurs couches logicielles et matérielles. Chacune de ces couches n'a pas forcément un comportement déterministe, ce qui est susceptible de rendre les résultats différents d'une machine à l'autre. Nous allons détailler ces différentes couches et les phénomènes non-déterministes qu'elles peuvent introduire sur la reproductibilité des calculs.

2.5.1 La non-associativité

Le déterminisme des opérations élémentaires d'IEEE-754-754 garantie une reproductibilité des opérations élémentaires à travers les architectures respectant cette norme. Cependant, l'addition et la multiplication flottante ne sont pas associatives en raison des erreurs d'absorptions. Le parenthèsage d'une somme importe comme le montre l'exemple suivant ($\beta = 10, p = 3$) :

$$\begin{aligned} &a = 1, b = 10^{10}, c = -10^{10} \\ &(a \oplus b) \oplus c = (1 + 10^{10}) - 10^{10} = 0 \\ &a \oplus (b \oplus c) = 1 + (10^{10} - 10^{10}) = 1 \end{aligned}$$

Autre exemple sur la distributivité de la multiplication sur l'addition dont le résultat exacte est $40,064$:

$$\begin{aligned} a = 12,8, b = 1,64, c = 1,49 & \\ a \otimes (b \oplus c) = 12,8 \otimes 3,13 & = 40,1 \\ (a \otimes b) \oplus (a \otimes c) = (12,8 \otimes 1,64) + (12,8 \otimes 1,49) & \\ = 20,1 \oplus 19,1 & = 39,2 \end{aligned}$$

Le problème de non-associativité peut se régler à l'aide d'un parenthèsage adéquat pour minimiser l'erreur d'exactitude lorsque les valeurs calculées sont connues a priori. En pratique, on ne connaît pas ces valeurs. Lors de la traduction du langage source vers le langage machine, le compilateur peut procéder à des optimisations qui peuvent casser la structure du calcul.

2.5.2 Les optimisations du compilateur

Après avoir traduit le langage source en langage intermédiaire, le compilateur procède à plusieurs passes d'optimisation pour rendre le code plus performant. Ces optimisations sont plus ou moins agressives, c'est-à-dire que la traduction s'éloigne plus ou moins de la sémantique du programme source.

Plusieurs optimisations pour le calcul flottant sont disponibles dans la plupart des compilateurs (gcc, icc, clang). Parmi ces optimisations on peut citer `-fassociative-math`, qui considère l'arithmétique flottante associative, ce qui n'est pas le cas (cf. 2.5.1). `-freciprocal-math`, qui autorise le compilateur à remplacer une division par une multiplication $x/y- > x*(1/y)$, une division coûtant moins de cycles de calcul qu'une multiplication. Les deux expressions n'ont cependant pas la même justesse.

La réécriture algébrique est aussi source de mauvaise traduction. L'exemple de la somme de Kahan [99] est une bonne illustration. Dans cette exemple, deux versions de la somme sont compilées, l'une sans drapeaux d'optimisation et l'autre avec des optimisations agressives. Le compilateur reconnaît que le corps de la boucle est équivalent à `sum = sum + f[i]`. Les calculs intermédiaires, servant à corriger des erreurs d'absorption, sont supprimés ce qui conduit au calcul d'une somme naïve. Nous verrons en détail le code généré en section 3.1.3.

Bien que les opérations élémentaires d'IEEE-754 soient reproductibles, l'inlining peut conduire à des résultats différents pour une même opération. Cette technique consiste à remplacer un appel de fonction par le corps de la fonction, ce qui permet d'augmenter les performances pour des fonctions avec peu d'instructions. Monniaux [131] donne l'exemple suivant avec l'architecture IA-32 [94] sur la figure 2.8.

La valeur affichée par le programme est soit ∞ si `f` est inlinée soit 10^{308} si `f` ne l'est pas. La raison est que la convention d'appel retourne la valeur dans un registre `x87` qui est un registre en précision double étendue sur 80 bits. Les calculs intermédiaires qui produisent un overflow en binary64 sont alors représentables en précision étendue.

Enfin dans le même esprit que l'inlining, la propagation de constante peut elle aussi tromper le programmeur. Cette technique consiste à remplacer chaque occurrence d'une variable constante par sa valeur et ainsi ne la calculer qu'une seule fois. Le problème est que la valeur calculée par le compilateur peut différer de celle calculée durant l'exécution, les deux n'utilisant pas les mêmes implémentations. D'autre part, le remplacement d'une valeur dans une instruction FMA peut introduire un arrondi intermédiaire ce qui ne donne pas le même résultat :

$$\circ(a + b \times c) \neq \circ(a + \circ(b \times c)).$$

L'exemple de la figure 2.9 issu de [158] illustre le problème de la propagation de constante. Une analyse du code C montre que le résultat retourné est 0. En effet, la valeur de la variable *zero* ligne 2 est 0 de par la division entière. Les valeurs de `sin(π + zero)` et `sin(π)` sont par conséquent égales et la ligne 3 retourne 0. Cependant, le compilateur n'arrive pas à inférer que la variable *zero* est invariablement 0 et ne propage pas la constante. Si `sin(π)` et `sin(π + zero)` sont équivalentes, la première est évaluée au moment de la compilation tandis que l'autre l'est au moment de l'exécution. La différence d'implémentation entre les deux sinus conduit à des résultats différents sur l'évaluation de π et la fonction retourne 3.30261141×10^5.

(a)

```
1  KahanSum(const float f[], int N)
2   float sum = f[0];
3   float c = 0.0, y, t;
4   int i;
5
6   for (i = 1; i < N; i++) {
7    y = f[i] - c;
8    t = sum + y;
9    c = (t - sum) - y;
10   sum = t;
11  }
12
13  return sum;
14 }
15
```

(b)

```
KahanSum(const float f[], int N)
 float sum = f[0];
 float c = 0.0, y, t;
 int i;

 for (i = 1; i < N; i++) {

  t = sum + (f[i] - c);
  c = (t - sum) - (f[i] - c);
  sum = t;
 }

 return sum;
}
```

(c)

```
1  KahanSum(const float f[], int N)
2   float sum = f[0];
3   float c = 0.0, y, t;
4   int i;
5
6   for (i = 1; i < N; i++) {
7
8   t = sum + (f[i] - c);
9   c = (sum+(f[i]-c)-sum)-(f[i]-c);
10  sum = sum + (f[i] - c);
11  }
12
13  return sum;
14 }
15
```

(d)

```
KahanSum(const float f[], int N)
 float sum = f[0];
 float c = 0.0, y, t;
 int i;

 for (i = 1; i < N; i++) {



  sum = sum + f[i];
 }

 return sum;
}
```

FIGURE 2.7 – Effets des optimisations sur la somme de Kahan (a) conduisant à l'expression d'une somme naïve (d). Premièrement, la variable `y` est propagée en ligne 6 et 7 comme en figure (b). Puis c'est la variable `t` qui est propagée en ligne 9 et 10 comme en figure (c). En considérant les opérations flottantes associatives (`-fassociative-math`), le compilateur détermine par réécriture algébrique que l'expression de la variable `c` est égale à 0. Enfin, le compilateur supprime `c` et aboutit à la somme naïve en figure (d).

```c
static inline double f(double x) {
  return x/1E308;
}

double square(double x) {
  double y = x*x;
  return y;
}

int main(void) {
  printf("%g\n", f(square(1E308)));
}
```

FIGURE 2.8 – Code illustrant l'impact de l'inlining sur la qualité numérique. La valeur retournée dépend de si la fonction `f` est inlinée ou pas. La raison est que la valeur est retournée dans un registre en précision double étendue lors de l'inlining ce qui ne produit pas d'overflow comme en binary64. Code exécuté sur une architecture IA-32.

```c
double sinit() {
  double zero = (rand()%10)/99;
  return sin(π + zero)/sin(π) - 1;
}

```

FIGURE 2.9 – Illustration du problème d'évaluation statique contre dynamique. Les deux sinus sont évalués sur 0 mais l'un des deux l'est à la compilation tandis que l'autre l'est durant l'exécution. Les différences d'implémentations changent le comportement attendu et le programme ne retourne pas 0.

```
1  float Sum
2  (const float A[ ], int n ) {
3
4    float sum=0;
5    for (int i=0; i<n; i++) {
6      sum = sum + A[i];
7
8
9
10   }
11
12
13
14
15   return sum;
16 }
17
```

```
1  float Sum
2  ( const float A[ ], int n ) {
3    int i, n4 = n-n%4;
4    float sum=0,sum1=0,sum2=0,sum3=0;
5    for (i=0; i<n4; i+=4) {
6      sum = sum + A[i];
7      sum1 = sum1 + A[i+1];
8      sum2 = sum2 + A[i+2];
9      sum3 = sum3 + A[i+3];
10   }
11   sum = sum + sum1 + sum2 + sum3;
12   for (i=n4; i<n; i++) {
13     sum = sum + A[i];
14   }
15   return sum;
16 }
17
```

FIGURE 2.10 – À droite la version vectorisable du code de gauche. La vectorisation modifie l'ordre de sommation ce qui change le résultat car les nombres flottants ne sont pas associatifs. L'ordre de sommation compte.

2.5.3 Le parallélisme

Le parallélisme subdivise la charge de travail entre plusieurs travailleurs. Suivant les paradigmes utilisés, la façon dont est découpé le travail et l'ordre dans lequel seront exécutés les calculs diffèrent. Ce degré d'indéterminisme introduit un problème sur la reproductibilité des résultats. Nous allons présenter différents problèmes de la parallélisation qui introduisent chacun à leur manière de la non-reproductibilité.

La vectorisation

La vectorisation est une parallélisation de type SIMD qui calcule en parallèle une même opération sur plusieurs données. Là où une instruction scalaire (par opposition à vectorielle) ne calcule qu'une opération à la fois, l'instruction vectorielle peut calculer sur un vecteur de données en une seule étape. La taille des vecteurs indique le nombre d'opérandes qui peuvent être contenus dedans. Les constructeurs rivalisent d'ingéniosité pour fournir des vecteurs toujours plus grands [**?** 169].

La parallélisation est utilisée dans le cas des réductions comme une somme ou un produit scalaire. Le compilateur découpe le tableau en sous tableaux pour calculer des sommes partielles puis reconstruit le résultat à partir de ces sommes partielles. Le problème est que les sommes partielles ne sont pas nécessairement calculées dans le même ordre que dans le cas séquentiel. La non-associativité étant, les résultats seront différents. Le caractère non-déterministe est dû au fait que le nouvel ordre introduit est choisi par le compilateur. Ce choix reste néanmoins accessible au programmeur par le biais des diagnostiques de vectorisation (`-ftree-vectorizer-verbose` sous `gcc`).

Un exemple extrait de [36] est donné sur la figure 2.10. On voit sur la figure de droite que le découpage est fait en entrelaçant les valeurs du tableau `A`. Chaque somme partielle bénéficie de la vectorisation mais l'ordre des additions n'est plus le même que sur la figure de gauche.

Le partitionnement des données

La modélisation de phénomènes physiques repose la plupart du temps sur la discrétisation d'espaces physiques en unités de surfaces ou de volumes dans le cas des modélisations par éléments ou volumes finis. Ces unités prises dans leur ensemble constituent un maillage de l'espace modélisé. Une étape nécessaire dans le processus de parallélisation est le découpage de ce maillage en groupes ou domaines qui seront distribués aux différents travailleurs.

Le partitionnement du maillage est une étape clef pour obtenir des performances optimales [160]. Schématiquement, on découpe le maillage en autant de travailleurs. Le découpage est donc différent suivant le nombre de travailleurs. Plusieurs techniques existent suivant les besoins [162]. Un partitionnement qui varie suivant le nombre de travailleurs utilisés rend l'ordre des calculs flottants potentiellement non-déterministe. Or comme les flottants ne sont pas associatifs, cela impacte la fidélité des résultats (cf. 2.4).

L'analyse du code openTelemac [111] révèle ce problème aux frontières des domaines. En effet, dans la version parallèle de ce code de dynamique des fluides, un produit scalaire intervenant à la fin de chaque étape de calcul est réparti sur les différents domaines. Ainsi, le produit scalaire ($\sum_{i \in N} x_i.y_i$) n'est plus effectué dans le même ordre suivant le nombre de travailleurs utilisés. De plus cet ordre dépend de la librairie choisie. En effet, ce code utilise MPI (*Message Passing Interface* [79]) un protocole de calcul à mémoire distribuée dont plusieurs implémentations existent [80, 69]. Dans notre cas, c'est la fonction `mpi_reduce` qui regroupe les sommes partielles suivant un arbre de réduction. Or la norme MPI "encourage fortement" [65, p.175] la reproductibilité de cet opérateur mais ne l'impose pas.

L'ordonnancement du travail

Le travail peut être reparti de deux manières : soit statiquement lors de l'implémentation, en attribuant un travail spécifique à chaque travailleur, soit dynamiquement en déléguant la répartition à un ordonnanceur de tâches (*task scheduler*). Si l'allocation statique permet d'avoir un contrôle fin sur la gestion du travail, l'ordonnancement de tâches d'une application parallèle est un problème NP-complet [178]. C'est pourquoi les travaux s'orientent vers des allocations dynamiques pour lesquelles de nombreuses heuristiques existent [12] afin d'optimiser le placement.

Cependant, la non-associativité des flottants reste un problème. En effet, dans le cas de l'ordonnancement dynamique, l'utilisateur n'a pas de contrôle sur l'ordre des opérations. La programmation asynchrone laisse l'ordre des tâches libre, sans points de synchronisation explicites. Ainsi, comme dans le cas de la vectorisation (cf. 2.10), la somme d'un tableau devient également assujettie aux problèmes de fidélité dans le cas d'une parallélisation par tâche. La figure 2.11 illustre ce principe qui vaut pour toutes les approches de type folding en parallèle comme MapReduce [48] ou les réductions OpenMP [23].

Le problème de non-associativité se pose également dans le cas d'une allocation statique. En effet, le graphe d'allocation est très dépendant de la machine ciblée pour atteindre les meilleurs performances. Or celui-ci détermine l'ordre des tâches et donc des opérations flottantes. Le portage d'un code de calcul sur un nouveau supercalculateur apporte donc des problèmes de fidélité.

La question de l'aléatoire

L'indéterminisme des résultats n'est pas seulement lié à la non-associativité des opérations flottantes. L'objet physique modélisé peut lui aussi contenir une part d'aléatoire, que ce soit le

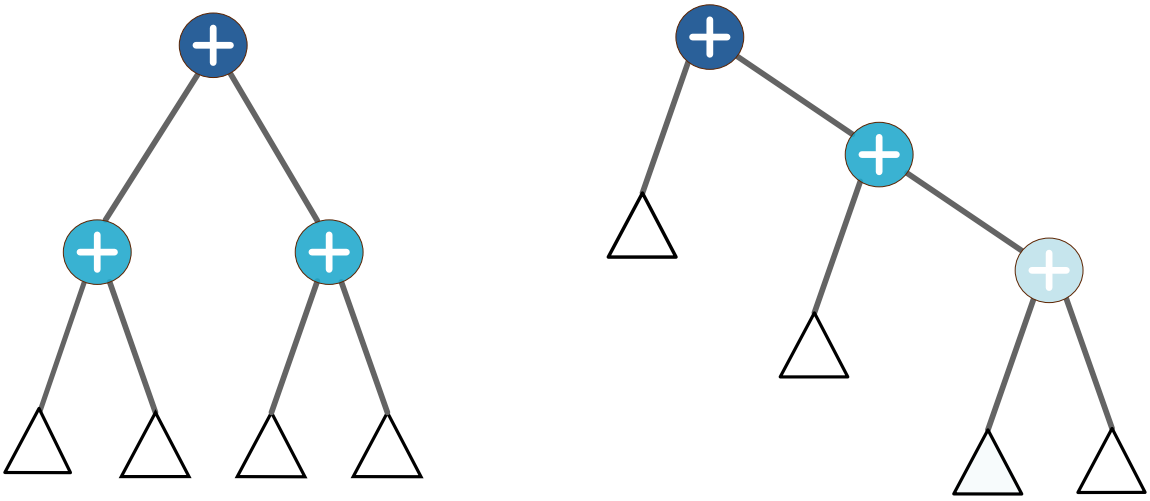


FIGURE 2.11 – Deux versions du folding dans le cas d'une sommation, avec à droite, la réduction en arbre et à gauche, la réduction linéaire. Les triangles représentent le résultat de calculs parallèles précédents (pouvant eux mêmes être issus d'un folding), les noeuds de mêmes couleurs étant indépendants. On voit que le choix de réduction implique un ordre des calculs différents.

modèle physique qui est instable ou la méthode numérique qui comporte de l'aléatoire. Se pose alors la question de la reproductibilité et la nécessité de sa présence. En effet, sous l'apparence d'un calcul parfaitement reproductible peut se cacher un phénomène physique insoupçonné.

Dans [166], les auteurs présentent le cas d'un flambage de poutre à l'aide d'Europlexus [120], un code de simulation de dynamique rapide. Dans ce cas présenté sur la figure 2.12, une poutre parfaite en 1D se voit appliquer deux forces verticales en directions opposées sur chacune de ses extrémités. Sur la figure de gauche, on voit que le code IEEE-754 classique est parfaitement reproductible et la poutre se brise toujours au même endroit. Or lorsqu'on exécute le même code en arithmétique MCA, la solution trouvée change d'une exécution à l'autre. Les 100 résultats MCA sont superposés sur la figure de droite. Une question épineuse se pose alors : laquelle des 101 (100 MCA + 1 IEEE) solutions est la plus vraie ?

En pratique, une poutre parfaite n'existe pas et contient des imperfections. L'introduction de l'aléatoire permet donc de refléter cette imperfection de la réalité et de représenter différents schémas de flambage. Pour que ce reflet ait du sens, il est nécessaire que le bruit ajouté et les erreurs d'incertitudes sur le phénomène physique soient du même ordre de grandeur. On remarque également que le résultat de l'exécution IEEE-754 flambe, même sans l'introduction de bruits stochastiques. Ici l'accumulation d'erreurs d'arrondi créée une imperfection artificielle qui fait pencher la poutre d'un côté plus que de l'autre de manière déterministe. L'exécution IEEE-754 peut être ainsi vue comme une réalisation possible de MCA.

Des solutions

Plusieurs solutions ont été proposées pour améliorer la fidélité. Par exemple les accumulateurs de Kulisch [106] sont des accumulateurs de grande taille (4288 bits pour du binary64) qui

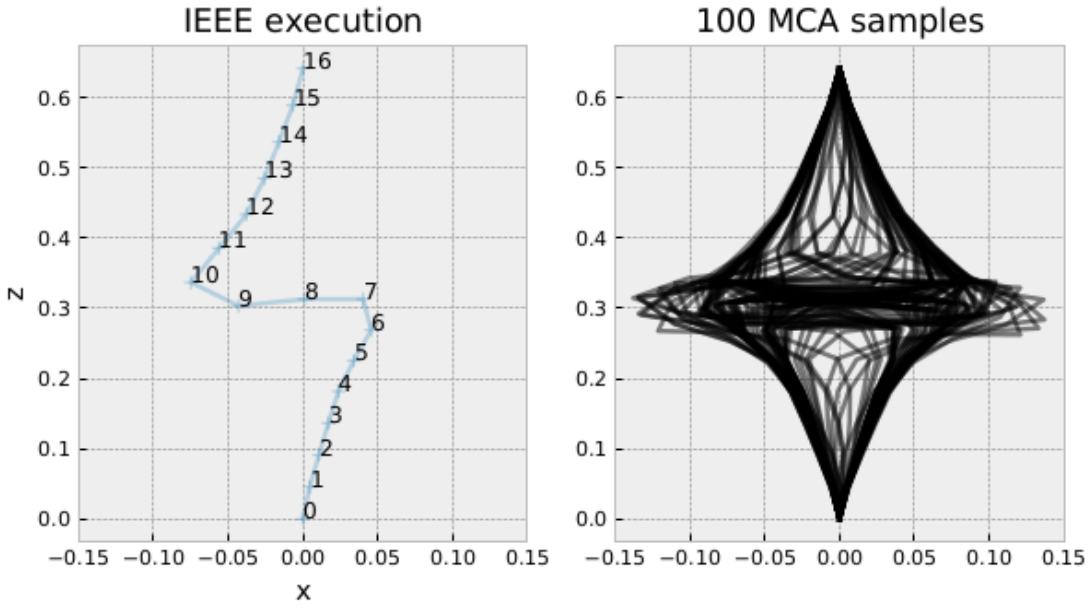


FIGURE 2.12 – Simulation du flambage d'une poutre 1D à l'aide du code Europlexus extraite de l'article [166]. Sur les deux figures, les abscisses représentent la déviation par rapport au centre en 0 et les ordonnées la hauteur par rapport au sol 0. Á gauche le résultat du calcul en IEEE-754 est déterministe et la poutre flambe toujours au même endroit. Á droite, les 100 résultats calculés avec la méthode MCA flambent tous suivant des schémas différents. L'erreur aléatoire introduite permet de révéler les différentes façons qu'a la poutre de flamber. Elles sont toutes des solutions acceptables de la physique simulée.

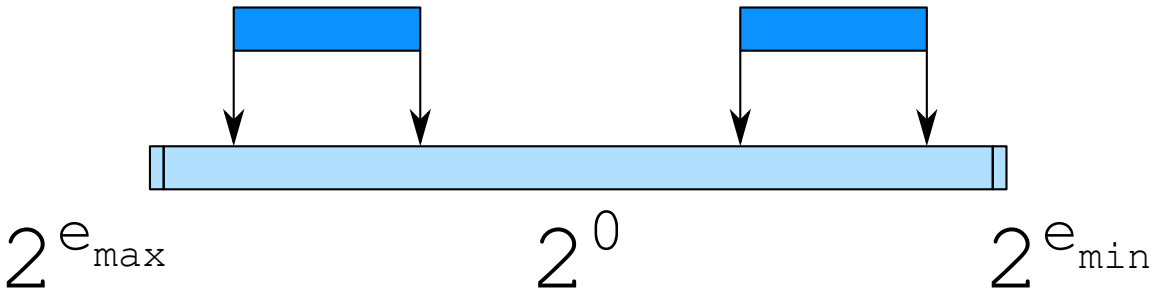


FIGURE 2.13 – Accumulateur de Kulisch. Sa taille est calculée pour contenir toute la dynamique des nombres d'un format donnée.

peuvent contenir toute la dynamique des nombres d'un format donné ce qui évite ainsi les problèmes d'absorptions. Une illustration en est donnée sur la figure 2.13.

Une autre partie concerne les librairies de calculs reproductibles [53, 54, 32, 92] de type BLAS [114]. Ces librairies utilisent des EFTs pour récupérer l'erreur perdue lors des calculs. Si ces librairies sont une étape vers des calculs plus justes, leurs performances restent cependant moins bonnes que leur version classique, et ce tout particulièrement pour les BLAS de niveau 3.

Cependant, l'étude du flambage de poutre sur Europlexus pose une question légitime sur la reproductibilité au bit près : est-elle souhaitable à chaque fois ? En effet, la basse fidélité est ici le signe d'une physique instable, chacune des solutions étant légitime d'un point de vue physique. Une haute fidélité a donc ici peu d'intérêt. L'analyse de sensibilité est également une piste intéressante pour révéler des instabilités liées à la conception du code mais il est complexe de les décorréler de l'incertitude du modèle physique. Pourtant, cette séparation améliorerait grandement la compréhension globale du code en différenciant ce qui contribue à la justesse de l'application de ce qui contribue à sa fidélité.

Enfin de manière plus générale, le coût humain pour maintenir cette reproductibilité est-il soutenable au vu des évolutions architecturales matérielles et logicielles toujours plus rapides ?

2.6 Arithmétiques stochastiques

La section 2.4.2 nous a montré que calculer avec une arithmétique de précision finie conduisait à souvent perdre une partie de l'information manipulée. Pour pallier ce problème, des efforts de reproductibilité ont été menés pour concevoir des codes de calculs plus stables d'une architecture à l'autre, à commencer par la norme IEEE-754 qui permet d'harmoniser l'exactitude des opérations élémentaires jusqu'aux BLAS reproductibles (cf. 2.5.3). Cependant, comme nous l'a montré l'exemple d'Europlexus (cf. 2.12), une haute fidélité n'est pas nécessairement la panacée lorsque le problème sous-jacent est instable par nature.

Calculer précisément l'erreur commise dans un code de calculs avec les méthodes formelles traditionnelles (cf. état de l'art) est un travail long et difficile, voire impossible, pour des applications disposant de nombreuses lignes de codes. Une autre façon de représenter cette erreur est de la considérer comme du bruit. En poursuivant l'analogie avec la théorie du signal, on peut alors voir les erreurs d'arrondis (cf. 2.4.2) comme un bruit venant perturber un signal. Ce signal étant vu comme le résultat exact d'un calcul. Estimer l'erreur revient donc ensuite à dissocier ce qui relève

```c
#include <stdio.h>

int main() {
 double a = 0;
 for (int i=0; i < 10000; i++) a += 0.1;
 printf("%0.17f\n", a);
 return 0;
}
```

FIGURE 2.14 – Résultat de l'exécution IEEE-754 `1000.00000000015882051`

Indice	IEEE-754	MCA
1	1000.00000000015882051	1000.00000000000773070
2	1000.00000000015882051	999.99999999999579359
3	1000.00000000015882051	1000.00000000000466116
4	1000.00000000015882051	999.99999999999795364
5	1000.00000000015882051	999.99999999999488409
6	1000.00000000015882051	999.99999999999204192
7	1000.00000000015882051	1000.00000000000386535
8	1000.00000000015882051	1000.00000000000568434
9	1000.00000000015882051	1000.00000000000454747
10	1000.00000000015882051	999.99999999999431566

TABLE 2.3 – Résultat de 10 exécutions du code de la figure 2.14. La version MCA, en introduisant un bruit stochastique permet d'identifier rapidement ce qui est juste de ce qui relève du bruit arithmétique.

du signal de ce qui relève du bruit.

Or le bruit introduit lors d'un calcul IEEE-754 est déterministe et reste donc le même d'une exécution à l'autre, sauf dans le cas d'une basse fidélité. En considérant ce bruit non plus comme déterministe mais aléatoire, il devient plus aisé de rapidement identifier le bruit du signal comme le montre l'exemple suivant sur la table 2.3. La modélisation du bruit numérique par une variable aléatoire est le coeur des arithmétiques stochastiques. L'idée est de voir chaque opération flottante comme un calcul exact plus une erreur aléatoire. En répétant les opérations stochastiques plusieurs fois, on peut alors obtenir une estimation de l'erreur commise.

Il existe deux principaux représentants des arithmétiques stochastiques : les méthodes CESTAC et MCA. La méthode CESTAC introduite par La Porte et Vignes en 1974 [181] propose d'introduire un arrondi aléatoire sur les résultats des calculs flottants pour modéliser les erreurs d'arrondi. L'Arithmétique de Monte Carlo quant à elle, théorisée par Stott Parker [146, 145] en 1997, modélise l'inexactitude d'une valeur flottante par une variable aléatoire.

Nous allons dans un premier temps présenter trois manières d'effectuer un arrondi aléatoire puis nous nous pencherons plus en détails sur la méthode MCA. Enfin, nous verrons comment estimer le nombre de chiffres significatifs d'un résultat stochastique grâce à un résultat théorique sur les intervalles de confiance.

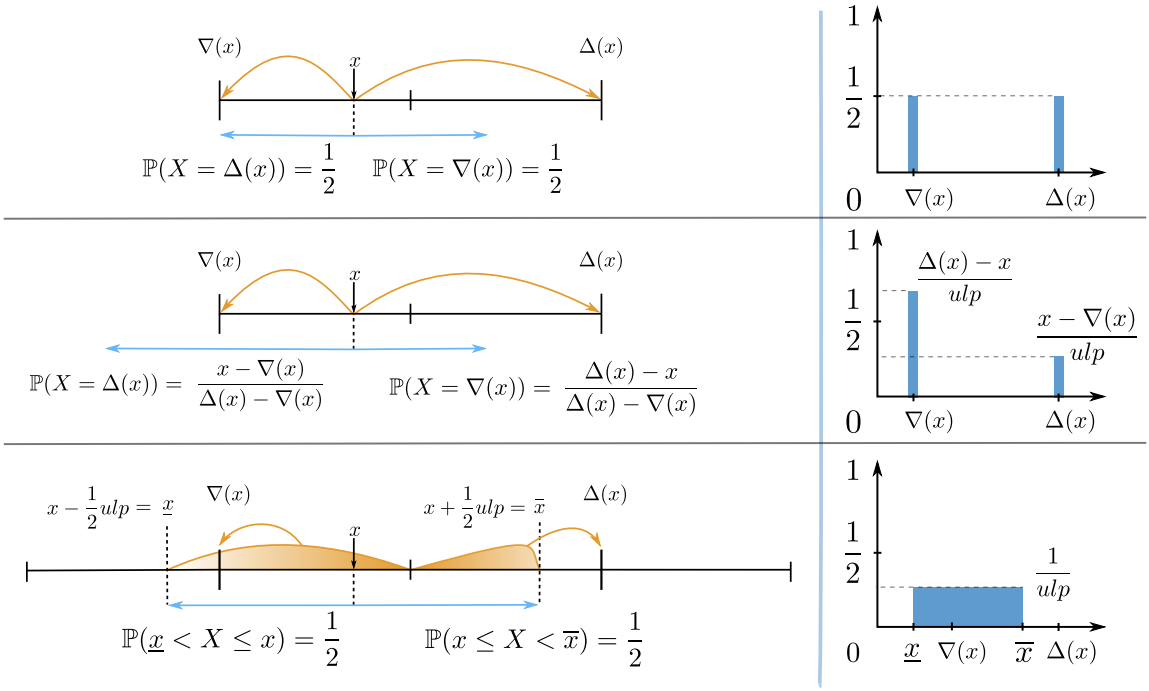


FIGURE 2.15 – Illustration de l'arrondi aléatoire classique (en haut), en moyenne (au milieu) et la version MCA (en bas) avec à droite les densités de probabilité correspondantes.

On utilisera les notations suivantes :

- x la variable réelle exacte que l'on souhaite calculer.
- $X_1, X_2, ...X_N$ les valeurs de N exécutions en arithmétique stochastique d'un programme calculant x, vues comme des réalisations d'une variable aléatoire X.
- $\mu = \mathbb{E}[X]$ l'espérance de X.
- $\tilde{\mu} = \frac{1}{N}\sum_{i=1}^{N} X_i$, la moyenne empirique de X estimant μ.
- $\sigma = \mathbb{S}[X]$ l'écart-type de X.
- $\tilde{\sigma} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(X_i - \tilde{\mu})^2}$, l'écart-type empirique de X estimant $\tilde{\sigma}$.

2.6.1 Arrondi aléatoire

L'arrondi aléatoire consiste à arrondir un flottant vers le haut (Δ) ou vers le bas (∇) de manière aléatoire pour estimer la sensibilité d'un code aux erreurs d'arrondi. Nous allons présenter trois variantes d'arrondi aléatoire illustrées sur la figure 2.15.

Dans la première version introduite dans la méthode CESTAC, la variable X a la même probabilité d'être arrondie soit vers le haut, soit vers le bas. De plus le résultat x n'est pas bruité s'il est représentable dans le format flottant utilisé. Cela se traduit par la fonction suivante :

$$random\_round(x) = \begin{cases} x & \text{si } x \in \mathbb{F} \\ \xi\Delta(x) + (1-\xi)\nabla(x) & \text{sinon} \end{cases}$$

où une variable aléatoire ξ suivant une loi de Bernoulli de paramètre $p = \frac{1}{2}$.

Une autre version proposée [64] pondère la probabilité d'un arrondi avec sa distance par rapport à x. Ainsi, si x est plus proche de $\nabla(x)$ il aura une plus grande probabilité d'être arrondi vers le bas et inversement pour l'arrondi dans l'autre sens :

$$random\_round\_average(x) = \begin{cases} x & \text{si } x \in \mathbb{F} \\ \xi\Delta(x) + (1-\xi)\nabla(x) & \text{sinon} \end{cases}$$

où $\mathbb{P}[\xi = 0] = \dfrac{\Delta(x) - x}{\Delta(x) - \nabla(x)}$ et $\mathbb{P}[\xi = 1] = \dfrac{x - \nabla(x)}{\Delta(x) - \nabla(x)}$.

Cet arrondi dispose de l'élégante propriété suivante : l'espérance d'une opération arithmétique est le résultat exact de cette opération. En effet,

$$\begin{aligned} \mathbb{E}[random\_round\_average(x)] &= \mathbb{P}[\xi = 0]\Delta(x) + \mathbb{P}[\xi = 1]\nabla(x) \\ &= \Delta(x)\frac{x - \nabla(x)}{\Delta(x) - \nabla(x)} + \nabla(x)\frac{\Delta(x) - x}{\Delta(x) - \nabla(x)} \\ &= \frac{x(\Delta(x) - \nabla(x))}{\Delta(x) - \nabla(x)} \\ &= x. \end{aligned}$$

Enfin la dernière version tirée de MCA propose d'ajouter un bruit uniformément distribué entre plus et moins un demi-ulp. Contrairement aux deux modes précédents, la variable aléatoire est continue avec le calcul intermédiaire effectué en précision infinie. La formule présentée est un cas particulier de la formule plus générale présentée dans la section 2.6.2 suivante.

$$random\_rounding\_mca(x) = x + ulp.\xi$$

où $\xi \in]-\frac{1}{2}, \frac{1}{2}[$ une variable aléatoire uniformément distribuée.

Comme pour l'arrondi aléatoire moyen, l'arrondi aléatoire MCA a une espérance égale au résultat exact :

$$\mathbb{E}[x + ulp.\xi] = \mathbb{E}[x] + ulp\mathbb{E}[\xi] = x$$

avec $\mathbb{E}[\xi] = 0$.

De plus, cet arrondi aléatoire préserve les opérations exactes puisqu'on rajoute (ou enlève) moins d'un demi-ulp comme le montre la figure 2.16.

2.6.2 Arithmétique de Monte Carlo

> *Monte Carlo Arithmetic can be viewed as making a distinction between exact and inexact values, and implementing this distinction by modeling inexact values as random variables.*

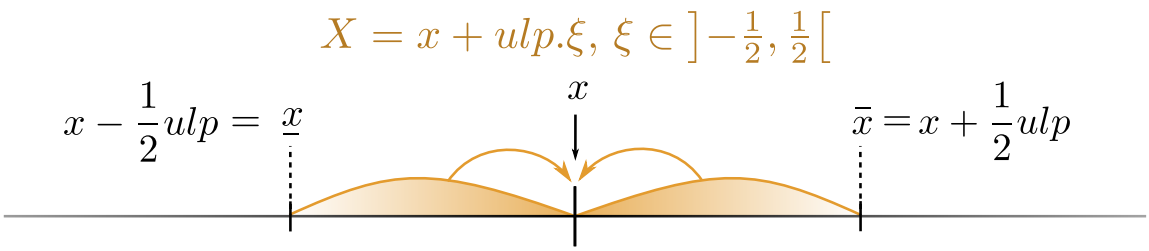


FIGURE 2.16 – Illustration de la préservation des opérations exactes durant l'utilisation de MCA en mode arrondi aléatoire.

S. Parker

Après avoir vu comment modéliser un arrondi aléatoire dans la section 2.6.1 précédente, nous allons maintenant présenter plus en détail la méthode MCA. L'Arithmétique de Monte Carlo théorisée par Stott Parker [146, 145] en 1997 modélise l'inexactitude d'une valeur flottante par une variable aléatoire. Dans le modèle MCA, la représentation inexacte d'un réel x sur t chiffres s'écrit :

$$inexact(x) = x + \beta^{e_x - t}\xi$$

où β est la base de calcul, $e_x = \lfloor \log_\beta |x| \rfloor + 1$ la magnitude de x et $\xi \in]-\frac{1}{2}, \frac{1}{2}[$ une variable aléatoire uniformément distribuée.

En pratique, le principe de MCA est de remplacer chacune des opérations flottantes d'un programme par leur équivalent stochastique et d'exécuter le programme plusieurs fois. En supposant que le programme analysé comporte k opérations, MCA modélise l'inexactitude de la k-ème opération dont le résultat exact est x^k par une variable aléatoire X^k. Ainsi chaque valeur $\tilde{x_i^k}$ qui est le résultat numérique de la k-ème opération de la i-ème exécution peut alors être vue comme une réalisation X_i^k de la variable aléatoire X^k. L'idée est d'approximer fidèlement la distribution des X_k en prenant une taille d'échantillon suffisamment élevée. Ce qui permet d'estimer la valeur réelle x_k par la moyenne empirique $\tilde{\mu}$ si on suppose que $\mathbb{E}[X_k] = x_k$.

L'hypothèse - $\mathbb{E}[X_k] = x_k$ est cependant une hypothèse forte. Si elle est vraie pour une opération arithmétique de base, il est en revanche faux de considérer le résultat d'une suite d'opérations arithmétiques de base comme le résultat théorique attendu. En effet, comme le montre la section 2.4.2 l'utilisation d'une précision finie conduit à une erreur systématique (cf. 2.2.7).

La validité de cette hypothèse repose sur le fait que l'estimation de μ par $\tilde{\mu}$ est bonne. Or si la valeur de μ est théorique et parfois inconnue, celle de $\tilde{\mu}$ repose sur les échantillons X_i calculés à l'aide d'une arithmétique finie puisque chaque opération MCA est arrondie vers le format flottant d'origine.

Une des forces du modèle MCA est l'utilisation de la variable t appelée *précision virtuelle*. Cette précision est qualifiée de virtuelle par opposition à la *précision machine* p utilisée en matériel. Ainsi, l'idée est de pouvoir simuler des précisions plus faibles que la précision machine lorsque $t < p$. Ceci est résumé sur la figure 2.17.

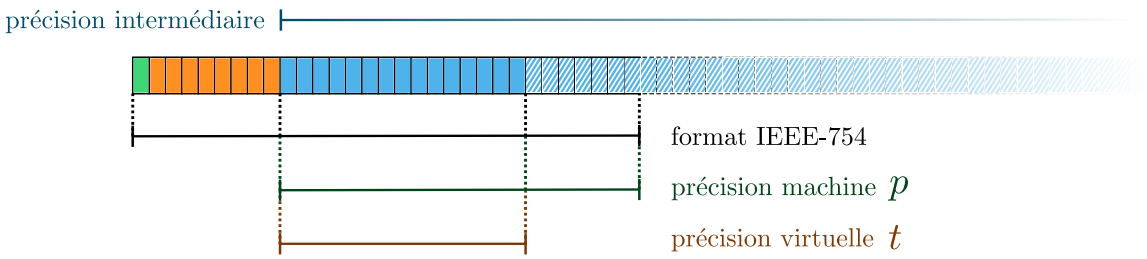


FIGURE 2.17 – Fonction de bruitage de MCA. La précision virtuelle t permet de simuler des précisions plus faibles en ajoutant un bruit (en bleu hachuré) de plus grande magnitude. Si $t = p$ on retrouve un arrondi aléatoire classique. En théorie, le calcul intermédiaire pour ajouter le bruit au nombre original est fait en précision infinie. En pratique seul $2p$ bits de précision suffisent (cf. [145, p.38]) ce qui revient à utiliser p guards digits.

MCA introduit trois modes pour simuler l'erreur stochastique des opérations élémentaires $\{+, -, \times, /\}$ flottantes. Si $x, y \in \mathbb{F}_{\beta,p}$ sont deux valeurs flottantes et $\bullet \in \mathbb{R} \to \mathbb{R}$ est l'opération réelle associée à l'opération flottante $\diamond \in \mathbb{F}_{\beta,p} \to \mathbb{F}_{\beta,p}$, alors :

- PB **Precision Bounding** : $PB(x \diamond y) = \circ_p(inexact(x) \bullet inexact(y))$
  Le mode Precision Bounding (PB) borne la précision des opérandes et permet de détecter les annulations catastrophiques.
- RR **Random Rounding** : $RR(x \diamond y) = \circ_p(inexact(x \bullet y))$
  Le mode Random Rounding (RR) bruite le résultat d'une opération et permet de modéliser une erreur d'arrondi.
- **Full MCA** : $FULL(x \diamond y) = \circ_p(inexact(inexact(x) \bullet inexact(y)))$
  Le mode FULL MCA combine le mode PB et le mode RR.

Random Rounding

Le mode RR simule un arrondi aléatoire comme présenté à la section précédente 2.6.1. Ce mode préserve les opération exactes, c'est-à-dire les opérations dont le résultat est représentable dans le format flottant d'origine.

Precision Bounding

Le mode PB borne la précision des opérandes et permet de détecter les annulations catastrophiques. En effet, le mode RR seul ne permet pas de détecter les annulations puisqu'il introduit l'erreur aléatoire sur le résultat de l'opération. Or le problème avec les annulations catastrophiques c'est que le potentiel bruit dû aux erreurs d'arrondi accumulées au préalable et qui se trouve sur les bits de poids faibles, est remonté dans les bits de poids forts. Ainsi, dans une arithmétique déterministe telle que IEEE-754, où les annulations sont des opérations exactes, il est difficile de quantifier le nombre de bits justes. Le résultat d'une annulation est donc considéré comme exact à priori si on ne prend pas en compte les erreurs précédentes commises.

L'introduction du bruit stochastique sur les opérandes permet de résoudre ce problème comme l'illustre la figure 2.18. En effet, contrairement au mode RR, le bruit sera remonté durant la soustraction et fera apparaître clairement la baisse du nombre de chiffres significatifs comme le montre

Échantillon	RR	PB
0	0x1.c000000p-20	0x1.ccc9eep-20
1	0x1.c000000p-20	0x1.b9358cp-20
2	0x1.c000000p-20	0x1.be81ecp-20
3	0x1.c000000p-20	0x1.b02d66p-20
4	0x1.c000000p-20	0x1.d04fdap-20
5	0x1.c000000p-20	0x1.ba070ep-20
6	0x1.c000000p-20	0x1.c46ddcp-20
7	0x1.c000000p-20	0x1.ce87ecp-20
8	0x1.c000000p-20	0x1.c29868p-20
9	0x1.c000000p-20	0x1.c7a31ep-20
Moyenne	0x1.c00000p-20	0x1.c2d2eap-20
Écart-type	0	0x1.41ffdp-25
Nombre de bits significatifs	**∞**	**5,48**

TABLE 2.4 – Comparaison des modes RR et PB de MCA sur la soustraction flottante `0x1.fffffep0 - 0x1.ffffe2p0` avec une précision virtuelle $t = 24$. Puisque l'opération est exacte, le mode RR la préserve et ne la bruite pas. En revanche le mode PB en introduisant du bruit sur les entrées permet de mettre en lumière l'annulation. Ceci se traduit par un nombre de chiffres significatifs estimé plus précis pour le mode PB. Une illustration est également donnée sur la figure 2.18.

les résultats de la table 2.4. Dans cet exemple simple, on exécute 10 fois le programme réalisant la soustraction `0x1.fffffep0 - 0x1.ffffe2p0` en mode RR puis PB. Puisque le résultat est exact, le mode RR préserve l'opération et ne la bruite pas. Or seuls les premiers bits du résultat sont à priori justes. En introduisant du bruit sur les opérandes, le mode PB permet de signifier la précision réelle qui résulte de la soustraction.

2.6.3 Estimation du nombre de chiffres significatifs

L'introduction de l'aléatoire facilite l'identification des chiffres non-significatifs. En effet, on comprend de manière intuitive que des réalisations X_i MCA ne partagent pas les mêmes chiffres là où l'erreur aléatoire est venue les perturber. De manière analogue, les X_i échantillons MCA obtenus après n exécutions possèdent un certain nombre s de chiffres en commun comme le montre l'exemple sur la table 2.4. Pour estimer ce nombre s, Parker propose de regarder le log du ratio entre la moyenne $\tilde{\mu}$ et l'écart-type $\tilde{\sigma}$ empirique des échantillons. L'idée peut se résumer ainsi : ôter les chiffres bruités au résultat exact. Si $\tilde{\mu} \simeq 2^{signal}$ et $\tilde{\sigma} \simeq 2^{bruit}$ alors $\log_2 \left| \frac{2^{signal}}{2^{bruit}} \right| = signal - bruit$. Parker formalise ce principe pour estimer le nombre de chiffres significatifs (noté s) d'un calcul.

Définition 2.6.1 (Nombre de chiffres significatifs MCA)**.**

$$s_{\text{MCA}} = \lim_{N \to \infty} -\log_\beta \left| \frac{\tilde{\sigma}}{\tilde{\mu}} \right| = -\log_\beta \left| \frac{\sigma}{\mu} \right|.$$

L'avantage de cette formulation est qu'elle ne nécessite aucune hypothèse sur la distribution de X. Néanmoins, aucun intervalle de confiance n'est donné pour aider à choisir le nombre d'échan-

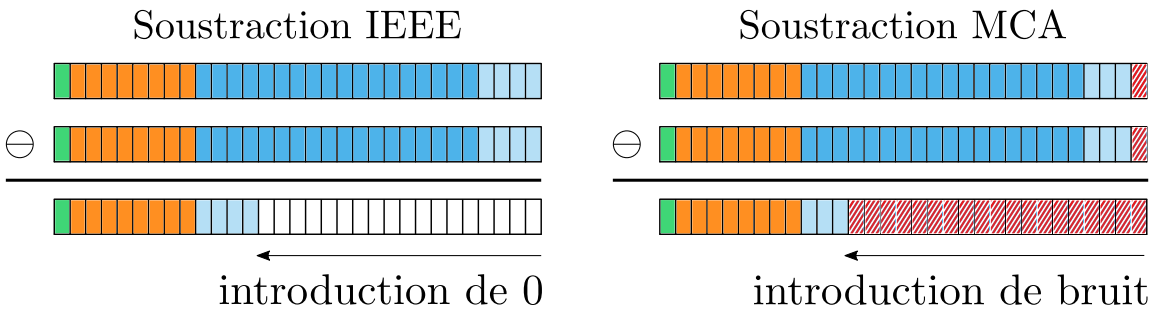


FIGURE 2.18 – Détection d'une annulation catastrophique avec MCA. Comme la soustraction est une opération exacte en IEEE-754, il est nécessaire de vérifier si les opérandes sont entachées d'erreurs numériques pour estimer le nombre de bits corrects. Avec le mode PB de MCA, l'erreur aléatoire introduite fait apparaître en moyenne les chiffres justes du bruit.

tillons à prendre. Récemment, Sohier et al. [166] ont proposés une définition probabiliste plus rigoureuse du nombre de chiffres significatifs ainsi que des intervalles de confiance associés dans le cadre des arithmétiques stochastiques en base binaire ($\beta = 2$).

Pour définir le s de X, les auteurs utilisent une variable aléatoire Z construite comme l'erreur absolue ou relative avec une valeur de référence fournie par l'utilisateur avec $\mathbb{E}[Z] = 0$. Cette référence peut être soit une variable réelle (y) soit une variable aléatoire (Y), ce qui est utile dans le cas d'un résultat non-déterministe par exemple. Ainsi pour l'erreur relative, $Z = \frac{X}{y} - 1$ ou $Z = \frac{X}{Y} - 1$.

Le nombre de bits significatifs est défini comme :

Définition 2.6.2 (Nombre de bits significatifs en commun)**.** x et y ont s bits en commun si

$$\left|\frac{x}{y} - 1\right| \leqslant \frac{1}{2} 2^{1-s} = 2^{-s}.$$

À partir de cette définition 2.6.2, les auteurs définissent le nombre de chiffres significatifs comme le plus grand s tel que :

Définition 2.6.3 (Nombre de chiffres significatifs probabilistes)**.** En posant que le k-ème bit de X est significatif avec une probabilité p comme :

$$\mathbb{P}\left(|Z| \leqslant 2^{-k}\right) \geqslant p$$

alors on définit s comme étant le plus grand réel tel que :

$$\max_{s \in \mathbb{R}} \left\{ \mathbb{P}\left(|Z| \leqslant 2^{-s}\right) \geqslant p \right\}.$$

Intervalles de confiance

À partir de la définition 2.6.3, les auteurs proposent deux formules pour calculer s, une dans le cas où X suit une distribution normale et l'autre dans le cas plus général où aucune hypothèse n'est faite sur la distribution sous-jacente. Nous allons présenté les principaux résultats théoriques

obtenus dans cet article.

Cas d'une loi normale. Dans le cas où Z suit une loi normale $\mathcal{N}(0, \sigma)$, il est prouvé (Théorème 4.1 [166, p.12]) que le k-ème bit est signifiant avec une probabilité :

$$\mathbb{P}[|Z| \leqslant 2^{-k}] = 2\mathbb{P}\left[\frac{Z}{\sigma} \leqslant \frac{2^{-k}}{\sigma}\right] - 1 = 2F\left(\frac{2^{-k}}{\sigma}\right) - 1$$

où F est la fonction de répartition de la loi normale de moyenne 0 et de variance 1. Le terme du milieu est dû à la symétrie de la loi normale.

En utilisant la fonction cumulative inverse de la loi normale F^{-1}, on peut alors retrouver le nombre de chiffres significatifs s avec une probabilité p :

$$s = -\log_2(\sigma) - \log_2\left(F^{-1}\left(\frac{p+1}{2}\right)\right).$$

Néanmoins en pratique seule une estimation $\tilde{\sigma}$ de l'écart-type est connue. Les auteurs proposent donc d'utiliser cette estimation pour calculer une borne sur le s réel avec une probabilité p et un niveau de confiance $1 - \alpha$:

$$s \geqslant -\log_2(\tilde{\sigma}) - \left[\frac{1}{2}\log_2\left(\frac{n-1}{\chi^2_{1-\alpha/2}}\right) + \log_2\left(F^{-1}\left(\frac{p+1}{2}\right)\right)\right]$$

où F^{-1} est la fonction cumulative inverse de la loi normale, $\tilde{\sigma}$ l'écart-type empirique de Z pour n échantillons et $\chi^2_{1-\alpha/2}$ la loi du χ^2 [157].

Le s_{MCA} donné par Parker peut ainsi se voir comme une borne inférieure sur le vrai s de X. Les auteurs donnent l'exemple suivant : avec $n = 200$ échantillons, s_{MCA} fournit une borne inférieure avec une probabilité de 66% pour un niveau de confiance de 0.66.

Cas général. L'hypothèse de normalité n'est pas vraie dans tout les cas comme le montre Parker [145, p.49] ainsi que les auteurs du papier [166, p.24]. Il est difficile de raisonner sur la distribution de probabilité de X en l'absence d'hypothèses. Pour contourner ce problème, les auteurs utilisent une épreuve de Bernoulli.

Pour rappel la loi de Bernoulli permet de modéliser le résultat d'une épreuve comportant deux issues comme le lancer d'une pièce. Cette loi est définie par la probabilité p d'avoir un succès et de manière analogue par la probabilité $1 - p$ d'avoir un échec.

Les auteurs définissent un succès comme le fait que l'ensemble des réalisations d'un échantillon possèdent k premiers bits en commun avec la référence, ce qui définie s comme le plus grand bit (plus grand k) pour lequel tous les bits de poids fort sont identiques :

$$s = \max\left\{k \in [\![1, 53]\!] \mid \forall i \in [\![1, n]\!], |Z_i| \leqslant 2^{-k}\right\}.$$

Le nombre d'échantillons n à tirer est alors déterminé par la probabilité p et le niveaux de confiance $1 - \alpha$ désiré suivant l'inégalité :

$$n \leqslant \frac{\ln(\alpha)}{\ln(p)}.$$

2.7 Conclusion

L'arithmétique des ordinateurs est un domaine bien étudié mais qui refait surface à chaque transition technologique. La norme IEEE-754, utile pour uniformiser la reproductibilité des calculs s'essouffle et a du mal à gérer l'indéterminisme toujours plus présent dans les calculs et les architectures. Les développeurs se tournent également vers des formats fait maisons pour répondre aux besoins des nouvelles applications de *deep learning*. Un parallèle peut être dressé entre la situation actuelle et la situation d'avant la norme IEEE-754 où chaque constructeur développait son propre format flottant.

La question de la reproductibilité est également une question centrale puisque que nous avons montré qu'elle était présente à tous les niveaux du calcul, des instructions vectorielles qui peuvent modifier l'ordre des calculs jusqu'à l'ordonnancement des tâches en passant par le découpage des maillages. Il est donc important d'avoir des modèles de calcul qui puisse rendre compte de cet indéterminisme et mesurer ces effets sur la qualité numérique des résultats calculés.

Ainsi nous avons montré que l'usage des arithmétiques stochastiques offrait une solution élégante pour détecter des erreurs liées au modèle flottant et estimer le nombre de chiffres significatifs d'un résultat. Les arithmétiques stochastiques s'avèrent également être des outils efficaces pour juger de la pertinence ou non d'avoir une reproductibilité très précise des résultats. Enfin, les nouveaux outils théoriques sur les intervalles de confiance pour unifier les résultats connus permettent de renforcer leur intégration au sein des modèles et des codes de simulations. Ceci permet alors un meilleur processus de validation et de vérification.

Chapitre 3

Verificarlo

Verificarlo [56] est né d'une volonté d'automatiser la vérification et la validation des calculs numériques du modèle flottant (décrit en section 2.1) pour les codes HPC. Pensé comme un compilateur, le projet s'est ensuite élargi pour intégrer plusieurs aspects de l'analyse numérique des codes : la détection d'erreur, la génération de traces numériques, la visualisation de la qualité numérique, l'optimisation de la précision de calcul ... Cependant appliquer Verificarlo sur un code industriel pose des problèmes de passage à l'échelle. Par exemple, l'introduction de bruit aléatoire peut faire crasher l'application. Cela indique aux développeurs qu'il a des instabilités numériques mais ne l'aide pas à localiser les sources d'erreurs afin de les corriger. Autre exemple, l'utilisation de la précision mixte permet d'améliorer les performances en densifiant les calculs et diminuant le trafic mémoire (cf. section 5.4.5). Néanmoins sans méthodologie pour identifier les endroits du code qui peuvent être passés dans un format réduit, il peut être difficile pour ne pas dire impossible à un développeur de les trouver dans une application contenant plusieurs centaines de milliers de lignes de codes. Il s'agit alors de mettre en place des stratégies pour aider le développeur à naviguer rapidement dans ces espaces de recherche afin identifier les sources d'erreurs numériques pour pouvoir mettre en place des corrections ou trouver les endroits du codes pouvant être passés en binary32 par exemple.

Ce chapitre débutera par la présentation de Verificarlo où nous détaillerons son schéma conceptuel et expliquerons comment il remplace les instructions de calcul flottant par des appels génériques. Nous présenterons également les backends disponibles dans Verificarlo dont le backend `BITMASK` qui a été développé durant cette thèse. Enfin nous terminerons par présenter des méthodes de réduction de dimensionnalités qui sont utilisées pour déboguer ou optimiser les calculs flottants pour des codes HPC.

3.1 Schéma conceptuel

Verificarlo est un compilateur basé sur l'infrastructure de compilation LLVM [113] qui remplace automatiquement les opérations flottantes par leur équivalent dans d'autres modèles de calcul.

Le processus de compilation traduit un langage source lisible par un humain en langage compréhensible par la machine. Ce processus se décompose généralement en trois sections présentées sur la figure 3.1 : le *front-end* qui transforme le langage source en langage intermédiaire puis le *middle-end* qui applique des optimisations pour rendre le code généré plus performant et enfin le *back-end* qui

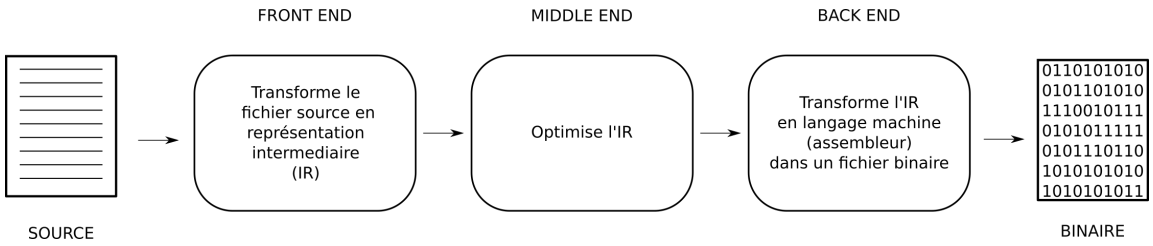


FIGURE 3.1 – Les trois étapes du processus de compilation : *front-end*, *middle-end* et *back-end*.

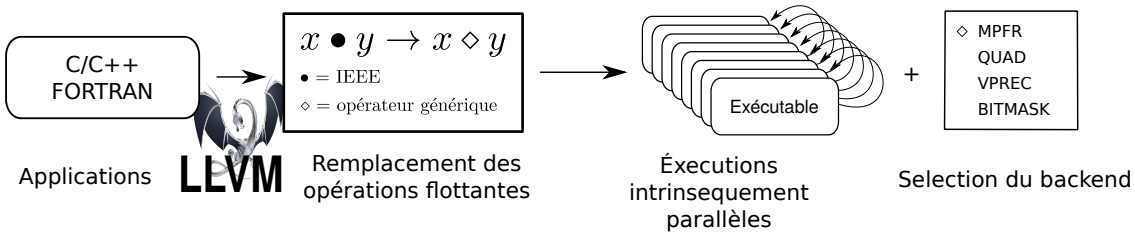


FIGURE 3.2 – Schéma conceptuel de Verificarlo. Construit autour du compilateur LLVM, Verificarlo accepte les langages reconnus par LLVM puis remplace les opérations flottantes par un opérateur générique. Le choix du mode de bruitage est fait à l'exécution. Le code peut être ensuite exécuté plusieurs fois en concurrence disjointe. A noter que les backends VPREC et BITMASK ne nécessitent qu'une seule exécution.

traduit le langage intermédiaire en langage assembleur. Dans cette thèse, nous utiliserons le terme de *backend* pour faire référence à une implémentation d'un modèle de calcul comme par exemple les backends `MPFR` et `QUAD` qui sont deux implémentations du modèle MCA.

Verificarlo peut être découpé suivant le même schéma comme le montre la figure 3.2. Verificarlo utilise le *front-end* clang de LLVM qui supporte plusieurs langages sources dont le `C` et le `C++`. Clang transforme ces langages sources en un langage intermédiaire appelé IR. C'est au niveau de l'IR que Verificarlo travaille pour remplacer les instructions flottantes par des appels génériques. Cette passe dite d'instrumentation intervient après application des optimisations par LLVM et avant la génération du code assembleur. Instrumenter après application des optimisations est important pour capturer les effets des optimisations sur la qualité numérique du code généré comme nous le verrons dans la section 3.1.3. Une fois instrumenté, l'utilisateur peut choisir par quel backend il souhaite remplacer les appels génériques durant l'exécution. Dans le cas de backend introduisant de l'aléatoire, plusieurs exécutions du même programme seront nécessaires. Cela est transparent en terme de surcoût pour l'utilisateur puisque les exécutions sont indépendantes les unes des autres.

La figure 3.3 montre un exemple d'utilisation de Verificarlo sur un test minimal.

3.1.1 Compilation

Le choix de LLVM a été guidé par sa polyvalence et sa flexibilité. En effet, LLVM accepte de nombreux langages en entrée dont le `C,C++` et le `FORTRAN` qui représentent encore aujourd'hui la majorité des langages utilisés dans les codes HPC. Ces langages sources sont traduits en un langage

```
bash$ cat > test.c <<HERE
#include <stdio.h>
int main() {
 double a = 0;
 for (int i=0; i < 10000; i++) a += 0.1;
 printf("%0.17f\n", a);
 return 0;
}
HERE

bash:$ verificarlo test.c -o test
bash:$ VFC_BACKENDS="libinterflop_mca.so" ./test
999.99999999999795364
bash:$ VFC_BACKENDS="libinterflop_mca.so" ./test
1000.00000000000102318
bash:$ VFC_BACKENDS="libinterflop_mca.so" ./test
1000.00000000000386535
```

FIGURE 3.3 – Exemple minimal d'utilisation de Verificarlo. L'instrumentation est transparente pour l'utilisateur car elle ne nécessite pas de réécriture et le binaire peut être exécuté directement. Le choix du backend se fait par l'intermédiaire de la variable d'environnement `VFC_BACKENDS` avec ici celui de `QUAD`. On voit ici que le résultat de la somme varie d'une exécution á l'autre.

```
float add(float a, float b) {
  return a + b;
}
```

```
define float @add(float, float) {
 %3 = alloca float
 %4 = alloca float
 store float %0, float* %3
 store float %1, float* %4
 %5 = load float, float* %3
 %6 = load float, float* %4
 %7 = fadd float %5, %6
 ret float %7
}
```

FIGURE 3.4 – À gauche, le code `C` d'une addition flottante. À droite, sa représentation intermédiaire généré par `clang 6.0.0` avec `-O0`.

intermédiaire noté IR (*Intermediate Representation*) grâce à des compilateurs frontend comme par exemple `clang` pour les langages `C` et `C++`. Pour le langage `FORTRAN`, Verificarlo utilise le module `DragonEgg` [118] du compilateur GNU C Compiler () puisque le frontend `flang` [142] est encore en cours de développement. Le module `dragonegg` permet de générer de l'IR LLVM à partir de GCC. L'IR est un langage assembleur virtuel, c'est-à-dire qu'il ne correspond pas à un jeu d'instruction d'une machine physique réelle mais permet de s'abstraire à la fois les langages d'origines et les architectures ciblées. LLVM et sa représentation intermédiaire seront détaillés en section 4.3.4. La figure 3.4 montre la traduction d'un code source écrit en langage `C` vers l'IR LLVM.

3.1.2 Instrumentation

Après la traduction du langage primaire en IR, une passe de compilation remplace toutes les opérations flottantes arithmétiques $+, -, \times, /$ et de comparaison $<, \leqslant, >, \geqslant, ==, !=$ trouvées par un appel générique. C'est l'étape d'instrumentation du code. Verificarlo permet de choisir les fonctions

à instrumenter. Cet appel générique est chargé depuis une table virtuelle (*vtable*) qui contient l'ensemble des opérations à implémenter. Cette interface présentée sur la figure 3.5 est codée sous forme de structure où chaque champ correspond à une opération flottante d'un type donné. Cette architecture permet de s'abstraire du backend utilisé et simplifie la passe de compilation. En effet, puisque les éléments de la structure ont une place fixe, la position de la fonction à charger dans la vtable est calculée suivant l'opération instrumentée. De plus, tout backend respectant cette structure s'interface sans problèmes avec Verificarlo.

La figure 3.6 montre l'instrumentation du code présenté en figure 3.4. À gauche, l'instruction `%7` qui est le résultat de l'addition flottante des registres `%5` et `%6` est remplacée à droite par le chargement d'une fonction ayant la même signature que l'instruction `fadd` : `float (float, float)` qui se lit
$\texttt{type}_{retour}$ `(`$\texttt{type}_{arg_1}$`, `$\texttt{type}_{arg_2}$`, ..., `$\texttt{type}_{arg_N}$`)`. Ce chargement est réalisé au travers de l'instruction LLVM `getelementpointer` qui est utilisée pour calculer l'adresse d'un élément d'un type agrégé (tableau ou structure). La place de l'élément est déterminée par la valeur du dernier argument (ici 0) ce qui correspond à la première fonction de la vtable. Dans la convention suivi par Verificarlo, cette fonction correspond à l'addition. Verificarlo utilise également les types vectoriels de LLVM pour pouvoir instrumenter les opérations flottantes vectorisées. Ces types vectoriels sont des vecteurs d'éléments de types natifs permettant d'exprimer la vectorisation au sein de l'IR.

3.1.3 Effets des optimisations

L'instrumentation à la fin des optimisations middle-end permet de conserver les optimisations appliquées par le compilateur et de rester au plus près du code originel compilé. En effet, nous avons vu en section 2.5.2 que les optimisations sur les calculs flottants appliquées par le compilateur pouvaient modifier et nuire à la qualité numérique du code assembleur généré.

Pour comprendre reprenons l'exemple de la somme de Kahan (cf. figure 2.7) présenté en section 2.5.2 et regardons en détail l'IR généré. La figure 3.7 présente l'IR du coeur de la boucle généré par `clang` sans optimisations. On retrouve le comportement attendu avec la préservation des opérations intermédiaires. Par contre, la figure 3.8 montre la déconstruction des termes correctifs et même la vectorisation du coeur de boucle lorsque les optimisations agressives sont activées avec les drapeaux `-O3 -ffast-math`. Ainsi, en effectuant l'instrumentation après les optimisations, Verificarlo capture uniquement les instructions flottantes qui ne sont pas éliminées par les passes d'optimisation ce qui lui permet de se rapprocher le plus possible de ce qui sera réellement exécuté sur le processeur.

Le choix des backends et des paramètres est passé à l'aide de variables d'environnement. Nous présentons ici les trois principales variables communes aux backends :

- `VERIFICARLO_BACKEND` : permet de choisir le backend à utiliser durant l'exécution parmi `MPFR`, `QUAD`, `VPREC` et `BITMASK`.
- `VERIFICARLO_PRECISION` : permet de choisir la précision virtuelle de travail.
- `VERIFICARLO_<BACKEND>_MODE` : permet de choisir le mode de calcul pour le backend sélectionné. Par exemple, le mode RR ou PB pour le modèle MCA.

```
struct mca_interface_t {
 /* Retourne x + y */
 float (*floatadd)(float x,
          float y);
 /* Retourne x - y
 float (*floatsub)(float x,
          float y);
 /* Retourne x * y
 float (*floatmul)(float x,
          float y);
 /* Retourne x / y
 float (*floatdiv)(float x,
          float y);
 /* Retourne x cmp y, ou `cmp` est
  dans {<,<=,>,>=,==,!=} */
 bool (*floatcmp)(float x,
         float y,
         enum FCMP);

 /* Retourne x + y */
 double (*doubleadd)(double x,
           double y);
 /* Retourne x + y */
 double (*doublesub)(double x,
           double y);
 /* Retourne x + y */
 double (*doublemul)(double x,
           double y);
 /* Retourne x + y */
 double (*doublediv)(double x,
           double y);
 /* Retourne x cmp y, ou `cmp` est
  dans {<,<=,>,>=,==,!=} */
 bool (*doublecmp) (double x,
           double y,
           enum FCMP);

 /* Modifie la graine du generateur
  de nombres aleatoires */
 void (*seed)(void);
 /* Modifie le mode de bruitage */
 int (*set_mca_mode)(int);
 /* Modifie la precision virtuelle */
 int (*set_mca_precision)(int);
};
```

```
@_vfc_current_mca_interface = external global
    {
 float (float, float)*,
 float (float, float)*,
 float (float, float)*,
 float (float, float)*,
 i1 (float, float, i8)*,
 double (double, double)*,
 double (double, double)*,
 double (double, double)*,
 double (double, double)*
 i1 (double, double, i8)*,
}
```

FIGURE 3.5 – Interface des backends avec la version `C` à gauche et la version IR à droite. Chacun des éléments de la structure est une opération à implémenter pour pouvoir s'interfacer avec Verificarlo. La convention choisi dans Verificarlo spécifie dans l'ordre, l'addition, la soustraction, la multiplication et la division. Les trois dernières fonctions du backend ne sont pas incluses car elles sont uniquement appelées lors de l'initialisation.

```
define float @add(float, float) {
 %3 = alloca float
 %4 = alloca float
 store float %0, float* %3
 store float %1, float* %4
 %5 = load float, float* %3
 %6 = load float, float* %4
 %7 = fadd float %5, %6
 ret float %7
}
```

```
define float @add(float, float) {
 %3 = alloca float
 %4 = alloca float
 store float %0, float* %3
 store float %1, float* %4
 %5 = load float, float* %3
 %6 = load float, float* %4
 %7 = load float (float, float)*, float (float
   , float)** getelementptr inbounds (
 ...
 @_vfc_current_mca_interface, i32 0, i32 0)
 %8 = call float %7(float %5, float %6)
 ret float %8
}
```

FIGURE 3.6 – Résultat de l'instrumentation de l'addition flottante au registre %7 sur le code de gauche par deux instructions à droite : %7 qui charge la fonction et %8 qui l'appel sur les arguments d'origines.

```
%18 = load float*, float** %3           ;
%19 = load i32, i32* %9              ;
%20 = sext i32 %19 to i64             ;
%21 = getelementptr float, float* %18, i64 %20 ;
%22 = load float, float* %21          ;
%23 = load float, float* %6           ;
%24 = fsub float %22, %23            ;
store float %24, float* %7           ; y = f[i] - sum
%25 = load float, float* %           ;
%26 = load float, float* %7          ;
%27 = fadd float %25, %26            ;
store float %27, float* %8           ; t = sum + y
%28 = load float, float* %8          ;
%29 = load float, float* %5          ;
%30 = fsub float %28, %29            ;
%31 = load float, float* %7          ; y = t - sum
%32 = fsub float %30, %31            ;
store float %32, float* %6           ; c = (t - sum) - y
%33 = load float, float* %8          ;
store float %33, float* %5           ; sum = t
br label %34
```

FIGURE 3.7 – Coeur de la boucle principale dans la sommation de Kahan compilée sans optimisations avec `clang 6.0.0`

```
%17 = phi i64 [ 0, %9 ], [ %29, %16 ]         ; i = 0
%18 = phi <4 x float> [ %15, %9 ], [ %27, %16 ]   ; sum1 = f[0:4]
%19 = phi <4 x float> [ zero, %9 ], [ %28, %16 ]  ; sum2 = 0
%20 = or i64 %17, 1                  ;
%21 = getelementptr float, float* %0, i64 %20     ;
%22 = bitcast float* %21 to <4 x float>*       ;
%23 = load <4 x float>, <4 x float>* %22       ; x1 = f[i : i + 3]
%24 = getelementptr float, float* %21, i64 4     ;
%25 = bitcast float* %24 to <4 x float>*       ;
%26 = load <4 x float>, <4 x float>* %25       ; x2 = f[i + 4 : i + 7]
%27 = fadd fast <4 x float> %23, %18          ; sum1 = x1 + sum1
%28 = fadd fast <4 x float> %26, %19          ; sum2 = x2 + sum2
%29 = add i64 %17, 8                  ; j = j + 8
%30 = icmp eq i64 %29, %13               ;
br i1 %30, label %31, label %16           ;
```

FIGURE 3.8 – Coeur de la boucle principale dans la sommation de Kahan compilée avec optimisations agressives (`-O3 -ffast-math`) avec `clang 6.0.0`

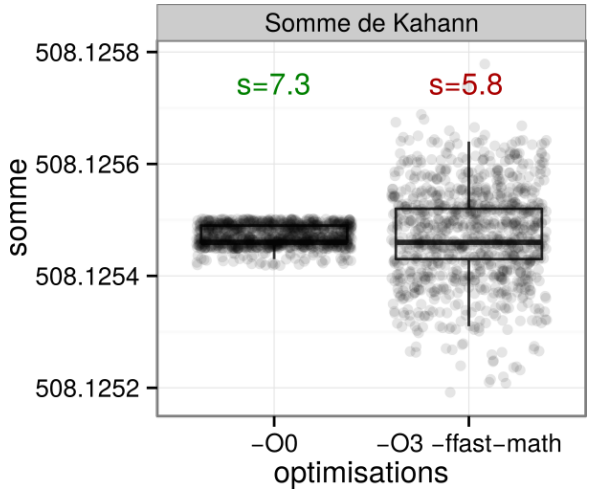


FIGURE 3.9 – Effet des optimisations de compilation sur la justesse des résultats de la somme de Kahan. Chaque point est une réalisation MCA représentant la valeur de la somme sur 100000 éléments binary32 en mode RR et précision virtuelle $t = 24$. Les 1000 réalisations tirées à droite montre une plus grande dispersion qu'à gauche illustrant la perte de justesse occasionnée par les optimisations. On retrouve cette variance des résultats dans le nombre de chiffres significatifs s avec la définition 2.6.1.

3.2 Les backends

Nous allons ici présenter trois des quatre backends disponibles dans la version actuelle de Verificarlo. Pour les backends MCA, bien qu'en théorie les calculs intermédiaires doivent être faits en précision infinie, l'utilisation de deux fois la précision de travail s'avère être en pratique suffisante [145, p38]. Le backend `VPREC` permettant de simuler des précisions variables sera présenté dans la section 5.1.2 du chapitre 5.

3.2.1 `MPFR`

Le backend `MPFR` est l'implémentation de référence du modèle MCA inspirée de la librairie MCALIB [68]. Il doit son nom à la librairie de calcul en multi-précision arbitraire `MPFR` [67] qu'il utilise. Ce backend remplace chaque opération flottante par son équivalent dans l'arithmétique de Monte-Carlo. La figure 3.11 présente l'implémentation d'origine de la fonction `_mca_inexact` qui ajoute le bruit stochastique à une valeur flottante `a`. Pour cela nous calculons l'exposant de la variable `a` (ligne 5) que nous utilisons pour calculer la magnitude du bruit qui sera ajouté à `a` (ligne 8). Le bruit dépend de la précision virtuelle utilisée (`MCALIB_T`). On utilise la variable `mpfr_rand` pour stocké ce bruit qui sera ajouté à la valeur flottante (ligne.23).

Optimisations proposées

Le backend `MPFR` est le premier backend à avoir été implémenté dans Verificarlo et est aujourd'hui considéré comme une implémentation de référence. Inspiré de l'implémentation de MCALIB, il souffre d'un surcoût conséquent variant d'un facteur $\times 100$ à $\times 1000$ suivant les cas dû à l'utilisation intensive de la librairie `MPFR`. Nous avons profilé le backend à l'aide de l'outil maqao [59] pour trouver les points chauds, c'est-à-dire les endroits dans lesquels une application passe la majeure partie de son temps, pour ensuite les optimiser.

Le tableau 3.1 montre les quatre fonctions les plus chaudes durant l'exécution du code 3.10. L'implémentation d'origine passe la majeur partie de son temps dans l'initialisation des variables `mpfr` avec la fonction `mpfr_set_d`. En réduisant le nombres de variables `mpfr` et donc leur initialisation, on améliore le temps passé dans le backend. De plus, cela diminue également le temps passé dans `__tls_get_addr`, une fonction de noyau Linux qui renvoie l'adresse d'une variable locale à un thread. En effet, la plupart des fonctions de la librairie `MPFR` sont *thread-safe* donc les variables locales aux fonctions sont déclarées dans la mémoire locale au thread (TLS).

La seconde partie couvrant la majorité du temps d'exécution est le calcul du bruit stochastique $2^{e_x - t}\xi$ avec les fonctions `__ieee754_pow_sse2` et `mpfr_mul` liées aux lignes 19 et 22 sur la figure 3.11, ainsi que les initialisations correspondantes (lignes 20 et 21). En remplaçant ce calcul par la fonction `mpfr_mul_2si`, on économise l'appel de la puissance qui est une fonction coûteuse ainsi que la variable `offset`.

Les optimisations apportées ont permis de diviser le temps d'exécution par $\times 2,5$ en moyenne. Le détail des facteurs d'accélération est présenté table 3.3.

3.2.2 `QUAD`

Le backend `MPFR` facilite l'implémentation du modèle MCA grâce à la librairie `MPFR` qui fourni l'ensemble des fonctions pour calculer et arrondir correctement les opérations flottantes pour des

```
for (i = 0 ; i < ITER; i++) {
  for (j = 0; j < SAMPLES; j++) {
    data[j] = a OPERATE b;
  }
}
```

FIGURE 3.10 – Code pour mesurer les performances des optimisations apportées au backend MPFR. Les paramètres choisis sont ITER=5000 et SAMPLES=10000. La variable OPERATE est parmi les quatre opérations de base $(+, -, *, /)$.

Fonction	Librairie	Couverture (%)	Temps (sec)
mpfr_set_d	libmpfr.so.4.1.4	15,71	12,10
mpfr_mul	libmpfr.so.4.1.4	8,15	6,24
__ieee754_pow_sse2	libm-2.23.so	7,84	6,04
__tls_get_addr	ld-2.23.so	6,70	5,16
mpfr_init2	libmpfr.so.4.1.4	5,19	4,00
int_free	libc-2.23.so	4,98	3,84
__exp1	libm-2.23.so	4,91	3,78
Total	-	100	77,04

TABLE 3.1 – Profil des fonctions couvrant plus de 50% du temps d'exécution dans la version d'origine du backend MPFR. Une part importante est passée dans l'initialisation des variables et le calcul de puissance. Code (figure 3.10) compilé avec `verificarlo -O0 -DSAMPLES=10000 -DITER=5000 -DOPERATE=+` et exécuté avec `VERIFICARLO_PRECISION=53` et `VERIFICARLO_MCAMODE=MCA` sur la machine sur une machine 8 coeurs Intel Xeon E31245 cadencé à 3.7GHz disposant de 16Go de RAM.

Fonction	Librairie	Couverture (%)	Temps (sec)
mpfr_set_d	libmpfr.so.4.1.4	19,98	4,72
_mca_inexact	libmcampfr.so.0.0.0	10,67	2,52
mpfr_add1sp	libmpfr.so.4.1.4	10,58	2,50
__tls_get_addr	ld-2.23.so	9,40	2,22
Total	-	100	24,58

TABLE 3.2 – Profil des fonctions couvrant plus de 50% du temps d'exécution dans la version optimisée du backend MPFR. La réduction du nombre de variables intermédiaires ainsi que des fonctions regroupant plusieurs opérations permettent de réduire le temps d'exécution. Code (figure 3.10) compilé avec `verificarlo -O0 -DSAMPLES=10000 -DITER=5000 -DOPERATE=+` et exécuté avec `VERIFICARLO_PRECISION=53` et `VERIFICARLO_MCAMODE=MCA` sur une machine 8 coeurs Intel Xeon E31245 cadencé à 3.7GHz disposant de 16Go de RAM.

```
static int _mca_inexact(mpfr_ptr a, mpfr_rnd_t rnd_mode) {
 if (MCALIB_OP_TYPE == MCAMODE_IEEE) {
  return 0;
 }
 mpfr_exp_t e_a = mpfr_get_exp(a) - 1;
 mpfr_prec_t p_a = mpfr_get_prec(a);
 mpfr_t mpfr_rand, mpfr_offset, mpfr_zero;
 e_a = e_a - (MCALIB_T - 1);
 mpfr_inits2(p_a, mpfr_rand, mpfr_offset, mpfr_zero, (mpfr_ptr)0);
 mpfr_set_d(mpfr_zero, 0., rnd_mode);
 int cmp = mpfr_cmp(a, mpfr_zero);
 if (cmp == 0) {
  mpfr_clear(mpfr_rand);
  mpfr_clear(mpfr_offset);
  mpfr_clear(mpfr_zero);
  return 0;
 }
 double d_rand = (_mca_rand() - 0.5);
 double d_offset = pow(2, e_a);
 mpfr_set_d(mpfr_rand, d_rand, rnd_mode);
 mpfr_set_d(mpfr_offset, d_offset, rnd_mode);
 mpfr_mul(mpfr_rand, mpfr_rand, mpfr_offset, rnd_mode);
 mpfr_add(a, a, mpfr_rand, rnd_mode);
 mpfr_clear(mpfr_rand);
 mpfr_clear(mpfr_offset);
 mpfr_clear(mpfr_zero);
}
```

FIGURE 3.11 – Backend MPFR original.

```
static int _mca_inexact(mpfr_ptr a, mpfr_rnd_t rnd_mode) {
 /* if we are in IEEE mode, we return a noise equal to 0 */
 /* if a is NaN, Inf or 0, we don't disturb it */
 if ((MCALIB_OP_TYPE == MCAMODE_IEEE) || (mpfr_regular_p(a) == 0)) {
  return 0;
 }
 /* get_exp reproduce frexp behavior, */
 /* i.e. exp corresponding to a normalization in the interval [1/2 1[ */
 /* remove one to normalize in [1 2[ like ieee numbers */
 mpfr_exp_t e_a = mpfr_get_exp(a) - 1;
 mpfr_prec_t p_a = mpfr_get_prec(a);
 MPFR_DECL_INIT(mpfr_rand, p_a);
 e_a = e_a - (MCALIB_T - 1);
 double d_rand = (_mca_rand() - 0.5);
 mpfr_set_d(mpfr_rand, d_rand, rnd_mode);
 /* rand = rand * 2 ^ (e_a) */
 mpfr_mul_2si(mpfr_rand, mpfr_rand, e_a, rnd_mode);
 mpfr_add(a, a, mpfr_rand, rnd_mode);
}
}
```

FIGURE 3.12 – Backend MPFR optimisé.

Format	Mode	Acceleration
binary32	RR	1,69
binary32	PB	2,24
binary32	MCA	2,51
binary64	RR	2,05
binary64	PB	2,72
binary64	MCA	3,07

TABLE 3.3 – Principaux facteurs d'accélération entre le backend `MPFR` d'origine (fig. 3.1) et la version optimisée (fig. 3.2) mesurés sur l'exécution du code de la figure 3.10. Les facteurs correspondent à la moyenne des trois accélérations obtenues sur chaque opération $(+, *, /)$.

precisions arbitraires. Cependant, ce backend souffre d'un ralentissement conséquent pour une utilisation HPC.

Pour accélérer l'ajout du bruit stochastique, le backend `QUAD` utilise des nombres flottants du standard IEEE-754. En effet, l'utilisation de types natifs est plus rapide que ceux de librairies en précision arbitraire comme `MPFR`. De plus, comme il suffit en pratique du double de la précision de travail pour effectuer les calculs intermédiaires, les calculs pour le binary32 peuvent être faits sur du binary64 ($2 \times 23 < 52$) et ceux du binary64 sur du binary128 ($2 \times 52 < 112$). Si les formats binary32 et binary64 sont disponibles sur la plupart des CPUs modernes, aucuns d'entre eux ne proposent de binary128 de manière native pour le moment. Il est donc nécessaire de l'émuler. Nous utilisons pour cela le type `__float128` de la GCC *Quad-Precision Math Library* qui est représenté sur deux mots de 64 bits. Cette bibliothèque fourni les opérations arithmétiques de base ainsi que les fonctions mathématiques standards de la `libm`. L'utilisation des `__float128` requiert une précaution puisque leur mantisse est à cheval sur deux mots mémoire. La gestion des dénormalisés nécessite donc plusieurs opérations pour décaler la mantisse à la bonne place suivant la précision virtuelle utilisée comme illustrée sur la figure 3.13.

3.2.3 `BITMASK`

Le backend `BITMASK` offre une approximation grossière de MCA en utilisant des masques de bits pour introduire des perturbations dans les calculs flottants. Il a été construit pour évaluer rapidement des applications HPC en présence de bruit mais ajoute cependant un bruit biaisé ce qui ne le rend pas propice à l'étude des propriétés arithmétiques fines comme le modèle MCA dans lequel l'erreur introduite est non biaisée (cf. 2.6.1). `BITMASK` offre cependant une bonne approximation comme nous le verrons section 3.2.3.

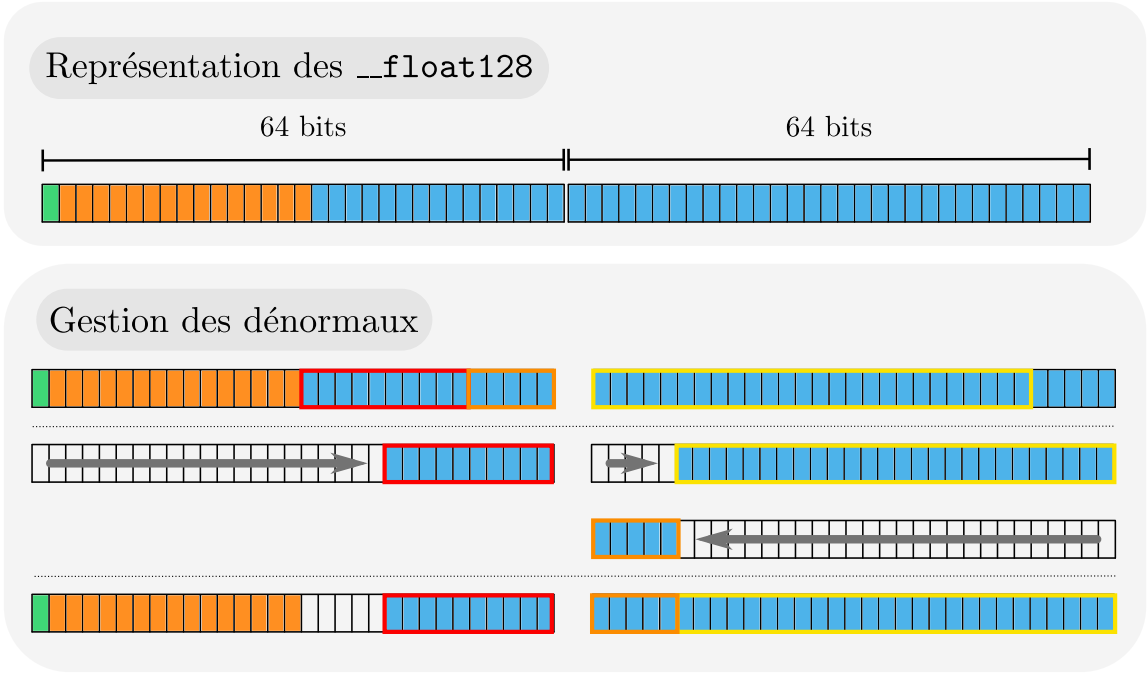


FIGURE 3.13 – Implémentation des types `__float128` de GCC construits sur deux mots de 64 bits. La mantisse necessite deux entiers de 64 bits puisqu'elle est codée sur 112 bits. Cela nécessite un traitement particulier comme pour la gestion des dénormalisés.

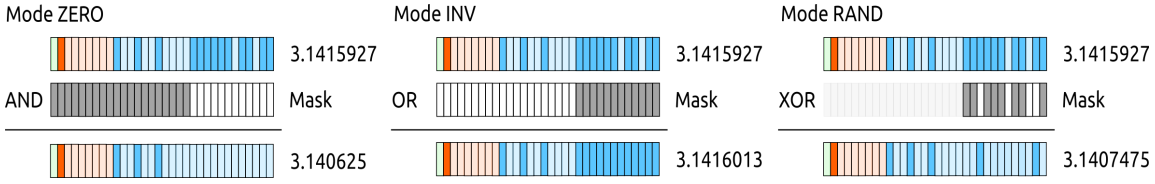


FIGURE 3.14 – Modes `BITMASK` avec en opaque les bits à 1 et en transparent les bits à 0. Pour le mode `RAND`, on applique uniquement le XOR sur la partie opaque de la mantisse.

Le backend BITMASK dispose de quatre modes de perturbation :

- **IEEE** : Exécute l'opération comme en IEEE-754.
- **ZERO** : Applique un masque de bits sur la mantisse avec un AND mettant les bits de poids faible supérieurs à la précision virtuelle à 0. Si x est un flottant de mantisse $m = m_0 m_1 ... m_{p-1}$ alors ZERO(x) :

	m_0	m_1	...	m_{t-2}	m_{t-1}	m_t	...	m_{p-1}
AND	1	1	...	1	0	0	...	0
	m_0	m_1	...	m_{t-2}	0	0	...	0

  Cela revient à un arrondi vers zero à précision t.
  $ZERO(x \diamond y) = \bowtie_t (x \diamond y)$
- **ONE** : Applique un masque de bits sur la mantisse avec un OR mettant les bits de poids faible supérieurs à la précision virtuelle à 1.

	m_0	m_1	...	m_{t-2}	m_{t-1}	m_t	...	m_{p-1}
OR	0	0	...	0	1	1	...	1
	m_0	m_1	...	m_{t-2}	1	1	...	1

- **RAND** : Applique un masque de bits aléatoire avec un XOR sur la mantisse sur les bits de poids faible supérieurs à la précision virtuelle. On note X les bits aléatoires et $Y_i = X_i \oplus m_i$ le résultat du XOR.

	m_0	m_1	...	m_{t-2}	m_{t-1}	m_t	...	m_{p-1}
AND	0	0	...	0	1	1	...	1
XOR	0	0	...	0	X_{t-2}	X_{t-1}	...	X_{p-1}
OR	m_0	m_1	...	m_{t-2}	0	0	...	0
	m_0	m_1	...	m_{t-2}	Y_{t-2}	Y_{t-1}	...	Y_{p-1}

Performance

Le backend BITMASK est très léger puisque l'exécution du code 3.10 ne prend que 4,92 secondes en moyenne contre 1,46 secondes pour le code original, soit un ralentissement de 3,37.

Biais statistique

Le backend BITMASK souffre d'un biais statistique pour les trois modes dont il dispose. Un backend biaisé signifie qu'en moyenne l'erreur introduite dans les opérations flottantes n'est pas nulle. Par conséquent l'espérance du résultat bruité est différente du résultat exacte. Pour rappel, nous avons montré à la section 2.6.1 que l'erreur ajoutée dans le modèle MCA est non-biaisée puisque :

$$inexact(x) = x + \beta^{e_x - t}\xi,\ \xi \in \left]-\tfrac{1}{2}, \tfrac{1}{2}\right[$$

conduit à

$$\mathbb{E}[x + \beta^{e_x - t}.\xi] = \mathbb{E}[x] + \beta^{e_x - t}\mathbb{E}[\xi] = x \ \texttt{car}\ \mathbb{E}[\xi] = 0.$$

Le mode ZERO en tronquant au t-ème bit est équivalent à un arrondi vers zéro $ZERO_t(x \diamond y) = \bowtie_t (x \diamond y)$. Pour le mode ONE puisque l'on introduit des 1 à partir de la position t, on peut représenter comme un arrondi vers zéro plus un terme $c_t = \sum_{i=t}^{p-1} 2^{-i} = 2^{-t+1} - 2^{-p+1}$ où t est la précision virtuelle et p la précision du format d'origine : $ONE_t(x \diamond y) = \bowtie_t (x \diamond y) * (1 + c_t)$. Pour une précision virtuelle $t \geqslant p$ donnée, ces deux modes encadrent l'arrondi au plus près :

```c
int main() {
  float somme = 0.0f;
  for (int i = 0; i < n; i++) {
    somme += 0.1f;
  }
  printf("somme = %.6a\n", somme);
  return 0;
}
```

FIGURE 3.15 – Code de la somme de 0,1 répété n fois. Compilé avec `verificarlo -O0` et exécuté avec `VERIFICARLO_PRECISION=24` et `VERIFICARLO_MODE=RR`.

$$|ZERO_t(x)| \leqslant |\circ_t(x)| \leqslant |ONE_t(x)|$$
$$|\bowtie_t(x)| \leqslant |\circ_t(x)| \leqslant |\bowtie_t(x)(1+c_t)|.$$

La figure 3.16 compare ces deux modes d'arrondi avec le mode d'arrondi au plus près pour le code 3.15. On remarque que l'accumulation du biais introduit par ces deux modes tend à dévier le résultat calculé du résultat en arrondi au plus près.

Les modes `ZERO` et `ONE` sont donc des modes non-aléatoires biaisés qui orientent l'erreur d'arrondi toujours dans la même direction. Le mode `RAND` pour sa part repose sur un tirage aléatoire dans la construction du masque à appliquer. Cependant, puisque nous utilisons un XOR comme opération pour appliquer le masque, l'erreur introduite dépend du nombre de 0 dans la mantisse du nombre à bruiter. En effet, si l'on prend le cas extrême lorsque la mantisse ne possède que des 0, l'erreur introduite ne sera que positive et fera augmenter le nombre. A l'inverse dans le cas ou la mantisse ne possède que des 1, l'erreur introduite sera négative et fera diminuer le nombre. En effet, si on suppose que X est la variable aléatoire représentant le masque tiré aléatoirement et que $x \in \mathbb{F}_p$ est la variable à bruiter à la précision virtuelle t telles que :

$$x = (-1)^{-s}.2^e(1+m), m = \sum_{i=1}^{p-1} m_i 2^{-i}, m_i \in \{0,1\}$$
$$X = (-1)^{-s}.2^e(1+M), M = \sum_{i=1}^{p-1} M_i 2^{-i}, M_i \in \{0,1\}.$$

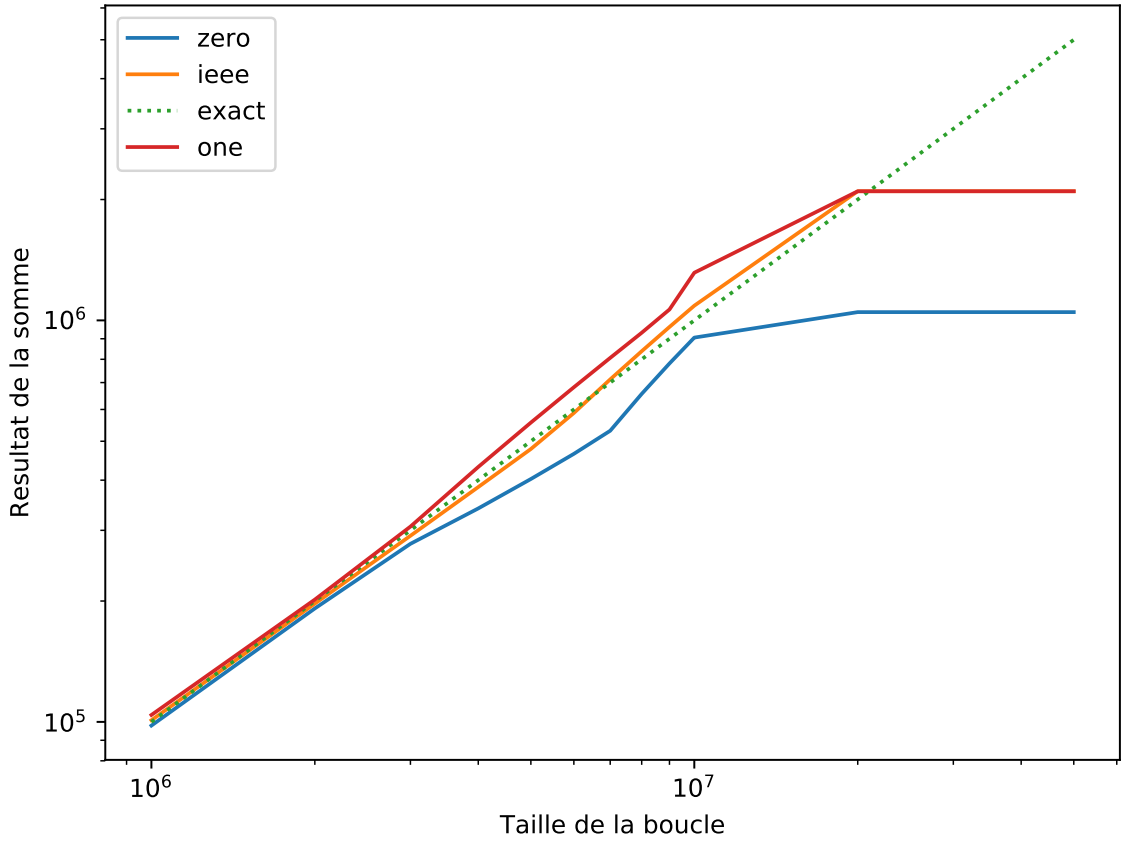


FIGURE 3.16 – Illustration du biais présent dans les modes ZERO et ONE du backend `BITMASK`. Le code testé est présenté sur la figure 3.15. Les modes ZERO (bleu) et ONE (rouge), par le biais qu'il introduise tendent à s'éloigner du résultat IEEE-754 avec arrondi au plus près (jaune). Le mode d'arrondi au plus près permet d'avoir un résultat plus proche du résultat exact (vert) que les modes du backend `BITMASK` qui orientent le résultat toujours dans le même sens. Le plateau final est du à une absorption de 0,1 par l'accumulateur de la somme.

x	0	0	1	1
X	0	1	0	1
$x \oplus X$	0	1	1	0
$(x \oplus X) - x$	0	1	0	-1

TABLE 3.4 – Table des gains de l'opération $x \oplus X$

on définit l'opérateur XOR $\oplus$ de la manière suivante :

$$\begin{aligned} x \oplus_t X &= (-1)^s 2^e \left(1 + \sum_{i=1}^{p-1} (M_i \oplus m_i) 2^{-i}\right) \\ &= (-1)^s 2^e \left(1 + \sum_{i=1}^{t-1} (M_i \oplus m_i) 2^{-i}) + \sum_{i=t}^{p-1} (M_i \oplus m_i) 2^{-i}\right) \\ &= (-1)^s 2^e \left(1 + \sum_{i=1}^{t-1} m_i 2^{-i} + \sum_{i=t}^{p-1} (M_i \oplus m_i) 2^{-i}\right). \end{aligned}$$

On définit $\mathcal{C}_k$ l'ensemble de tous les nombres flottants ayant pour mantisse k bits égales à 1. Nous allons montrer que l'erreur introduite est toujours positive (respectivement négative) pour $x \in \mathcal{C}_0 \implies |x \oplus X| \geqslant |x|$ (respectivement $x \in \mathcal{C}_p \implies |x \oplus X| \leqslant |x|$).

Pour cela, calculons l'erreur moyenne pour un bit situé à la place i à partir de la table 3.4.

$$\begin{aligned} \mathbb{E}[(m_i \oplus M_i) - m_i] &= (0.\mathbb{P}[m_i \oplus M_i = 0 | m_i = 0] + 1.\mathbb{P}[m_i \oplus M_i = 1 | m_i = 0] \\ &\quad + 0.\mathbb{P}[m_i \oplus M_i = 1 | m_i = 1] - 1.\mathbb{P}[m_i \oplus M_i = 0 | m_i = 1]) 2^{-i} \\ \mathbb{E}[(m_i \oplus M_i) - m_i] &= (\mathbb{P}[m_i \oplus M_i = 1 | m_i = 0] - \mathbb{P}[m_i \oplus M_i = 0 | m_i = 1]) 2^{-i}. \end{aligned}$$

On voit que l'erreur moyenne introduite pour un bit dépend du bit m_i du nombre d'origine. Analysons maintenant l'erreur de l'opération $(|x| \oplus X) - |x|$ On prend la valeur absolue de $|x|$ pour montrer le signe de l'erreur moyenne introduite.

$$
\begin{aligned}
\mathbb{E}[(|x| \oplus_t X) - |x|] &= |(-1)^s 2^e (1 + (M \oplus m))| - |(-1)^s 2^e (1 + m)| \\
&= 2^e((M \oplus m)) - m) \\
&= 2^e \left(\sum_{i=1}^{t-1} \underbrace{(M_i \oplus m_i) - m_i}_{=0 \Longleftarrow t \geqslant p} - \sum_{i=t}^{p-1} (M_i \oplus m_i) - m_i 2^{-i} \right) \\
&= 2^e \left(\sum_{i=t}^{p-1} \left(\underbrace{\mathbb{P}[(M_i \oplus m_i) = 1 | m_i = 0]}_{=\frac{1}{2}} - \underbrace{\mathbb{P}[(M_i \oplus m_i) = 0 | m_i = 1]}_{=0 \Longleftarrow x \in \mathcal{C}_0} \right) 2^{-i} \right) \\
&= 2^e \left(\sum_{i=t}^{p-1} \tfrac{1}{2} 2^{-i} \right) \\
&= 2^e \left(2^{-t} - 2^{-p} \right) \\
\mathbb{E}[(|x| \oplus_t X) - |x|] &\geqslant 0.
\end{aligned}
$$

Pour le cas $x \in \mathcal{C}_p$, il suffit de voir que la probabilité $\mathbb{P}[(M_i \oplus m_i) = 1 | m_i = 0] = 0$ car $x \in \mathcal{C}_p$. On a donc que :

$$
\begin{aligned}
\mathbb{E}[(|x| \oplus_t X) - |x|] &= 2^e \left(- \sum_{i=t}^{p-1} \tfrac{1}{2} 2^{-i} \right) \\
&= -2^e \left(2^{-t} - 2^{-p} \right) \\
\mathbb{E}[|x| \oplus_t X - |x|] &\leqslant 0.
\end{aligned}
$$

On voit que le biais introduit est étroitement relié à la proportion de 0 et de 1 dans la mantisse du nombre à bruiter. Nous avons comparé l'erreur introduite par les modes `MPFR` et `QUAD` du backend MCA et le mode `RAND` du backend `BITMASK` en fonction du nombre de 0 dans la mantisse du flottant bruité en calculant l'erreur absolue mise à l'échelle η :

$$
\eta = \frac{\tilde{x} - x}{\beta^{\lfloor \log_\beta |x| \rfloor + 1}} = 2^{-t} \epsilon.
$$

En divisant par 2^{-t}, on obtient le bruit ajouté ϵ. La figure 3.17 montre le biais moyen en fonction du nombre de 0 dans la mantisse du nombre bruité. Contrairement au backend `MPFR` et `QUAD`, l'erreur du backend `BITMASK` en mode `RAND` évolue en fonction du nombre de 0 dans la mantisse du nombre bruité ce qui confirme le biais de ce mode.

3.3 L'application aux codes industriels

Verificarlo est une boîte à outils pour le calcul numérique : ses backends permettent différentes analyses comme la détection d'instabilités numériques avec les backends MCA ou l'exploration de

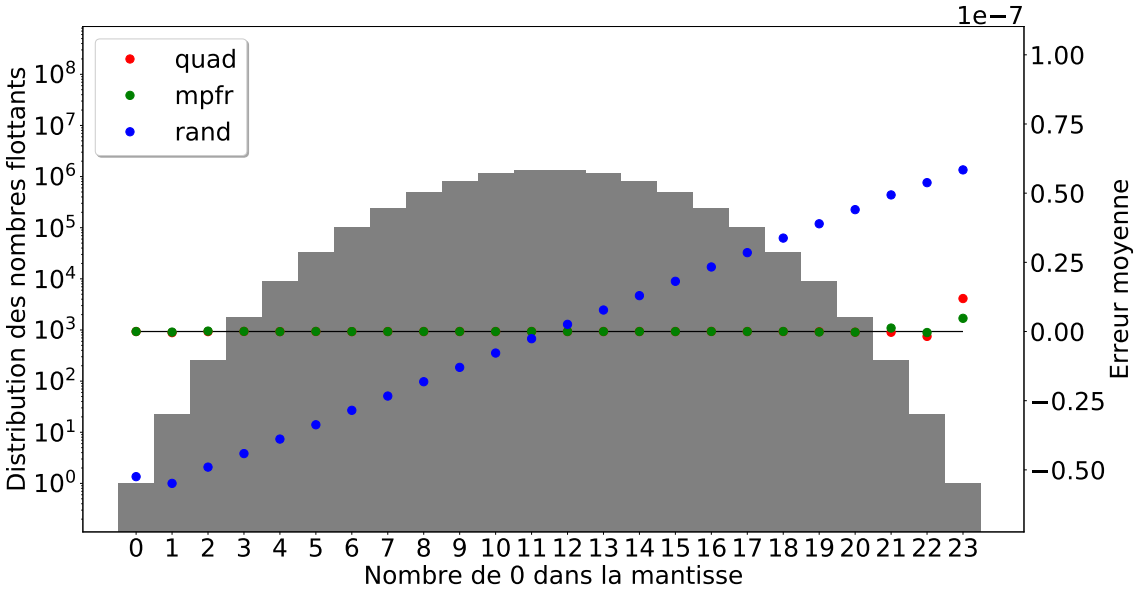


FIGURE 3.17 – Erreur moyenne introduite en fonction du nombre de 0 dans la mantisse du nombre bruité. Contrairement aux backends `MPFR` et `QUAD`, le mode `RAND` du backend `BITMASK` est biaisé et varie suivant le nombre bruité.

formats réduits avec le backend `VPREC`. Cependant, il existe un écart important entre ce que souhaite l'utilisateur comme informations pertinentes et ce que lui fournit l'utilisation des backends. Il est en effet intéressant pour l'utilisateur de savoir d'où provient une erreur numérique ou quelles parties de son code peuvent bénéficier d'un format de calcul plus petit.

Or retrouver ces informations nécessite, par exemple, de tracer la justesse des calculs intermédiaires pour détecter une instabilité ou explorer les différents formats possibles pour chacune de instructions flottantes dans le cadre d'une optimisation. On comprend alors intuitivement que tracer ou explorer chacune des instructions flottantes sur des codes industriels contenant plusieurs centaines de millier, voire des millions de lignes de code peut vite s'avérer être impossible en pratique du à l'explosion combinatoire.

D'autre part, dans le cadre de l'optimisation, changer de format flottant induit un changement sur les résultats puisque la précision ou la dynamique n'est plus la même (cf. 5.3). De plus, les codes de calcul que l'on souhaite aborder peuvent contenir plusieurs résultats importants d'un point vue scientifique. Fournir une borne sur l'erreur commise lors du passage en précision plus faible est difficile sans avoir recours aux méthodes d'analyses formelles. Mais ces méthodes passent mal à l'échelle ce qui rend leur utilisation difficile dans un contexte industriel. C'est pourquoi, il est nécessaire d'avoir un critère permettant de valider si la solution calculée est acceptable ou non.

La difficulté d'analyser des codes industriels peut se résumer suivant les trois points suivants :

- Crash : L'introduction de bruit aléatoire peut conduire à des comportements imprévus par le programme original ce qui provoque un arrêt brutal. Sans outils pour rapidement déceler les instructions responsables du crash, l'utilisation de Verificarlo est difficile.
- Explosion combinatoire : Que ce soit pour la détection de bugs numériques ou l'optimisation de

format de calcul, le nombre d'états possibles à analyser est grand. L'approche naïve consistant à explorer chacun des états est donc à proscrire.

- Dimensions : L'analyse peut se faire suivant différents points de vue : spatial ou temporel. Chacune de ces dimensions requiert une approche spécifique. De plus, l'exploration conjointe de ces différentes dimensions accroît d'autant plus l'espace de recherche.

C'est pourquoi l'utilisation de techniques de réduction de dimensionnalité est important pour une utilisation efficace de Verificarlo. Nous allons dans cette section présenter les dimensions d'espace et de temps qui nous semblent importantes à explorer lorsqu'on s'intéresse au débugage numérique ou à l'optimisation des formats de calcul. Puis, nous définirons la notion de critère de validité qui nous sera utile pour pouvoir juger de la qualité d'un résultat ou d'une solution calculée par un code bruité par nos soins. Il est en effet difficile de pouvoir estimer si une solution est suffisamment exacte sans une connaissance du domaine scientifique dans lequel la simulation opère. Enfin nous présenterons des techniques de réduction de dimensionnalité qui ont été adaptées aux calculs flottants.

3.3.1 Les dimensions

La dimension spatiale

La dimension spatiale recouvre tout les aspects liés à la localité dans un code source. Ainsi, déterminer quelle section d'un code est responsable d'une erreur numérique sur le résultat ou quelle section peut être exécutée avec moins de précision, relève d'un problème de spatialité. La taille de ces sections définit la résolution spatiale. Un code ou une application $\mathcal{A}$ est organisé suivant la hiérarchie suivante : des modules m qui rassemblent plusieurs fonctions f. Chacune de ces fonctions est constituée de plusieurs lignes de code l qui sont chacune traduites en une ou plusieurs instructions assembleurs i :

$$\mathcal{A} = \bigcup_{m \in \mathcal{M}} m, \quad \mathcal{M} = \bigcup_{f \in \mathcal{F}} f, \quad \mathcal{F} = \bigcup_{l \in \mathcal{L}} l, \quad \mathcal{L} = \bigcup_{i \in \mathcal{I}} i, \quad i = instruction.$$

Définition 3.3.1 (Résolution spatiale)**.** On définit la résolution spatiale $\mathcal{R}_\mathcal{S} \in \{\mathcal{M}, \mathcal{F}, \mathcal{L}, \mathcal{I}\}$ comme le type minimal d'une section. La plus petite résolution est l'instruction assembleur et la plus grande le module. En se plaçant au niveau des fonctions, on raisonnera alors sur les fonctions et non sur les niveaux plus bas des lignes de code ou des instructions assembleurs.

La résolution définit la taille notée $|.|$ de l'ensemble de recherche $\mathcal{S}$ avec :

$$|\mathcal{S}| = \mathcal{P}(\mathcal{R}_\mathcal{S}) = 2^{\mathcal{R}_\mathcal{S}}.$$

Ainsi, plus la résolution sera fine et plus l'espace de possibilités $\mathcal{S}$ sera grand. On comprend que $\mathcal{S}$ croît très rapidement et que le choix d'une résolution très fine requiert des algorithmes pour explorer rapidement le très grand nombre de possibilités. Même à une grosse granularité comme celle de la fonction, pour un code contenant environ 300 fonctions, ce n'est pas moins de 2^{300} états qu'il faut visiter soit plus du nombre d'atomes dans l'univers.

La dimension temporelle

La dimension temporelle décrit l'évolution d'un élément spatial $x \in \mathcal{R}_\mathcal{S}$ au cours du temps. L'élément de base de la dimension temporelle est l'invocation qui décrit la n-ème apparition d'un élément spatiale au cours du temps :

Définition 3.3.2 (Invocation)**.** On définit x_i comme la i-ème invocation d'un élément spatial $x \in \mathcal{R}_\mathcal{S}$:

$$x_i \equiv (x, i) \in \mathcal{R}_\mathcal{S} \times \mathbb{N}.$$

La temporalité est un aspect important lorsque l'on s'intéresse au comportement numérique d'un code. En effet, d'un point de vue spatial, une fonction ou une instruction assembleur représente un point dans l'espace, peu importe qu'il soit invoqué une ou plusieurs fois au cours du temps. De même, lorsque l'on s'intéresse à la réduction de format, la précision requise au cours du temps par une section de calcul diffère au fur et à mesure du temps. Les algorithmes itératifs fonctionnent sur un raffinement de la solution au cours du temps ce qui nécessite de plus en plus de précision. La méthode de Newton-Raphson en est un bel exemple qui sera présenté dans la section 5.2.

On utilisera la notation suivante pour parler des i-èmes invocations d'un sous-ensemble $X \in \mathcal{R}_\mathcal{S}$:

$$\{X\}_i \iff \{x_j \in X \mid i = j\}$$

où bien la notation suivante pour évoquer toutes les invocations :

$$\{X\}_{i \geqslant 0} \iff \{x_i \in X \mid i \geqslant 0\}.$$

3.4 Les critères de validité

Verificarlo n'est pas un outil reposant sur des analyses formelles. Ainsi, lorsque l'on souhaite l'utiliser de manière automatique pour débusquer des bugs ou réduire le format de calcul, on est obligé de recourir à des heuristiques permettant de réduire la dimension des espaces de recherche qui sont associés. Néanmoins, pour guider notre heuristique dans l'espace d'optimisation, nous devons nous donner des critères permettant de valider ou rejeter les solutions trouvées durant l'exploration. On définit de manière assez générale un critère de validité c comme un test qui rejette ou accepte une hypothèse $\mathcal{H}$:

$$c : \mathcal{H} \to \{0, 1\}.$$

Nous distinguons deux types de critères : d'un côté les critères numériques qui s'attachent à vérifier que la qualité du résultat calculé n'est pas trop dégradée, c'est-à-dire quelle soit suffisamment juste pour l'expert du domaine et de l'autre les critères temporels qui vérifient que le nombres d'étapes nécessaires au bon déroulement du calcul soit en dessous d'un seuil, comme par exemple, pour s'assurer qu'une méthode itérative converge en un nombre maximum d'étapes.

3.4.1 Le critère numérique

Le critère de validité numérique permet de distinguer les bonnes des mauvaises solutions lorsqu'on cherche à débuguer ou optimiser un code. Par exemple, lorsqu'on introduit du bruit stochastique dans une fonction pour tester son impact sur le résultat d'un calcul, nous avons besoin de tests qui permettent de dire si l'erreur globale, c'est-à-dire l'erreur sur le résultat final, est acceptable ou pas. De même lorsque l'on s'intéresse à l'optimisation des formats de calcul, on souhaite avoir un test permettant de valider ou d'invalider une configuration du code qui utilise une précision réduite.

Cependant la définition d'un critère numérique n'est pas facile et dépend de plusieurs paramètres. Premièrement du domaine scientifique étudié, une simulation de chimie ne calcule pas les mêmes propriétés physiques qu'un code de dynamique des fluides. La nature des résultats n'est donc pas

la même. Deuxièmement, le résultat qui intéresse l'expert n'est pas toujours une grandeur scalaire dont on sait comparer à une référence (cf. section 2.4). En effet, lorsque le résultat est une grandeur vectorielle ou un objet mathématique plus complexe, il est plus difficile de le comparer à une référence. Le choix d'une norme pour comparer deux vecteurs reviendra donc à l'expert du domaine par exemple.

Pour ces raisons, nous estimons qu'il appartient à l'expert du domaine de fournir des tests permettant de valider ou d'invalider une solution calculée. Pour rester le plus général possible, nous définissons le critère de validité numérique comme une fonction qui à deux ensembles de valeurs flottantes donne une valeur réelle qui nous permet ensuite de comparer cette valeur à un seuil de tolérance. L'idée étant d'avoir une version générale de la fonction d'erreur relative (cf. définition 2.4.2). L'interprétation du seuil et de la comparaison est à la charge de l'expert.

Définition 3.4.1 (Critère de validité numérique)**.** Soient $\mathbb{F}_p$, $x \in \mathbb{F}_p^n$ une valeur de référence, $\tilde{x} \in \mathbb{F}_p^n$ le résultat d'une exécution bruitée d'un programme approchant x, $d : \mathbb{F}_p^n \times \mathbb{F}_p^n \to \mathbb{R}^+$ une distance entre x et $\tilde{x}$ et ϵ un seuil de validité. Alors, on définit donc un critère de validité numérique c_n comme :

$$c_n \equiv d(x, \tilde{x}) \leqslant \epsilon$$

Remarque : Pour utiliser l'erreur relative, if suffit de choisir $d(x, \tilde{x}) = \|x - \tilde{x}\|$ et $\epsilon = \|x\|\epsilon'$ de manière à avoir $d(x, \tilde{x}) \leqslant \epsilon$.

Ce critère de validité numérique permet de contrôler que l'*erreur globale*, c'est-à-dire que l'erreur commise sur le résultat final soit inférieure à un seuil de tolérance. On peut distinguer l'erreur globale de l'*erreur locale* qui est l'erreur commise localement à un calcul flottant. Cette erreur locale est intéressante lorsque l'on souhaite étudier la propagation des erreurs au cours des calculs [156]. Les erreurs commises localement n'ont cependant pas souvent de contribution majeure sur l'erreur globale [109].

3.4.2 Le critère temporel

Le critère de validité temporelle assure que l'exécution d'un programme modifié, i.e auquel on a rajouté du bruit numérique ou modifié le format de calcul, n'excède pas un certain temps d'exécution. En effet, lorsqu'on ajoute du bruit ou que l'on simule un format de calcul réduit, on influe, non seulement sur la qualité du résultat produit, mais également sur le temps d'exécution et ce dans les deux sens :

- Accélération : La réduction de précision permet par exemple, pour une même taille d'unité vectorielle, d'inclure plus d'éléments et donc accroître la densité de calcul (cf. 2.5.3)
- Ralentissement : Une précision trop faible conduit certains solveurs itératifs à converger longuement pour atteindre les critères de convergence (cf 5.2).

Ainsi, les critères de validité temporels sont utilisés pour garantir que le nombre d'invocations d'une section de code soit en dessous d'un certain seuil.

Définition 3.4.2 (Critère de validité temporelle)**.** Soient $k > 0$ le nombre d'unités spatiales x indicé par j à vérifier et x_{j_i} la i-ème invocation de x_j et n_j la contrainte sur x_j alors le critère de validité temporelle c_t est :

$$c_t \equiv \forall j, 1 \leqslant j \leqslant k, \max_{i \in \mathbb{N}} \{x_{j_i}\} \leqslant n_j.$$

Pour le cas de l'optimisation, les critères temporels servent à contraindre le code pour en contrôler le nombre d'itérations dans les cas de solveurs itératifs par exemple. En effet, l'optimisation d'algorithmes auto-correcteurs peut conduire à des solutions qui requièrent peu de précision mais énormément d'itérations. L'idée étant que ces algorithmes peuvent continuer à converger même en présence d'erreurs en itérant un peu plus sur les calculs. En introduisant des erreurs ou en utilisant une précision plus faible, on rallonge le nombre d'itérations nécessaires pour converger.

3.5 Les méthodes de réductions de dimensionnalité

Nous allons ici présenté trois méthodes de réduction de dimensionnalité qui ont été adaptées au cas du calcul flottant pour rapidement identifier des calculs ayant une instabilité numérique ou perturbant la qualité numérique d'un résultat par exemple.

3.5.1 Le Δ-debugging

Le Δ-debugging est un algorithme de recherche dichotomique proposé par Zeller [185] pour trouver un ensemble minimal de configurations responsable d'un bug. Une configuration C est un ensemble de changements par rapport à un code de référence, chaque configuration étant disjointe des autres. Par exemple, dans une utilisation avec Verificarlo, un changement équivaut à l'instrumentation d'une section de code $x \in \mathcal{R}_{\mathcal{S}}$ et une configuration est l'ensemble des parties du code qui ont été instrumentées.

À partir d'un code de référence stable, c'est-à-dire sans bugs et qui respecte des critères de validités (cf. 3.4), le Δ-debugging cherche à trouver la configuration contenant le minimum de changements qui fasse échouer les critères. Pour cela, le Δ-debugging utilise une recherche dichotomique : il commence par diviser l'espace des configurations initiales $C = c_1, c_2, ..., c_n$ en deux et teste l'une des partitions de configurations puis son complément. Si le test échoue sur l'une des partitions, l'ensemble minimal est trouvé. Sinon, on réitère le processus sur chacun des sous-ensembles et leur complément tant que les critères sont respectés. La figure 3.18 illustre son fonctionnement. Le Δ-debugging a une complexité en temps de $O(n \log n)$ en moyenne et $O(n^2)$ dans le pire cas qui arrive lorsque le test échoue avec une seule configuration.

Cet algorithme a été adapté pour le cas flottant afin de trouver l'ensemble de fonctions responsables d'une instabilité numérique [63], le changement étant ici l'introduction d'un bruit stochastique dans une fonction. Le Δ-debugging a également été testé pour des problèmes d'optimisation de précision [155, 75] avec pour changement le passage d'une fonction de binary64 en binary32.

3.5.2 *Shadow-memory* en multi-précision

La *shadow-memory* est une technique pour suivre la trace d'une information comme une variable ou une instruction durant l'exécution d'un processus en gardant un double de celle-ci en mémoire, ce qui permet de capter des informations supplémentaires. La shadow-memory est utile pour réduire le coût d'analyse puisqu'elle permet de se concentrer uniquement sur les calculs effectués par le programme et d'avoir le chemin exact des opérations effectuées. De plus, avoir la valeur des opérandes permet de calculer l'erreur potentielle commise durant l'opération flottante ou de comparer l'exécution du programme dans une précision de calcul plus grande. Il est ainsi possible de propager l'erreur locale à chaque opération au sein du code jusqu'à l'erreur globale de l'application, c'est-à-dire l'erreur mesurée sur le résultat final.

```
1  let
2  dd2(c, r) =
3    let c1,c2 ⊆ c with
4      c1 ∪ c2 = c, c1 ∩ c2 = ∅, |c1|≃|c2|≃|c|/2
5    in
6      if |c| = 1 else
7        (* Ensemble minimal trouvee *)
8        c
9      else if test(c1 ∪ r) = ✗ then
10       (* Echec dans la configuration c1 *)
11       dd2(c1, r)
12     else if test(c2 ∪ r) = ✗ then
13       (* Echec dans la configuration c2 *)
14       dd2(c2, r)
15     else
16       (* Cas d interference *)
17       dd2(c1, c2 ∪ r) ∪ dd2(c2, c1 ∪ r)
18 in
19   dd(c) = dd2(c, ∅)
20
```

FIGURE 3.18 – Algorithme du Δ-debugging dans sa forme simple. La fonction dd utilise la fonction $dd_2(c, r)$ où c est la configuration initiale et r l'ensemble des configurations valides. La récursion comporte quatre cas possibles : l.6) la configuration trouvée est de taille minimale. Dans ce cas l'algorithme s'arrête ; l.9) c'est la configuration c_1 qui échoue ou bien l.12) c'est la configuration c_2 qui échoue. Dans ces deux cas on cherche la configuration minimale soit dans c_1 soit dans c_2 ; l.15) Cas d'interférence : c'est l'union de changements dans c_1 et de changements dans c_2 qui provoque l'échec. On cherche un ensemble minimal dans c_1 en fixant c_2 (qui est inclue dans la solution) et dans c_2 en fixant c_1.

Couplée à une analyse abstraite, la shadow-memory permet de retrouver l'expression arithmétique responsable d'une erreur globale comme cela est notamment développé dans l'outil Herbgrind [156]. Puisqu'elle intervient au niveau de l'exécution, la shadow-memory est proche des instructions réellement exécutées, en particulier des possibles erreurs dues aux optimisations du compilateur, ce qui permet de suivre pas à pas l'évolution du calcul et détecter plus rapidement les erreurs numériques.

3.5.3 Différentiation automatique

La différentiation automatique évalue la dérivé numérique d'une opération arithmétique ou mathématique et chaîne ces dérivés suivant l'ordre des calculs dictés par le programme. Contrairement à la différentiation symbolique, la différentiation automatique permet de prendre en compte les erreurs d'arrondi ainsi que les erreurs d'annulations puisqu'elle suit l'ordre des opérations.

La différentiation automatique est notamment utilisée pour calculer de manière précise la sensibilité aux incertitudes sur les variables en entrée. En effet, le gradient d'une fonction f à plusieurs variables au voisinage d'un point $x = (x_1, x_2, \ldots, x_n)$ donne la direction d'accroissement de cette fonction suivant ses composantes, c'est-à-dire ses dérivés partielles $\frac{\partial f}{\partial x_i}$.

Définition 3.5.1 (Gradient d'une fonction)**.** Soit $f : \mathbb{R}^n \to \mathbb{R}$ une fonction différentiable au voisinage de $x \in \mathbb{R}^n$ alors le gradient de f noté ∇ est l'application linéaire :

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right).$$

Plus la norme d'une de ces dérivés est forte et plus elle aura d'incidence dans le gradient. Autrement dit, une petite perturbation sur cette composante aura un impact fort sur le résultat. On peut ainsi calculer les dépendances du résultat au bruit en entrée sur l'une des composantes [180]. Le même principe peut être appliqué pour déterminer si l'erreur dûe à la réduction de précision impacte fortement le résultat du calcul [127].

La différentiation automatique est comme son nom l'indique automatique ce qui permet de décharger une grande partie d'analyse du comportement numérique à des outils spécialisés l'implémentant de manière efficace [87, 78, 66].

3.6 Conclusion

Nous avons présenté dans ce chapitre Verificarlo, un outil pour analyser le comportement des calculs flottants dans des codes. Verificarlo a été au départ pensé comme un compilateur mais s'est depuis élargi et est aujourd'hui arrivé au statut de boîte à outil puisqu'il permet, non plus seulement d'analyser la stabilité numérique des codes à travers ses implémentations de l'arithmétique de Monte Carlo, mais également de simuler des formats de taille variable par l'intermédiaire du nouveau backend `VPREC` qui sera présenté en section 5.3 du chapitre 5.

L'interface modulaire de Verificarlo permet de facilement ajouter des nouveaux backends pour tester de nouveaux formats ou de nouveaux modèles de calcul. Cela nous a permis d'aisément ajouter le backend `BITMASK`. Ce backend offre un compromis intéressant entre performance et qualité du bruit introduit dont nous verrons l'utilité dans le chapitre 4 suivant. Pour utiliser Verificarlo dans un contexte HPC, une attention particulière a été apportée dans cette thèse pour améliorer la

performance des backends. Ainsi nous avons améliorer les performances du backend `MPFR` d'un facteur $\times 2,5$ ce qui a permis de faciliter son usage.

La dernière partie de ce chapitre a montré que l'application de Verificarlo sur des codes industriels nécessitait des méthodes pour réduire les espaces de recherche. Ces méthodes sont utiles pour utiliser de manière efficace Verificarlo et obtenir des informations pertinentes sur le code analysé. Ainsi dans le chapitre 4, nous montrerons comment on peut utiliser Verificarlo pour explorer la dimension temporelle des calculs. Dans le chapitre 5, nous montrerons que Verificarlo peut être employé dans un autre registre du calcul flottant. Nous verrons en effet que le backend `VPREC` couplé avec une exploration temporelle permet de réduire la précision de calcul nécessaire.

Chapitre 4

Veritracer : Un outil de débogage numérique

Le débogage numérique dans le contexte HPC est un problème difficile pour deux principales raisons : premièrement, il est difficile d'estimer la justesse d'un résultat sans en connaître la solution et uniquement mesurer la fidélité peut amener à de mauvaises interprétations comme le démontre le simple exemple de la suite de Muller (cf.equation 4.1). Secondement, une fois la mauvaise qualité connue, il est difficile d'identifier les calculs qui entachent la solution d'erreurs parmi des centaines de milliers de lignes code.

Dans ce chapitre nous nous sommes intéressé au débugage numérique de codes HPC en exploitant la dimension temporelle des calculs, dimension jusque là sous-exploitée par les outils existants, qui permet de comprendre l'évolution de la qualité numérique des calculs au cours du temps. Ce chapitre est divisé en deux parties.

Dans la première partie nous présentons Veritracer, un traceur numérique pour visualiser la qualité numérique des calculs flottants d'un code au cours du temps. Nous présentons comment Veritracer enrichie les traces qu'il génère par des informations contextuelles comme le contexte d'appels qui donne le chemin d'appels d'un calcul. Puis nous détaillons les optimisations apportées pour que Veritracer puisse gérer des traces de données contenant plusieurs millions de mesures. Cela permet d'utiliser Veritracer sur le code HPC de chimie quantique ABINIT et nous mène à la seconde partie.

La seconde partie du chapitre étudie le débugage numérique du code ABINIT. Tracer tous les calculs d'ABINIT à l'aide de Veritracer est impossible puisque le nombre d'invocations à tracer est trop grand. De plus, dans une démarche de débugage, seuls les calculs entachant le résultat du code nous intéressent. Pour résoudre ce problème nous présentons deux méthodologies dont l'objectif est de révéler les fonctions ayant la plus grosse contribution à l'erreur globale (i.e. erreur sur le résultat final). La première méthodologie identifie les fonctions qui ont une contribution à l'erreur globale nous permettant de réduire le nombre de fonctions à analyser de 2952 à 88. La seconde classe ces fonctions par degré de sensibilité donnant au développeur un moyen de hiérarchiser ses priorités d'action. Nous montrons que l'application de ces méthodologies a automatiquement révélé la fonction `simp_gen` comme fonction parmi les plus sensibles. De manière indépendante, les développeurs d'ABINIT avaient suspecté cette fonction de provoquer des instabilités numériques dans ABINIT, renforçant notre analyse.

```
11 double muller(double u_n, double u_nm1) {
12  return 111.0 - 1130.0/u_n + 3000.0/(u_n*u_nm1);
13 }
14
15 int main(int argc, char *argv[]) {
16  int i = 0;
17  double u_nm1 = 2.0, u_n = -4.0, t = 0;
18  for (i = 0; i < 30; i++) {
19   t = u_n;
20   u_n = muller(u_n, u_nm1);
21   u_nm1 = t;
22  }
23  return 0;
24 }
25
```

FIGURE 4.1 – Code réalisant l'équation 4.1 en binary64.

Dans la dernière partie de ce chapitre, nous démontrons l'efficacité de Veritracer pour étudier finement la fonction `simp_gen`. Nous commençons par montrer la mauvaise qualité du résultat de `simp_gen` au cours du temps. Puis nous montrons qu'il est possible d'améliorer la qualité de `simp_gen` grâce à la méthode compensée `Dot2`. Nous montrons que Veritracer montre l'amélioration de la qualité dans 30 contextes d'appels parmi les 31 de `simp_gen`. Enfin nous montrons que l'analyse du flot de données permet d'expliquer la mauvaise qualité du contexte d'appels restant.

4.1 Comment enrichir les informations fournis par Verificarlo ?

Verificarlo permet d'instrumenter automatiquement les opérations flottantes pour les remplacer par leur équivalent dans MCA. Verificarlo est utile pour voir les effets d'un bruit aléatoire sur la qualité des résultats. Il permet notamment d'en estimer la fidélité grâce aux formules pour calculer le nombre de chiffres significatifs (cf. 2.6.3). Son fonctionnement est détaillé dans le chapitre 3.

Néanmoins, uniquement regarder la fidélité des résultats peut conduire à des interprétations erronées comme le montre l'exemple de la suite de Muller [5]. Dans cette suite (u_n) définit par la récurrence suivante :

$$u_{n+1} = \begin{cases} 2 & \text{si } n = 0 \\ -4 & \text{si } n = 1 \\ 111 - \dfrac{1130}{u_n} + \dfrac{3000}{u_n u_{n-1}} & \text{sinon} \end{cases}$$

n'importe quelle évaluation en précision finie converge vers la valeur 100. Or la limite exacte est 6. La raison est que 6 et 100 sont toutes les deux racines de (u_n) :

$$u^3 - 111u^2 + 1130u - 3000 = (u-5)(u-6)(u-100)$$

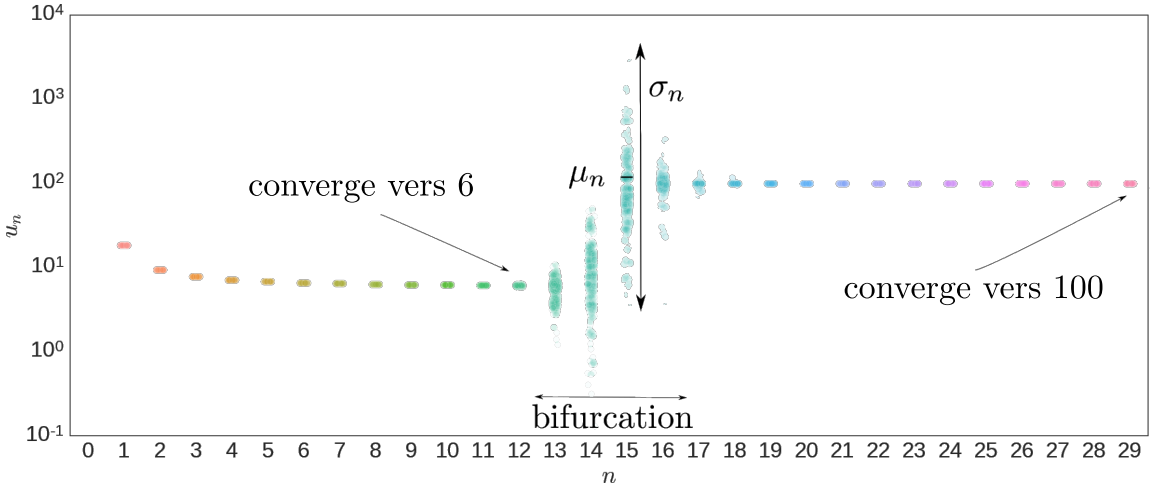


FIGURE 4.2 – Superposition de 500 exécutions MCA du code figure 4.1 (`-precision 53`, `-mode FULL`). Le code converge vers la solution exacte 6 avant de bifurquer vers 100. La même variable `u_n` située ligne 10 possède une justesse très inégale au cours du temps.

Les accumulations d'erreurs font diverger le calcul vers la valeur 100 inexorablement. Nous avons souhaité connaître la justesse de ce résultat avec la méthode MCA. L'évaluation de la suite (sur le code associé figure 4.1) avec Verificarlo donne un résultat égal en moyenne à 100 avec un écart-type de l'ordre de la précision machine pour 500 exécutions. Soit 53 bits significatifs d'après la définition 2.6.1, autrement dit tout les bits de la mantisse sont justes.

Or lorsque l'on regarde de plus près l'évolution temporelle des calculs comme le montre la figure 4.2 qui superpose les valeurs u_k pour $1 \leqslant k \leqslant 30$, on voit que l'ensemble des exécutions convergent vers 6 avant de bifurquer à partir l'itération 12. L'introduction du bruit aléatoire met en exergue l'instabilité numérique du calcul entre les itérations 12 et 17. De plus, l'échelle logarithmique sur l'axe des ordonnées reflète bien l'écart considérable entre les valeurs des différentes exécutions. Les calculs des itérations situées dans la zone de bifurcations ne contiennent plus d'information puisque tous les chiffres significatifs ont été perdus. Après cette zone de turbulence, les exécutions convergent toutes vers la solution 100.

On en conclut donc que le que le résultat est fidèle. Fidèle mais inexact. L'analyse temporelle se révèle ici indispensable pour identifier le problème d'instabilité numérique et montre la nécessité d'étudier la qualité des résultats intermédiaires au cours du temps.

Veritracer a été conçu pour analyser l'évolution des calculs flottants au cours du temps. Il repose sur l'analyse de traces numériques générées automatiquement par le code qui a été instrumenté avec des sondes durant la compilation. Ces sondes impriment le résultat du calcul ainsi que des informations liées comme le nom de la variable associée au calcul ou le nom de la fonction appelante. Ces traces sont ensuite agrégées et peuvent être visualisées dynamiquement à l'aide de la bibliothèque Python `matplotlib`.

4.2 Schéma conceptuel de Veritracer

Veritracer est une chaîne d'outil Python qui permet d'instrumenter le code et analyser puis visualiser les traces générées. Le schéma conceptuel de Veritracer est présenté sur la figure 4.3 avec pour chaque étape du pipeline une section de ce chapitre.

Section 4.3 : décrit l'instrumentation des opérations flottantes durant la compilation pour insérer les sondes de traçage ainsi que la récupérations des informations de débugage.

Section 4.5 : décrit l'exécution automatique des binaires en parallèles à travers une commande python minimale.

Section 4.6 : décrit l'analyse automatique des traces générées et leur agrégation au sein d'un fichier.

Section 4.7 : décrit la visualisation des valeurs agrégées dans le fichier généré par la commande `veritracer analyze`.

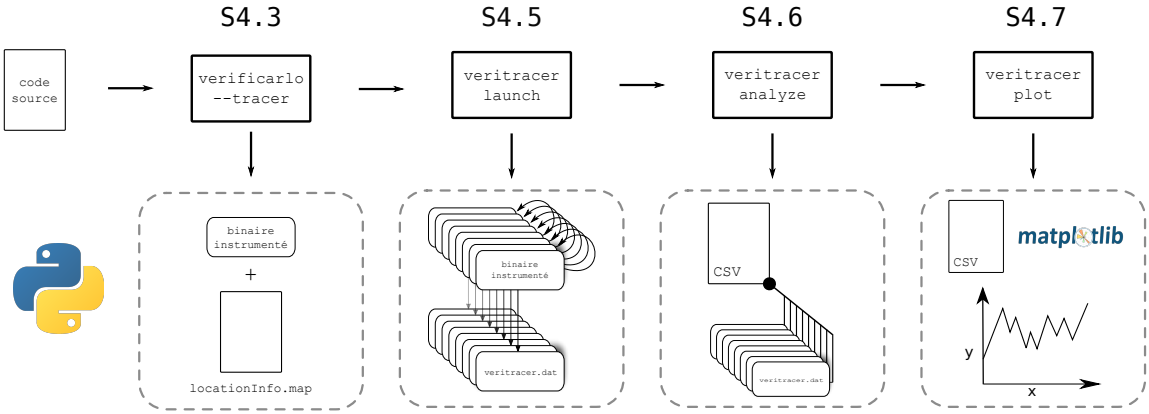


FIGURE 4.3 – Pipeline d'exécution de Veritracer. L'option `-tracer` active l'instrumentation des opérations flottantes pour le traçage. La commande `veritracer launch` lance les exécutions en parallèles. Ces exécutions génèrent des fichier `veritracer.dat` qui sont analysés par la commande `veritracer analyze`. Cette dernière agrègent les valeurs dans un fichier Comma Separated Values (CSV). Enfin, la commande `veritracer plot` permet de visualiser les valeurs du fichier CSV.

4.3 Instrumentation

L'instrumentation insère du code dans un programme cible afin d'en analyser le comportement. Nous allons tout d'abord présenter les deux classes d'instrumentations. Puis nous montrerons l'approche retenue dans Veritracer, les difficultés rencontrées et comment nous les avons surmontées.

Il existe deux grands types d'instrumentation : l'instrumentation binaire qui agit au niveau de l'exécutable et l'instrumentation source qui opère au niveau du code source. Les deux approches présentent des avantages et des inconvénient que nous allons analyser.

4.3.1 Instrumentation binaire

L'instrumentation binaire nécessite un outil qui puisse décoder dynamiquement chaque instruction assembleur pour les modifier ensuite. Il est en effet difficile (mais néanmoins possible [59]) de modifier un code assembleur de manière statique puisque cela nécessite de désassembler le code et d'ajouter des instructions ce qui peut modifier le calcul des adresses par exemple. Les principaux outils d'instrumentation binaire sont Intel Pin [119], DynamoRIO [18] et Valgrind [137]. Ce dernier est utilisé dans beaucoup d'outil d'analyse de code de part sa modularité. On peut également citer gdb [70], le débogueur du projet GNU's Not UNIX (GNU).

Les principaux avantages de l'instrumentation binaire sont sa facilité d'utilisation, sa portabilité et sa fidélité. Facilité d'utilisation puisque l'instrumentation est faite directement au niveau du binaire ce qui ne nécessite aucune recompilation. Portabilité car ce type d'instrumentation est agnostique au langage et au compilateur. Cela permet d'instrumenter des binaires dont le code source n'est pas disponible comme les logiciels propriétaires ou les bibliothèques externes [1]. Enfin fidélité au niveau de l'analyse puisque le code analysé est le code qui est réellement exécuté sur la machine.

Les principaux désavantages de l'instrumentation binaire sont son surcoût et la perte des informations liées au code source. Le surcoût est lié au décodage des instructions puisque chaque instruction doit être décodée puis exécutée ce qui pollue les mémoires caches. On peut néanmoins payer ce surcoût une seule fois par instruction en stockant l'instruction décodée dans une table de hachage.

D'autre part, ne pas avoir à disposition le code source limite le niveau des informations de débugage disponible. Il est en effet possible de décoder les informations DWARF inclues dans le binaire mais pas les informations de compilation de haut niveau. Ces informations sont utiles pour connaître les relations entre le code source et le code assembleur exécuté comme par exemple savoir à quelle variable correspond une instruction assembleur.

4.3.2 Instrumentation du code source

L'instrumentation du code source consiste à introduire des sondes dans le code puis de compiler cette version modifiée du code originale. Ces sondes peuvent être introduites manuellement ou automatiquement. La partie automatique se décline également en deux versions : la transformation source à source et la transformation au niveau du langage intermédiaire. La première rend dépendant vis à vis des langages supportés, la deuxième vis à vis du compilateur utilisé. Néanmoins, la deuxième solution permet de supporter tous les langages supportés par le compilateur et permet de rester proche du code originale exécuté en introduisant du code après application des optimisations. Le contenu des sondes sera ensuite exécuté durant l'exécution de l'application.

L'avantage de l'instrumentation au niveau source est son surcoût minime puisqu'il est directement lié au temps d'exécution des sondes. A noter que même pour des sondes avec très peu d'instructions, les appels répétés à la sonde dans des sections critiques comme des boucles peuvent entraîner un facteur de ralentissement non négligeable en détruisant la vectorisation ou en polluant les mémoires caches. Le surcoût reste néanmoins plus faible comparé à une instrumentation binaire. Autre avantage, l'accès aux informations de débugage durant la compilation permet de relier précisément les instructions assembleurs aux variables. Ces informations facilitent la compréhension des analyses effectuées sur le code.

1. A condition que ceux ci ne soient pas obfusqués.

Le principale désavantage de l'instrumentation au niveau source est la dépendance vis à vis du compilateur et du langage utilisé. Ce qui limite le spectre des codes pouvant être analysés et oblige à recompiler les bibliothèques externes. D'autre part l'introduction de code externe modifie la performance de l'application originale ce qui peut être un problème lorsqu'on souhaite mesurer précisément les performances.

4.3.3 Instrumentation de l'IR

Dans Veritracer l'instrumentation est faite au niveau de l'IR LLVM, un assembleur de haut niveau présenté dans la section 4.3.4, ce qui offre un bon compromis entre l'instrumentation binaire et l'instrumentation au niveau du code source.

Comme l'instrumentation binaire, l'instrumentation au niveau de l'IR permet d'être proche du code assembleur qui sera exécuté sur machine. L'instrumentation est faite avant la génération du code exécuté ce qui permet de préserver les optimisations appliquées par le compilateur et d'en mesurer les effets sur la qualité des calculs. De plus travailler au niveau de l'IR permet de supporter tous les langages acceptés par LLVM.

Comme pour l'instrumentation au niveau code source, on dispose d'un faible surcoût d'instrumentation du à un appel de fonction. Le principal facteur de ralentissement vient alors du contenu des fonctions.

4.3.4 LLVM

LLVM [113] est une suite d'outil de compilation sur laquelle repose Veritracer. LLVM a été introduit dans la section 3.1.1 du chapitre 3 consacrée à Verificarlo. LLVM a été construit pour faciliter les analyses interprocédurales et les analyses basées sur des profilages de code. Le coeur de LLVM est son langage intermédiaire (IR) proche de l'assembleur dans sa syntaxe mais fournissant des informations de haut-niveau qui facilitent les analyses. De plus l'IR est agnostique au langage source utilisé.

Représentation intermédiaire

L'IR utilise la forme Static Single Assignment (SSA) [40] qui assigne chaque variable à un registre virtuel une et une seule fois. Ainsi chaque instruction qui produit une valeur comme l'instruction `add` par exemple génère un nouveau registre virtuel. Cette forme facilite l'analyse de dépendance entre les variables pour l'optimisation du code comme le montre l'exemple sur la figure 4.4.

L'IR est un langage assembleur virtuel qui s'abstrait du matériel à travers des registres virtuels. Dans ce format, il n'existe pas de bijection entre les registres virtuels et physiques.

Dans LLVM la mémoire est divisée en trois parties : la pile, le tas et la zone globale. La mémoire est allouée sur la pile avec l'instruction `alloca` et est liée au contexte d'appel de la fonction courante. Cette mémoire est ensuite désallouée automatiquement à la sortie de la fonction. La mémoire sur le tas est gérée manuellement par les fonctions `malloc` et `free` qui respectivement alloue et libère l'espace mémoire. La zone globale contient quant à elle toutes les variables globales et les objets alloués statiquement.

LLVM possède une vision unifiée de la mémoire et tous les objets adressables sont accédés à travers leur adresse. Ainsi les variables sont accédés en lecture et en écriture à travers les instructions `load` qui charge une valeur depuis une adresse mémoire dans un registre et `store` qui écrit la

	Original		Forme SSA		Optimisé
1	$y = 1$	1	$y_1 = 1$	1	
2	$x = 2$	2	$x_1 = 2$	2	
3	$x = y$	3	$x_2 = y_1$	3	$x_2 = 1$
4		4		4	

FIGURE 4.4 – Illustration des avantages de la forme SSA. Après avoir été transformé en forme SSA, le code est facilement optimisable par le compilateur qui peut éliminer la ligne 2 inutile et propager la constante de y_1 à x_2

valeur d'un registre en mémoire. Ce mécanisme de transition facilite l'analyse des lectures/écritures puisqu'il suffit d'analyser les `load` et les `store` dans l'IR.

Néanmoins, ce transfert entre mémoire et registre est coûteux en pratique et l'emploi de certaines passes d'optimisations permet de garder l'utilisation de valeur en mémoire dans les registres.

La gestion des types complexes

Veritracer gère les types natifs et les types composés comme les pointeurs, les tableaux, les structures et la composition de ces types. Puisque chaque objet adressable est accédé à travers son adresse, LLVM utilise l'instruction `getelementptr` pour calculer l'adresse mémoire de chaque élément. Comme chaque élément d'un type composé peut également être un type composé, de multiples index sont nécessaires. Pour les pointeurs, la valeur retournée par `getelementptr` est la variable vers laquelle il pointe, pour les structures elle retourne un pointeur sur le champ choisi et pour les tableaux un pointeur sur l'élément de l'indice choisi. L'exemple de la figure 4.5 illustre ces propos. Pour récupérer la variable d'origine, nous remontons récursivement la chaîne de pointeurs.

4.3.5 Insertion des sondes de traçage

Les sondes qui sont insérées dans le code ont le prototype suivant :

```
void _veritracer_probe_<type>_<format>(<type> value,
  <type*> ptr,
  uint64_t hashLocationInfo);
```

où `<type>` est le type de la valeur (`value`) flottante tracée (`float` ou `double`), `ptr` l'adresse de `value` et `hashLocationInfo` est la valeur de hachage associée aux informations de débugage. Le `<format>` correspond au format de sortie, binaire ou texte. Les `<type>` supportés sont `float`, `double` et leur version vectorielle (float2, float4, ...).

Instrumentation automatique

L'instrumentation automatique est réalisée à l'aide d'une passe de compilation LLVM. Celle ci se présente sous la forme d'une classe `C++` héritée de la classe `ModulePass`. Les trois fonctions importantes de cette classe sont les fonctions `runOnModule`, `runOnFunction` et `runOnBasicBlock` qui comme leur nom indique sont exécutées sur chaque module, fonction et basic block du programme.

```
1  struct RT {
2   char A;
3   int B[10][20];
4   char C;
5  };
6  struct ST {
7   int X;
8   double Y;
9   struct RT Z;
10 };
11
12 int *foo(struct ST *s) {
13  return &s[1].Z.B[5][13];
14 }
```

```
%struct.ST = type { i32, double, %struct.RT }
%struct.RT = type { i8, [10 x [20 x i32]], i8 }

define i32 @foo(%struct.ST*) #0 {
 %2 = alloca %struct.ST*, align 8
 store %struct.ST* %0, %struct.ST** %2, align 8
 %3 = load %struct.ST*, %struct.ST** %2, align
     8
 %4 = getelementptr inbounds %struct.ST, %
     struct.ST* %3, i64 1
 %5 = getelementptr inbounds %struct.ST, %
     struct.ST* %4, i32 0, i32 2
 %6 = getelementptr inbounds %struct.RT, %
     struct.RT* %5, i32 0, i32 1
 %7 = getelementptr inbounds [10 x [20 x i32]],
      [10 x [20 x i32]]* %6, i64 0, i64 5
 %8 = getelementptr inbounds [20 x i32], [20 x
     i32]* %7, i64 0, i64 13
 %9 = load i32, i32* %8, align 4
 ret i32 %9
}
```

FIGURE 4.5 – Illustration de l'instruction `getelemenptr` sur la requête ligne 14. Le premier et deuxième argument sont le type et pointeur vers la variable. Ensuite, les arguments sont les indices d'accès.

Notre passe d'instrumentation exécute le contenu de `runOnBasicBlock` illustrée sur la figure 4.6 sur chaque basic block des fonctions à instrumenter. Pour chaque instruction pouvant être instrumenté, notre passe insère un appel à la sonde avec les arguments adéquats à l'aide de la fonction `insertProbe` (l.7).

Instrumentation manuelle

Pour des instrumentations ciblées d'un code nous avons également proposé l'insertion de sonde manuelle. En effet, par défaut Veritracer trace l'ensemble des instructions d'un code. Or on souhaite parfois pouvoir uniquement tracer une variable en particulier. Nous disposons pour cela de deux sondes :

```
void vfc_probe_binary32(float*, const char*);
void vfc_probe_binary64(double*, const char*);
```

avec comme premier argument l'adresse de la variable à tracer et en deuxième argument le nom associé. Ces appels sont ensuite traduits vers les sondes internes présentées au début de la section 4.3.5.

4.3.6 Informations de débugage

Pour chaque fonction instrumentée, Veritracer ajoute une sonde par variable flottante pour imprimé sa valeur numérique. Néanmoins il est nécessaire d'avoir des informations pour associer les mesures aux variables correspondantes. Pour cela Veritracer utilise les informations de débogage ajoutées par le compilateur. Les informations de débugage sont des informations relatives au langage source qui permette à l'humain de comprendre les relations entre le code assembleur et le code source. En effet, les instructions assembleur n'ont besoin de manipuler que des valeurs (scalaire ou

```
1  bool runOnBasicBlock(Module &M, BasicBlock &B, vfctracerFormat::
       Format &Fmt) {
2    bool modified = false;
3    for (Instruction &ii : B) {
4     vfctracerData::Data *D = vfctracerData::CreateData(&ii);
5     if (D == nullptr || not D->isValidOperation() || not D->
      isValidDataType())
6       continue;
7     modified |= insertProbe(Fmt, *D);
8     if (VfclibBacktrace && modified)
9       insertBacktraceCall(&M, ii.getFunction(), &ii, &Fmt, D);
10   }
11   return modified;
12  };
13
```

FIGURE 4.6 – Code de la fonction `runOnBasicBlock` exécuté sur chacun des basic blocks des fonctions à instrumenter. La fonction itère sur chacune des instructions (l.3) du basic block et créée un objet `Data` pour chacune d'entre elle. Si l'objet crée est valide alors une sonde est inséré à l'endroit de cette instruction

adresse). Le nom ou la localisation d'une variable est inutile au processeur pour faire des calculs. Les informations de débugage sont donc ajoutées par le compilateur à la demande du développeur par l'intermédiaire de l'option `-g` notamment.

Les informations dans LLVM

LLVM maintient des informations de débugage lors de compilation sur les descriptions des types, des fonctions, des variables ou des fichiers sources. Ces informations sont maintenues par LLVM et accessibles à travers la manipulation de l'IR. LLVM [2] traque ces informations de débugage à travers l'utilisation d'intrinsèques (3 au total) dont la fonction `void @llvm.dbg.declare(metadata, metadata, metadata)`. Une métadonnée est une donnée sur la structure du code en lui-même qui n'est pas nécessaire pour la bonne exécution du programme. Cette fonction prend trois arguments : le 1er est une métadonnée qui contient l'adresse de la variable en question. Le 2ème est l'information en elle même. Enfin le 3ème est une métadonnée contenant des informations supplémentaires.

La figure 4.7 illustre l'utilisation de ces fonctions dans l'IR. Les lignes 5 et 9 sont les `declare` associés aux instructions `%3` et `%4` qui sont les deux arguments de de la fonction `muller`. Les deuxièmes arguments, respectivement `!23` et `!25`, sont les informations de débugage à proprement parler. Ces variables sont de type `DILocation` qui renseigne sur le nom, la place de l'argument, le scope, le fichier, la ligne et le type de la variable. Chacun des champs est lui même une valeur comme pour la ligne par exemple ou une `metadata` qu'il convient de parcourir récursivement.

2. Actuellement la dernière version est la `9.0.0`.

```
define double @muller(double, double) #0 !dbg !22 {
 %3 = alloca double, align 8
 %4 = alloca double, align 8
 store double %0, double* %3, align 8
 call void @llvm.dbg.declare(metadata double* %3,
               metadata !23,
               metadata !11), !dbg !24
 store double %1, double* %4, align 8
 call void @llvm.dbg.declare(metadata double* %4,
               metadata !25,
               metadata !11), !dbg !26
 %5 = load double, double* %3, align 8, !dbg !27
 %6 = fdiv double 1.130000e+03, %5, !dbg !28
 %7 = fsub double 1.110000e+02, %6, !dbg !29
 %8 = load double, double* %3, align 8, !dbg !30
 %9 = load double, double* %4, align 8, !dbg !31
 %10 = fmul double %8, %9, !dbg !32
 %11 = fdiv double 3.000000e+03, %10, !dbg !33
 %12 = fadd double %7, %11, !dbg !34
 ret double %12, !dbg !35
}

!1 = !DIFile(filename:"muller.c", directory:"/home/yohan/These/data/muller")
!22 = !DISubprogram(name:"muller", scope: !1, file: !1, line: 11, type: !7,
          isLocal: false, isDefinition: true, scopeLine: 11,
          flags: DIFlagPrototyped, isOptimized: false, unit: !0,
          variables: !2)
!23 = !DILocalVariable(name:"u_n", arg: 1, scope: !22, file: !1, line: 11,
            type: !9)
!24 = !DILocation(line: 11, column: 22, scope: !22)
!25 = !DILocalVariable(name: "u_nm1", arg: 2, scope: !22, file: !1, line: 11,
            type: !9)
!26 = !DILocation(line: 11, column: 34, scope: !22)


```

FIGURE 4.7 – IR généré par clang 4.0.1 sans optimisations (`-O0`) avec l'option de debug `-g` à partir du code figure 4.1. En bas sont présentées les informations de débugage relatives à la fonction `muller` ainsi qu'à ses arguments.

```
valeur hachage    :type    ; fonction; location     ; nom
7571129480221044601 :binary64 ; muller ; muller.c 11.0 ; u_n
7699751665090218657 :binary64 ; muller ; muller.c 11.0 ; u_nml
17754378457844497782:binary64 ; main   ; muller.c 20.7 ; t
14962935724446938191:binary64 ; main   ; muller.c 21.9 ; u_n
1291600118943258965 :binary64 ; main   ; muller.c 22.11 ; u_nml
```

FIGURE 4.8 – Exemple du ficher `locationInfo.map` qui associe à chaque valeur de hachage les informations de débugage correspondantes.

Les informations sélectionnées

Nous avons choisi de retenir les informations suivantes :

- le nom de la variable
- le type de la variable
- la fonction appelante
- la ligne originale dans le code source

Pour éviter de copier ces informations dans le fichier de sortie à chaque appel de la sonde nous utilisons une fonction de hachage, et nous stockons les métadonnées dans un dictionnaire global. Ces informations qui sont des chaînes de caractères sont concaténées et hachées durant la compilation. L'ensemble des correspondances valeurs hachées/informations de débugage est ensuite sauvegardé dans un fichier à la fin de la compilation. Un exemple de ce fichier est donné sur la figure 4.8.

4.3.7 Problèmes liés aux passes d'optimisations

Nous avons vu dans la section 3.1.3 les problèmes que pouvait engendrer les optimisations appliquées par le compilateur sur la qualité numérique du code généré. Par exemple les drapeaux de compilations `-O3 -ffast-math` lorsqu'ils étaient appliqués à la somme de Kahan transformaient celle ci en somme naïve, détruisant toute la compensation mis en place. Ces optimisations touchent également aux informations de débogage. En effet, ces optimisations qui sont découpées en plusieurs passes transforment chacune d'entre elles l'IR pour améliorer ses performances ou enlever les instructions mortes ou inutiles. Cela se traduit par une réduction du trafic de données entre la mémoire et les registres virtuels. La passe de compilation `-mem2reg` promeut par exemple les valeur en mémoire dans les registres ce qui supprime les `load` et `store` ainsi que les variables locales devenus inutiles.

Pour résoudre ce problème, on peut spécifier à Veritracer de tracer les calculs intermédiaires, c'est-à-dire les instructions parmi {`fadd`,`fsub`,`fmul`,`fdiv`,`ret`}. Les quatre premières correspondent aux opérations arithmétiques de base. La dernière `ret` est une instruction utilisé pour retourner le flux de contrôle à la fonction appelante avec éventuellement une valeur de retour, équivalent au mot clé `return` en langage `C`. On trace également la valeur retourné par `ret` lorsqu'elle est de type flottant. On cherche ensuite les informations de débugage pour les opérandes des instructions. Si une des opérandes n'a pas d'information, on utilise la valeur du registre LLVM. La figure 4.10 montre les informations trouvée par Veritracer pour la fonction `muller` à partir du code présenté figure 4.9.

```
define double @muller(double, double) local_unnamed_addr #0 !dbg !20 {
 tail call void @llvm.dbg.value(metadata double %0, i64 0, metadata !22, metadata !13), !dbg !24
 tail call void @llvm.dbg.value(metadata double %1, i64 0, metadata !23, metadata !13), !dbg !25
 %3 = fdiv double 1.130000e+03, %0, !dbg !26
 %4 = fsub double 1.110000e+02, %3, !dbg !27
 %5 = fmul double %0, %1, !dbg !28
 %6 = fdiv double 3.000000e+03, %5, !dbg !29
 %7 = fadd double %4, %6, !dbg !30
 ret double %7, !dbg !31
}
```

FIGURE 4.9 – IR généré par clang 4.0.1 avec optimisations (`-O3`) et information de débugage activé (`-g`). Il n'est plus suffisant de traquer les `store` et `load` car ceux ci ont été supprimés.

```
location       ; nom
muller.c 12.24 ; %3 = double 1.130000e+03 / u_n
muller.c 12.16 ; %4 = double 1.110000e+02 - %3
muller.c 12.42 ; %5 = u_n * u_nm1
muller.c 12.37 ; %6 = double 3.000000e+03 / %5
muller.c 12.29 ; %7 = %4 + %6
muller.c 12.3  ; ret %7
```

FIGURE 4.10 – Informations de débugage trouvées par Veritracer lorsqu'on demande a celui ci de tracer les calculs intermédiaires. Le nom des valeurs est construit sur `<nom du registre llvm>` = `<nom op1> <nom op> <nom op2>` où `op1`,`op2` représentent les opérandes de l'opération `op`. Si une opérande ne correspond pas à une variable on utilise le nom du registre LLVM correspondant.

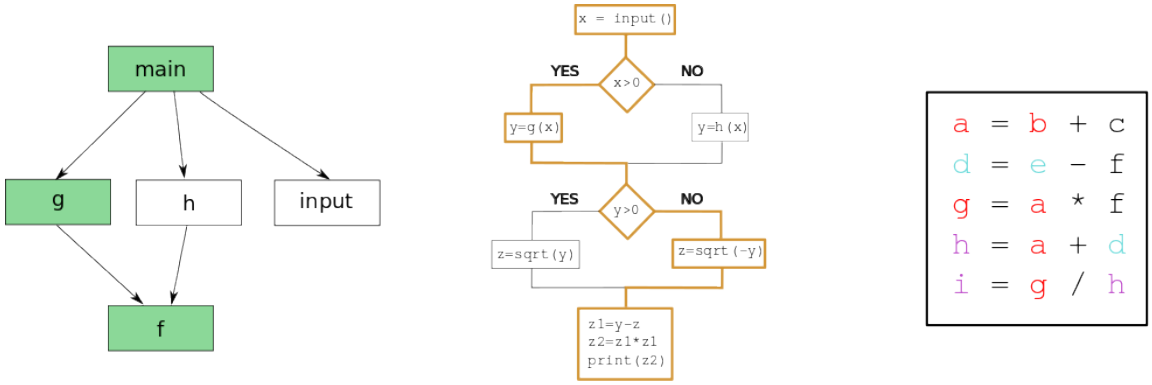


FIGURE 4.11 – Illustration de trois types d'analyse du flot d'informations. A gauche, l'analyse du contexte d'appels qui donne la chaîne des fonctions appelantes pour une fonction donnée. Ici, le contexte d'appels de la fonction `f` est `f <- g <- main`. Au centre le flot de contrôle qui regarde les chemins empruntées en fonction des branchements. À droite, l'analyse du flot de données qui traque les dépendances entre les calculs. On voit sur l'exemple que la variable `f` est influencée par les variables `b` et `e`.

4.4 L'analyse du flot d'information

Verificarlo est basé sur une méthode MCA asynchrone. Cela impose de s'assurer que les valeurs comparées proviennent du même chemin d'exécution. En effet, le bruit injecté par MCA peut conduire à des chemins d'exécutions différents. De plus, d'un point de vue temporel, deux mesures d'une même variable peuvent ne pas être corrélées si elles ne partagent pas la même histoire. Histoire qui est liée au chemin d'exécution.

Pour pallier ce problème, nous avons besoin de pouvoir comparer le flot d'information des variables tracées. Il existe différentes granularités pour analyser le flot d'information comme le montre la figure 4.11 : au niveau du contexte d'appels, du flot de contrôle ou du flot de données que nous allons chacun présenter dans cette section.

4.4.1 Analyse contextuelle

L'analyse contextuelle étudie le contexte d'appels d'une fonction durant l'exécution. Le contexte d'appels d'une fonction est la pile d'appels de fonctions qui ont précédés l'appel à la fonction courante. Ce contexte est dynamique et spécifique à chaque appel de fonction. Ainsi deux appels à la même fonction peuvent avoir deux contextes d'appels différents. Par exemple le contexte d'appels de la fonction `h` dans la figure 4.12.

Pour connaître le contexte d'appels des variables tracées, nous avons utilisé la fonction `backtrace` de la GNU C Library. Cette fonction retourne le contexte d'appels (*backtrace* en anglais) de l'endroit où est appelé cette fonction. Après avoir avoir inséré les sondes de traçage, nous introduisons un deuxième appel à une sonde qui écrit le contenu de backtrace dans un fichier. Ce contenu est similaire au profil présenté dans la figure 4.12.

Cette technique présente l'avantage d'être facile d'implémentation. Le problème est que plusieurs requêtes à la même trace d'appels peuvent être demandés au sein d'une même fonction comme pour

```c
float f(float x) {
  return g(x)+1;
}
float g(float x) {
  return h(x)*2;
}
float h(float x) {
  return x*x;
}

int main() {
  f(2);
  g(2);
  return 0;
}
```

(a) Code illustrant les contextes d'appels

```
./backtrace(h+0x12)
./backtrace(g+0x4d)
./backtrace(f+0x4d)
./backtrace(main+0x23)

./backtrace(h+0x12)
./backtrace(g+0x4d)
./backtrace(main+0x38)
```

(b) Contextes d'appels de la fonction `h`

FIGURE 4.12 – Contextes d'appels associés de la fonction `h` construit comme `<nom du binaire>(<fonction>+<adresse de l'instruction>)`. L'adresse de l'instruction correspond à un décalage à partir de l'adresse de la fonction.

le cas d'une boucle par exemple. Le souci est que la trace d'appels peut être potentiellement grande pour une fonction de calcul spécifique comme un produit scalaire ou une addition de vecteur. En effet ces fonctions élémentaires sont souvent appelées après une longue chaîne d'autres fonctions dans un code de simulation. La trace d'appels est donc potentiellement grande. Cela est très impactant sur le surcoût engendré si cette trace d'appels est demandée plusieurs fois consécutivement comme dans une boucle. Or les boucles sont légion dans les codes de simulations.

Pour éviter ce goulet d'étranglement, nous prenons la trace d'appels une seule fois à chaque entrée de fonction. Cela permet d'économiser autant d'appels qu'il y a de sondes de traçage. Á la sortie de la fonction ou à l'entrée d'une nouvelle, la *backtrace* est écrit dans un fichier (`backtrace.dat`) ainsi que la valeur de hachage de toutes les variables tracées entre temps. L'ordre ainsi que le nombre de fois qu'une valeur a été tracée est également préservé et écrit dans le fichier.

4.4.2 Analyse du flot de contrôle

L'analyse du flot de contrôle étudie les différents chemins d'exécution que peut prendre le programme. Cette analyse est complémentaire à l'analyse contextuelle puisqu'elle est intraprocédurale. Elle est utile lorsque l'on bruite les calculs flottants avec MCA car ceux ci peuvent faire apparaître des branchements instables. Un branchement instable est une branche dont la condition est vérifiée ou invalidée d'une exécution à l'autre comme le montre l'exemple de la figure 4.13.

Pour détecter les branchements instables nous utilisons l'outil gcov de llvm-cov qui permet de faire de la couverture de code. Il suffit pour cela de rajouter le mot clef `-coverage` durant la compilation. La couverture de code fonctionne ensuite sur le mode incrémental, chaque fois qu'une

```
1  int main() {
2
3   double a = 0.0;
4   for (int i=0; i < 200; i++) {
5    a += (i%2) ? 0.1 : -0.1;
6   }
7
8   float result = -1;
9   if (a >= 1.0) {
10   result = 1;
11  } else {
12   result = 0;
13  }
14
15  return 0;
16 }
```

FIGURE 4.13 – Illustration d'un code avec un branchement instable.

instruction est exécutée un compteur spécifique à cette instruction est incrémenté. Ces compteurs sont enregistrés dans des fichiers `*.gcda`. En exécutant plusieurs fois le code on peut donc savoir combien de fois chaque branche a été prise. Sans aléatoire dans le code d'origine, les compteurs de la version originale et de la version bruitée doivent être les mêmes. La figure 4.14 montre la différence de couverture entre l'exécution IEEE-754 et MCA.

4.4.3 Analyse du flot de données

L'analyse du flot de données s'attache à analyser l'enchaînement des instructions qui impactent une variable ou un calcul en particulier. L'analyse de flux de donnée n'a pas été implémentée mais les moyens théoriques pour le faire ont été étudiés. Nous avons retenue deux méthodes :

La taint analysis [138, 161] suit les calculs affectés par un ensemble de variables *teintées* par l'utilisateur. Les règles de propagation pour le coloriage sont définies par une politique de coloriage donnée par l'utilisateur. Les variables teintées étant dans notre cas les variables bruitées par MCA. En suivant le flot de teintage, on peut alors reconstruire le chemin de contamination qui représente les variables affectées par l'erreur aléatoire introduite. L'avantage de cette méthode est que l'on suit uniquement les variables potentiellement entachées d'erreurs numériques. L'inconvénient est qu'il faut mettre en place une shadow memory.

L'exécution symbolique est l'autre piste intéressante. Cette méthode rejoint les méthodes d'analyse statique présentées dans la section 1.3.1. Pour rappel, ces méthodes travaillent sur l'analyse statique du code, c'est-à-dire sans l'exécuter celui ci, afin de construire une abstraction mathématique du programme. L'avantage est que les informations sont données statiquement sans surcoût d'exécution [3]. De plus les informations données sont rigoureuses. L'inconvénient est que les analyses statiques raisonnent a priori sans connaître les valeurs possibles prises par un programme. On peut

3. Le coût se trouve répercuté sur le temps de compilation.

```
function main called 50 returned 100% blocks
     executed 88%
  -:  4:int main() {
  -:  5:
 50:  6: double a = 1.0;
20100:  7: for (int i=0; i < 200; i++) {
branch 0 taken 99%
branch 1 taken 1%
10000:  8:  a += (i%2) ? 0.1 : -0.1;
10000:  9: }
  -:  10:
 50:  11: int positive = -1;
 50:  12: if (a < 1.0) {
branch 0 taken 0%
branch 1 taken 100%
#####:  13:  positive = 1;
#####:  14: } else {
 50:  15:  positive = 0;
  -:  16: }
  -:  17:
 50:  18: return 0;
  -:  19:}
```

(a) Exécution IEEE

```
function main called 50 returned 100% blocks
   executed 100%
  -:  4:int main() {
  -:  5:
 50:  6: double a = 1.0;
20100:  7: for (int i=0; i < 200; i++) {
branch 0 taken 99%
branch 1 taken 1%
10000:  8:  a += (i%2) ? 0.1 : -0.1;
10000:  9: }
  -:  10:
 50:  11: int positive = -1;
 50:  12: if (a < 1.0) {
branch 0 taken 56%
branch 1 taken 44%
   28:  13:  positive = 1;
   28:  14: } else {
   22:  15:  positive = 0;
  -:  16: }
  -:  17:
 50:  18: return 0;
  -:  19:}
```

(b) Exécution RR $t = 53$.

FIGURE 4.14 – Différence de couverture entre 50 exécutions IEEE et 50 exécutions MCA. L'instabilité de la branche est révélée par l'introduction du bruit.

alors vite se retrouver avec une explosion combinatoire du nombre de chemins d'exécution possibles sur des gros codes de calculs ce qui rend l'analyse compliquée.

4.5 La génération de traces numériques

La génération des traces est automatisée par la commande `veritracer launch` qui permet de lancer plusieurs exécutions d'un programme en parallèle. Le résultat de chacune des exécutions est mis dans un dossier séparé pour éviter toute collision.

La figure 4.15 présente l'arborescence d'une exécution avec la commande `veritracer launch`. Le répertoire principal (`xp`) contient un dossier pour chaque exécution MCA numéroté de 1 à n. Chacun de ces dossiers contient le fichier `veritracer.dat` avec les données brutes générées durant exécution, le fichier `backtrace.dat` qui contient les valeur de hachage des contextes d'appels de chaque variable tracée et le fichier `backtrace_map.dat` qui contient l'association entre valeur de hachage et contexte d'appels au format texte lisible par un humain.

Pour diminuer la taille des fichiers, un format de sortie binaire a été implémenté dans lequel les fichiers `veritracer.dat` contiennent, non pas la représentation American Standard Code for Information Interchange (ASCII) des valeurs mais leur encodage hexadécimale. Or la taille de l'encodage dépend du type de la variable analysée (un binary32 sera codé sur 4o contre 8o pour un binary64). Le nombre de bits à lire n'est donc connu qu'au moment où on lit le début de la ligne [4]. Le problème est donc que la parallélisation naturelle de la lecture est cassée. Pour contourner ce problème, nous avons introduit du padding. Cette technique consiste à introduire des octets vides pour aligner la mémoire ou éviter les conflits d'accès en mémoire cache [124]. Ainsi les lignes sont de taille fixe.

4. car celle ci commence par la taille du type.

```
.
|---locationInfo.map
|---test.c
|---xp
  |---1
  |   |---backtrace.dat
  |   |---backtrace_map.dat
  |   |---veritracer.dat
  |---2
      |---backtrace.dat
      |---backtrace_map.dat
      |---veritracer.dat
```

FIGURE 4.15 – Arborescence des fichiers de traces pour 2 exécutions. Le fichier `locationInfo.map` contient l'association entre valeur de hachage et information de débugage des variables tracées. Ce fichier est généré au moment de la compilation et les suivants à l'exécution. Les fichiers `veritracer.dat` contiennent les valeurs des variables tracées. Les fichiers `backtrace.dat` contiennent les valeurs de hachages des contextes d'appels de chaque variable. Enfin les fichiers `backtrace_map.dat` contiennent l'association entre valeur de hachage et contexte d'appels.

4.6 Le post-traitement

Une fois les traces générées, il faut ensuite les agréger. Le problème est que, comme il a été évoqué en section 4.4, l'analyse des traces est asynchrone donc les chemins d'exécutions ne sont pas connus et peuvent être différents d'une exécution à l'autre lorsqu'il y a branchements instables (cf. figures 4.13 et 4.14). Pour pallier ce problème, nous utilisons les informations des contextes d'appels. Cela évite de fusionner des traces n'ayant pas le même flot de contrôle. Une première analyse regroupe les traces qui possèdent le même contexte d'appels. Ensuite les traces sont fusionnées au sein de chaque groupe pour donner un fichier CSV.

4.6.1 Séparation des contextes d'appels

Chaque fichier `backtrace.dat` contient la suite des contextes d'appels de l'ensemble des variables tracées au cours de l'exécution. Le module `veritracer analyze` lit chacun des fichiers et détecte les divergences comme l'illustre la figure 4.16. Les exécutions ayant la même suite de contexte d'appels sont regroupées au sein de la même backtrace. Ce qui permet d'analyser les traces ayant empruntées un même chemin d'exécution. Un fichier est généré pour chaque backtrace différente.

4.6.2 Reconstruction des séries temporelles

L'utilisation des contextes d'appels permet de regrouper les traces ayant le même flot d'exécution. Une fois ces petits groupes constitués, on peut agréger leur contenu. Les fichiers de traces sont au format binaire. Chaque ligne est constituée des champs suivants : type de la variable, sa valeur de hachage, une estampille temporelle, son adresse et sa valeur numérique. Un exemple du fichier `veritracer.dat` est donné sur la figure 4.17.

Le but de la reconstruction est de calculer les propriétés statistiques suivantes pour chaque invocation : maximum, minimum, médiane, moyenne, écart-type. Le fichier final contient une ligne pour chaque invocation. Chaque ligne contient la valeur de hachage, une estampille temporelle, le

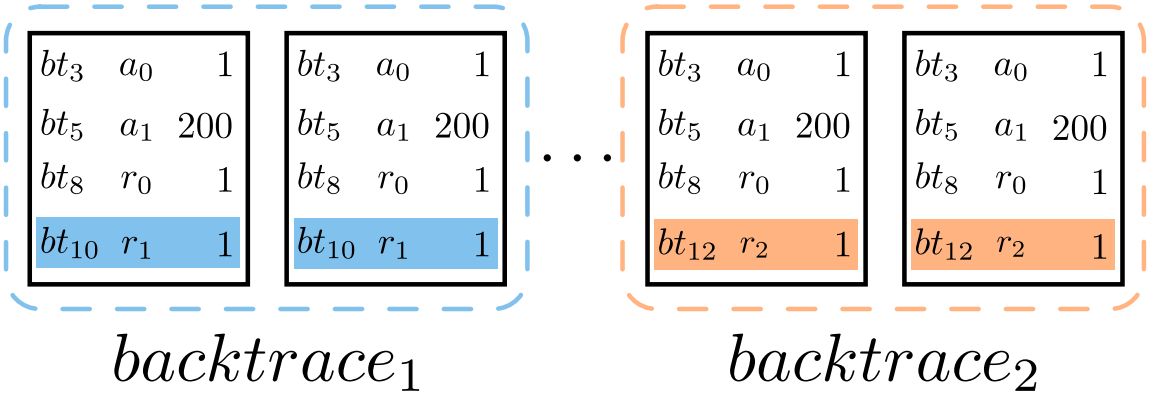


FIGURE 4.16 – Analyse des contextes d'appels et détection des divergences sur l'exécution du code figure 4.13. Les exécutions possèdent une liste de contexte d'appels différents suivant le résultat de la condition ligne 9. Les traces sont regroupées suivant leur liste de contextes d'appels. La ligne surlignée en bleu et en orange marque la divergence entre les deux groupes.

```
binary64 1569063160155255 13598624150298961041 (nil) -0x1.bfff0e748cefdp+6
binary64 1569063160155256 8477471546868834907 (nil) 0x1.17ffc781342dep+6
binary64 1569063160155256 17990129320275461441 (nil) 0x1.17ff8de60cd8ap+7
```

FIGURE 4.17 – Exemple du fichier `veritracer.dat` au format texte.

type de la variable, les statistiques mentionnées précédemment et le nombre de chiffres significatifs calculé suivant la définition 2.6.1 de Parker.

Les statistiques sont calculées pour chaque ligne du fichier, une ligne étant égale à une invocation. Puisque les fichiers partageant la même backtrace (cf.figure 4.16) contiennent le même nombre d'invocations et donc de lignes, la parallélisation de l'analyse est facile. On attribue un nombre de lignes à traiter pour chaque thread qui chacun agrège les lignes sur de sa section sur chacun des fichiers comme le montre la figure 4.18.

4.7 La visualisation de la qualité numérique au cours du temps

Les sections 4.5 et 4.6 ont présentées comment Veritracer génère et analyse les traces. Nous allons maintenant montrer dans cette section comment visualiser les informations recueillies. La visualisation des variables se fait avec la commande `veritracer plot` qui prend un fichier CSV généré par la commande `veritracer analyze`.

4.7.1 Modes de visualisation

Mode par horloge

C'est le mode par défaut qui est basé sur les estampilles temporelles collectées durant le traçage. L'avantage de ce mode est que les dépendances entre variables sont facilement visualisables puisqu'elles apparaissent dans l'ordre d'apparition.

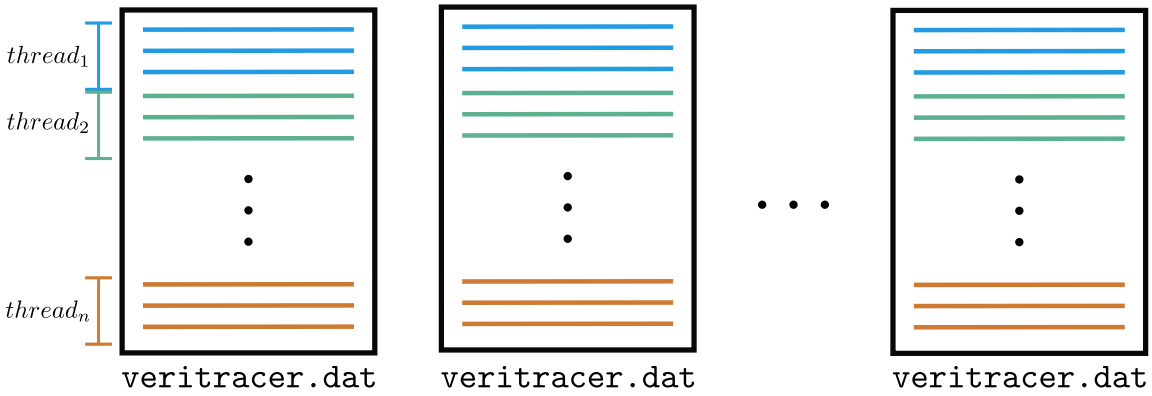


FIGURE 4.18 – Lecture parallèle des fichiers `veritracer.dat` partageant une même backtrace. Une même backtrace garantie que les fichiers ont le même nombre de ligne ce qui facilite la parallélisation de la lecture.

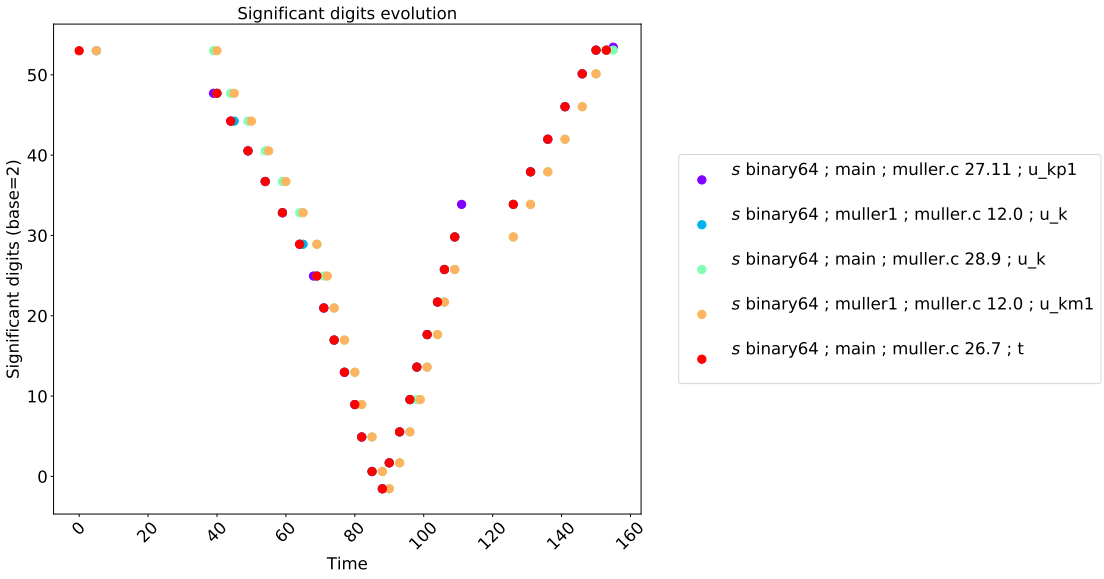


FIGURE 4.19 – Figure automatiquement générée par `veritracer plot`. Présente le nombre de chiffres significatifs des variables sur la suite de Muller (fig. 4.1) avec les estampilles temporelles. On voit que ce mode met bien en relief des dépendances entre variables.

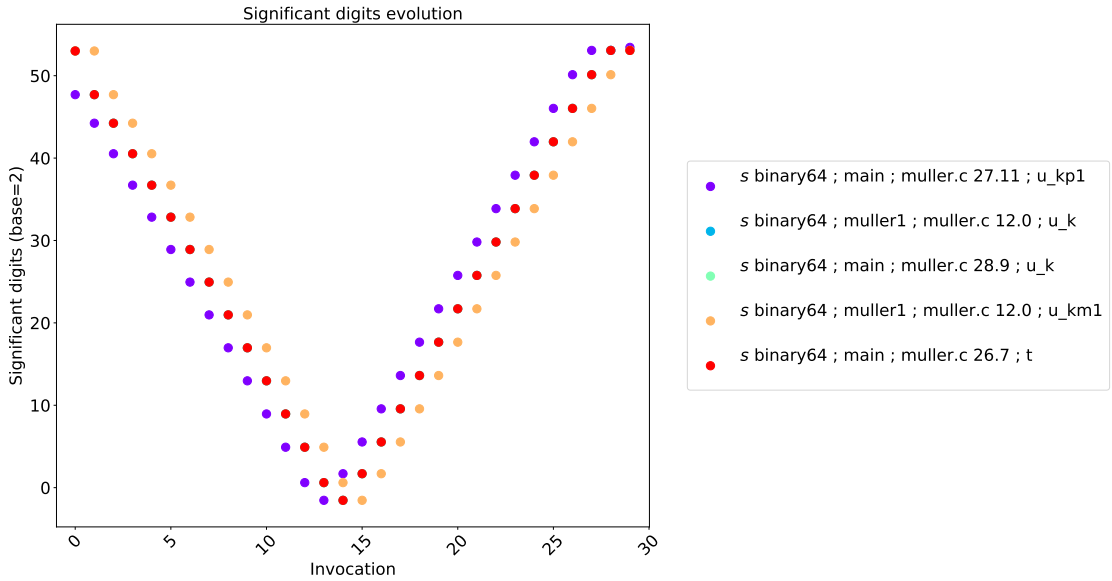


FIGURE 4.20 – Figure automatiquement générée par `veritracer plot`. Présente le nombre de chiffres significatifs des variables sur la suite de Muller (fig. 4.1) en mode par invocation. On voit que ce mode se prête bien aux schémas itératifs.

Mode par invocation

Dans ce mode les variables sont ordonnées par ordre d'invocation à partir de 1. Les variable tracées à des moments différents se retrouvent ainsi sur le même axe. Ce mode est utile lorsqu'on s'intéresse à une séquence d'appels comme dans le cas d'une boucle par exemple.

Mode *jitter*

Le mode par superposition (*jitter*) imprime pour une variable donnée, la valeur prise par chacune des réalisations MCA. Permet de voir rapidement les effets de divergence numérique pour une variable en particulier. C'est le mode qui a été utilisé pour générer la figure 4.2.

4.7.2 *veritracer-gui* : Vers un IDE Java intégré

Ces travaux ont été réalisé en collaboration avec Damien Thenot durant son stage au laboratoire Li-Parad sous la direction de Pablo De Oliveira et moi même.

Nous avons développé une interface graphique pour regrouper l'ensemble des informations disponibles par Veritracer au sein d'une seule application. Nous nous somme basé sur l'application `tracecompass` de l'Integrated Development Environment (IDE) Eclipse qui offre une infrastructure performante et ergonomique pour lire et analyser les traces d'un système.

Cette interface graphique est organisée selon des vues. Chaque vue permet d'interagir avec la trace pour afficher les informations souhaitées.

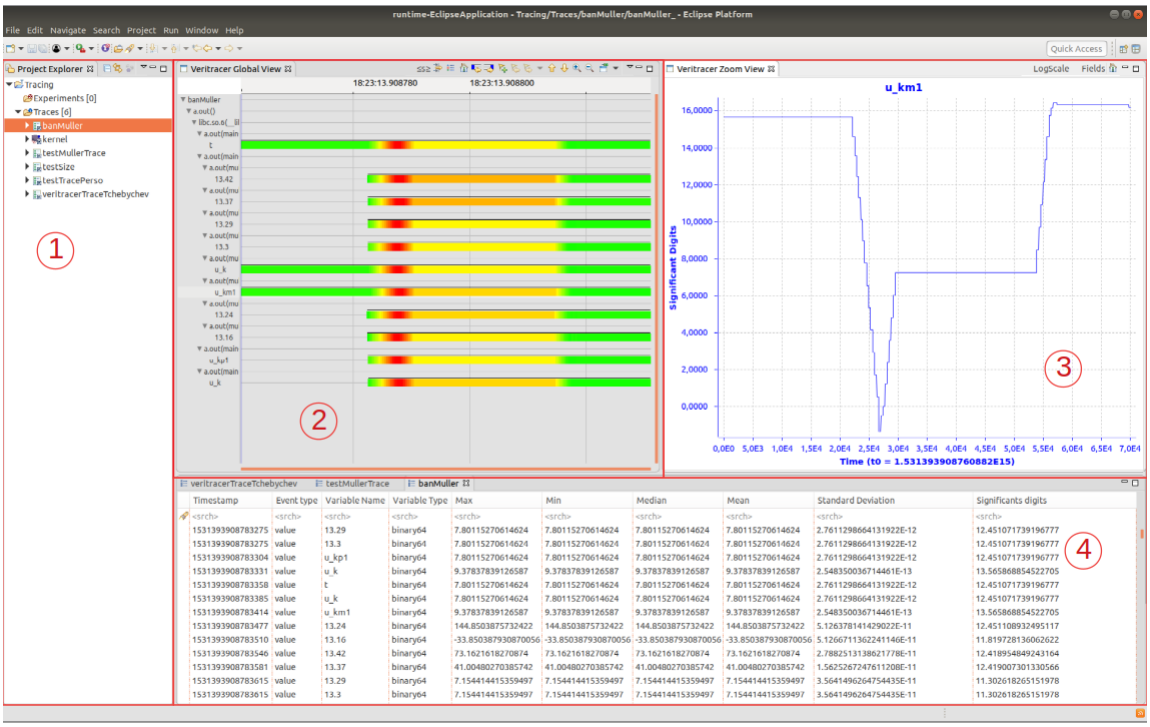


FIGURE 4.21 – Capture d'écran de l'interface graphique développée à l'aide de tracecompass. Les différentes vues correspondent à : 1) l'explorateur de projet qui permet de naviguer à travers différentes traces. 2) la vue globale qui affiche pour chaque variable, le nombre de chiffres significatifs (s) au cours du temps. La couleur indique la qualité de s, vert signifiant un nombre maximal et rouge un nombre minimal de chiffres significatifs. 3) La vue zoom qui permet de voir précisément la valeur de s au cours du temps. La moyenne et l'écart-type de la valeur sont également disponibles. 4) La vue des évènements qui regroupe l'ensemble des données sur les traces.

1. L'explorateur de projet permet de suivre plusieurs traces et de les sélectionner dans la vue active.
2. La vue globale affiche le nombre de chiffres significatifs pour l'ensemble des variables tracées avec un code couleur pour rapidement identifier les zones dont la qualité numérique est mauvaise (en rouge).
3. La vue zoom permet d'étudier de plus près la qualité numérique d'une variable en particulier. La vue est interactive et permet de zoomer et se déplacer à travers le temps.
4. La vue des évènements qui présente l'ensemble des informations disponibles dans les traces. Cette vue est également interactive et permet de sélectionner un ensemble d'évènement à afficher sur la vue globale ou la vue zoom. De plus, une barre de recherche permet de filtrer les évènements suivant des expressions régulières pour rechercher des variables en particulier.

Une description complète de l'interface graphique ainsi que les codes sources associés sont disponibles sur la branche Veritracer du projet verificarlo : github.com/verificarlo/verificarlo/tree/-veritracer/postprocess/visualizer.

Veritracer-gui marche sur la suite de Muller (cf. figure 4.2), l'évaluation d'un polynôme de tchebychev ainsi que la fonction `simp_gen` d'ABINIT qui sera présenté dans la section 4.8 suivante.

Nous avons présenté le fonctionnement de Veritracer sur des exemples de petite taille qui ont démontrés l'intérêt d'analyser la qualité numérique des calculs flottants au cours du temps. Néanmoins l'objectif de cette thèse est de développer des outils utilisables dans un contexte HPC. Pour rappel, le contexte HPC implique des codes intensifs en calculs flottants développés pour être exécutés sur des machines massivement parallèles. On souhaite donc pour identifier rapidement les sources d'erreurs numériques parmi des centaines de milliers de ligne de code. Nous allons dans la section 4.8 qui suit démontrer comment utiliser Veritracer pour débuguer le code HPC de chimie quantique ABINIT.

4.8 Le code de calcul ABINIT

ABINIT [73] est un programme de chimie quantique développé principalement par le CEA et l'université Catholique de Louvain ainsi que de nombreux collaborateurs industriels et académiques. ABINIT calcule à partir des équations de Density Functional Theory (DFT) les propriétés optiques, mécaniques ou vibrationnelles d'un matériau. ABINIT travaille sur des compositions chimiques variées allant des molécules aux nanostructures en passant par les solides (et largement utilisé par les mondes industriel et académique). ABINIT est un code FORmula TRANSlator (FORTRAN) composé de 1307 fichiers, 5835 fonctions pour 851615 lignes de codes [5]. L'ensemble des calculs flottants utilisent le format binary64.

4.8.1 Calcul de distance inter-atomique sur H_2

Nous allons présenter ici un premier cas d'étude sur l'atome de dihydrogène (H_2). Ce cas d'étude va nous permettre de présenter les éléments utiles à l'analyse qui seront repris dans la suite de ce chapitre.

5. Version 8.3 d'ABINIT.

Ce cas d'école consiste à calculer la distance inter-atomique optimale, c'est-à-dire la distance entre les deux atomes d'hydrogène minimisant l'énergie globale du système. ABINIT propose plusieurs méthodes dont l'une d'entre elle consiste à calculer l'énergie du système pour plusieurs distances et trouver le minimal en interpolant les points. L'énergie du système physique que l'on note `etotal` est le principal résultat physique calculé par ABINIT. Cette énergie est exprimée en Hartree. Ce résultat nous servira de critère de référence dans la suite de nos expérience.

Nous avons bruité entièrement l'application pour tester sa stabilité. La figure 4.22 présente quatre exécutions avec des précision virtuelles variant de 52 à 41. On voit que le calcul peut absorber jusqu'à 7 bits d'erreur sans que l'on constate une dégradation du calcul. À partir de 44 bits, on constate qu'une partie des énergies calculées n'est sont plus significatif avec des ordres de magnitude de 10^{12} contre -1 pour l'exécution d'origine. À partir de 41 bits, plus aucune énergies calculées n'est dans le bon ordre de magnitude.

Cette première analyse montre que l'instrumentation avec Veritracer est robuste et passe à l'échelle sur le code ABINIT puisque nous avons uniquement remplacé le compilateur FORTRAN par Verificarlo. L'analyse nous a aussi montré qu'ABINIT pouvait absorber jusqu'à 7 bits de bruit sur l'ensemble des calculs flottants sans que les résultats soient dégradés.

4.8.2 La Pérovskite : un cas plus dur

Le cas test sur la molécule de dihydrogène a servi de premier test pour vérifier l'instrumentation avec Veritracer. Ayant montré que l'instrumentation était robuste et que le code pouvait absorber jusqu'à 7 bits de bruit, nous nous sommes ensuite tournés vers des cas plus complexes, faisant appel à des modèles physiques plus sophistiqués que ceux mis en jeux dans H_2.

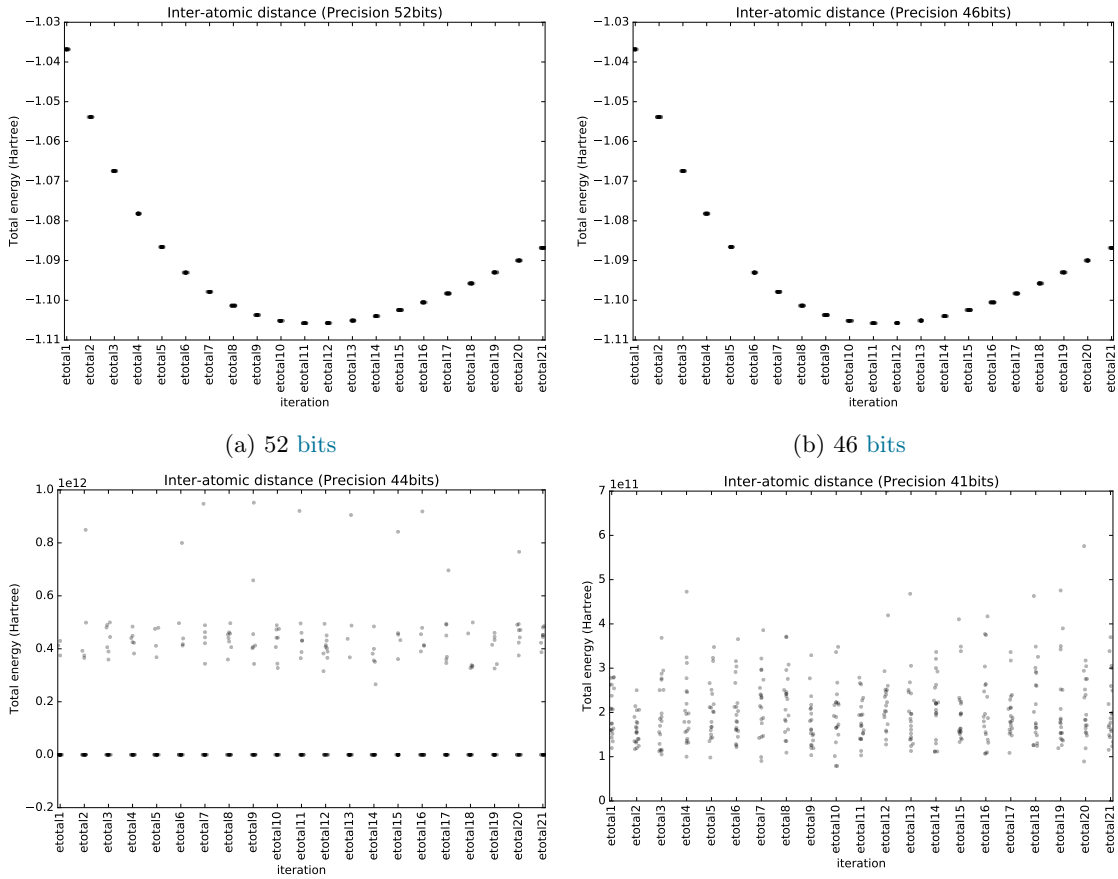
Nous avons pour cela étudié le cas de la molécule de Pérovskite. La Pérovskite est une espèce minérale nommée en l'honneur du minéralogiste russe Lev Alexeïevitch Perovski. Elle est surtout très analysée pour la conception de panneaux solaires en raison de son absorption de la lumière. Parmi les différentes variantes de Pérovskite, nous avons étudiée celle de $BaTiO_3$ dont une illustration est donnée sur la figure 4.23.

4.9 Les méthodes de réduction de dimensionnalité

Nous allons appliqué les méthodes de réduction de dimensionalité présentées dans la section 5.1.1. Le but des ces méthodes est d'identifier rapidement les fonctions ayant une contribution sur l'erreur globale. Nous commencerons par présenter la méthode de filtrage qui permet d'éliminer les fonctions n'ayant pas de contribution directe sur l'erreur globale. Puis verrons la méthode de classification qui permet d'ordonner les fonctions suivant leur degré de sensibilité. Cette notion que nous définirons permet de hiérarchiser les fonctions suivant leur niveau de criticité.

4.9.1 Filtrage

L'étape de filtrage est la première étape pour réduire l'espace de recherche. Elle consiste à ne garder que les fonctions qui ont une contribution directe sur l'erreur globale. L'idée est de pousser les critères de sensibilité au maximum pour réduire l'espace de recherche et supprimer les fonctions qui n'impactent pas directement le résultat final d'`etotal`. Nous voulons étudier un code sans connaissance préalable pour rapidement identifier des fonctions d'intérêts, c'est-à-dire des fonctions ayant au moins une contribution directe sur l'erreur d'`etotal`.



(a) 52 bits

(b) 46 bits

(c) 44 bits

(d) 41 bits

FIGURE 4.22 – Différent profils d'énergie en fonction de la magnitude de bruit injecté sur l'ensemble des opérations d'ABINIT pour le cas H_2. On prend un échantillon de 32 exécutions MCA en mode RR pour chaque précision virtuelle testée. On superpose les 32 énergies calculées par ABINIT (axe y) pour chaque distance inter-atomique (axe x). On constate une dégradation des résultats à partir d'une précision virtuelle de 44 bits. Les résultats ne sont plus dans l'ordre de grandeur de l'énergie attendue avec des résultats de l'ordre de 10^{12} Hartree contre -1 pour le résultat de référence. L'application peut absorber jusqu'à 7 bits de bruit sans que l'on constate une dégradation des résultats.

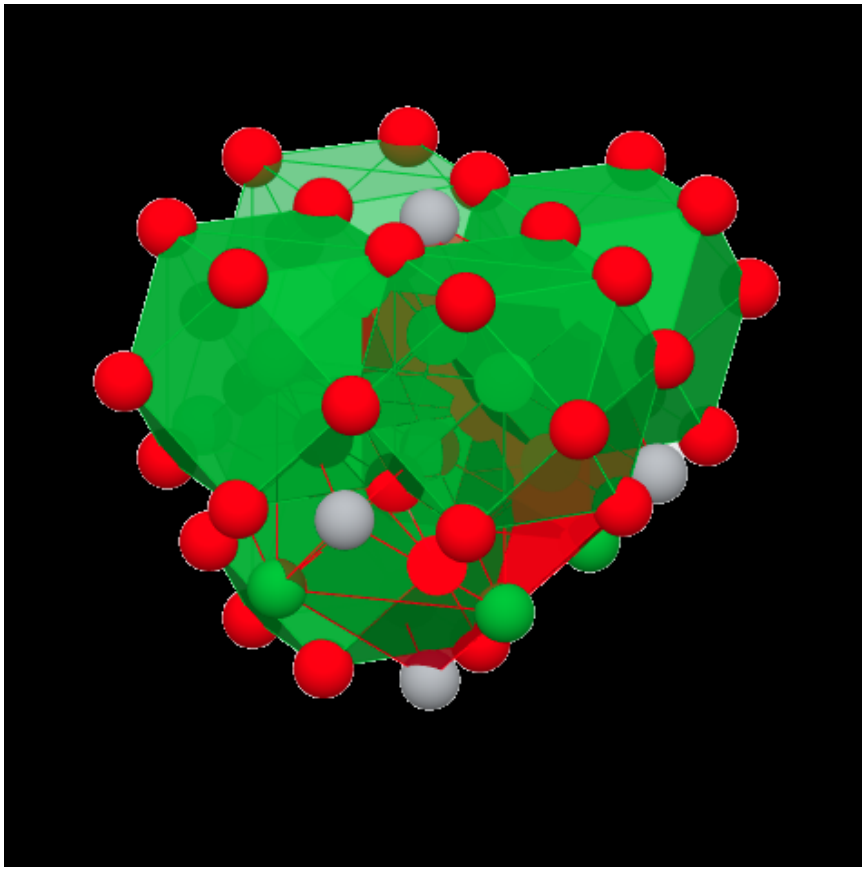


FIGURE 4.23 – Illustration de la molécule Pérovskite $BaTiO_3$ tirée du materialproject [147]. Les points verts, rouges et gris représentent respectivement les atomes de Baryum, d'Oxygène et de Titane. Les faces représentent les liaisons électriques.

Principe

Pour réduire l'espace de recherche nous proposons d'analyser les fonctions une par une. Pour cela nous instrumentons chacune des fonctions séparément et produisons un exécutable pour chaque fonction (une fonction = une version). Chaque version est ensuite exécutée en introduisant un bruit d'amplitude maximale ($t = 1$). L'idée est de détecter les fonctions dont la contribution directe à l'erreur globale n'est pas nulle, autrement dit les fonctions ayant un impact sur l'erreur g lobale. Cette étape permet d'éliminer les fonctions qui ne contribuent pas au résultat final comme par exemple les fonctions qui n'utilisent pas d'arithmétique flottante.

Pour accélérer les versions instrumentées d'ABINIT, nous utilisons le backend `BITMASK`. Pour rappel le backend `BITMASK` peut être vu comme un modèle d'injection de bruit grossier qui offre un compromis entre performance d'exécution et qualité du bruit injecté, puisque le bruit injecté par le ce backend est biaisé peu importe le mode utilisé (cf. section 3.2.3). Chaque version d'ABINIT est exécutée en mode ZERO et en mode ONE avec une précision virtuelle $t = 1$. Si le critère de validité n'est pas respecté pour l'une des deux exécutions, la fonction est étiquetée comme *critique*.

Critère de validité numérique

Nous prenons comme critère de validité numérique la comparaison avec la valeur de référence d'`etotal`, c'est-à-dire la valeur calculée par le programme sans bruitage dans les formats IEEE-754 d'origine. Suivant si ce critère est valide ou pas, nous étiquetons les fonctions comme *critique* ou *non critique*. Cela suivant que la valeur d'`etotal` a été modifiée ou pas. De même les fonctions pour lesquelles l'exécution ne converge pas ou crash sont étiquetées *critiques*.

Définition 4.9.1 (Fonction critique)**.** Une fonction f est étiquetée *non-critique* si

$$f^{critique} \equiv |etotal_{reference} - \tilde{etotal}| = 0$$

où $\tilde{etotal}$ est le résultat de l'exécution bruité.

Résultats

Les résultats de cette analyse montrent que seules 88 fonctions sur les 2952 [6] d'origines sont étiquetées comme *critique*. Soit une réduction de l'espace de recherche de **33**×. Cela montre qu'une grande partie des fonctions d'ABINIT ne contribue pas directement à l'erreur globale. L'approche par filtrage permet donc de réduire l'espace de recherche et d'identifier les fonctions à haut potentiel d'erreur globale.

Ce facteur de réduction peut néanmoins être nuancé par le point suivant. L'analyse par filtrage est dépendante du jeu de données utilisée puisque suivant la nature de l'objet étudié, les modèles physiques utilisés ne seront pas les mêmes. En effet, ABINIT est divisé en plusieurs modules de calculs et chacun de ces modules est appelé suivant la physique modélisée. Ainsi la molécule $BaTiO_3$ mobilise des calculs supplémentaires par rapport au cas H_2. De plus, pour deux jeux de données qui utilisent les mêmes modèles physiques (et donc les mêmes modules de calculs) peuvent avoir des comportement numérique différent. Pour améliorer le filtrage, nous pensons qu'il serait intéressant d'avoir des jeux de données représentatifs pour couvrir l'ensemble des calculs d'ABINIT, comme cela est fait pour la couverture de code par tests unitaire.

6. Nous n'avons retenu que les fonctions ayant au moins une opération flottante, soit 2952 fonctions sur les 5835 d'origines.

4.9.2 Classification

L'étape de classification intervient après l'étape de filtrage pour analyser plus finement le comportement de chaque fonction. Cette méthode a pour but d'ordonner les fonctions suivant leur degré de sensibilité.

Principes

Une fois l'ensemble de fonctions *critiques* trouvées, nous souhaitons pouvoir donner une estimation de la sensibilité numérique d'une fonction, c'est-à-dire pouvoir une relation d'ordre pour comparer les fonctions et hiérarchiser les besoins. D'un point de vu développeur, il peut en effet être plus important de d'abord réparer les fonctions très sensibles d'un point de vue numérique que d'optimiser la précision des fonctions les moins sensibles.

Pour cela nous introduisons la notion de degré de sensibilité d'une fonction.

Définition 4.9.2 (Degré de sensibilité)**.** Soit un exécutable E dans lequel une seule fonction f est instrumentée avec Veritracer. On définit le degré de sensibilité comme la précision virtuelle minimale pour laquelle l'exécution E n'invalide pas un critère de validité numérique.

Si l'on suit cette définition, les fonctions avec un haut degré de sensibilité sont donc des fonctions potentiellement plus sensibles à l'ajout de bruit numérique et qui requièrent des analyses plus approfondies. Au contraire, les fonctions ayant un faible degré de sensibilité sont des fonctions plus robustes pouvant potentiellement être passées dans un format de calcul plus réduit.

Suivant le même principe que la méthode de filtrage, nous construisons [7] une version d'ABINIT pour chaque fonction étiquetée *critique*.

Puisque nous avons moins de fonctions à explorer et qu'on cherche à avoir une analyse plus fine, nous utilisons le mode `RAND` qui offre une meilleure variabilité sur le bruit introduit que les modes déterministes ZERO et ONE. Cela nous donne une estimation plus précise sur la sensibilité au bruit numérique pour un surcoût plus faible que les backends `QUAD` et `MPFR`.

Chacune des versions est exécutée n fois avec le backend `BITMASK` et le mode `RAND` pour une précision virtuelle t fixée. On calcul ensuite la moyenne et l'écart-type des n valeurs d'`etotal` calculées par ABINIT. Pour que le critère soit validé, il faut que les n valeurs d'`etotal` soient égales à la valeur $etotal_{ref}$ de référence.

Pour trouver la précision minimale, on utilise une recherche dichotomique : Si le critère est vérifié cela signifie que la précision virtuelle t utilisée n'était pas assez basse pour perturber le résultat d'`etotal`. On descend donc la valeur de t et on recommence l'exécution. Et inversement si le critère est invalidé on augmente t jusqu'à trouver le t minimal.

Résultats

La figure 4.24 présente le résultat de la classification. Sur l'axe des abscisses sont présentées les précisions virtuelles minimales et sur celui des ordonnées les 88 fonctions étiquetées *critique*. La ligne rouge à la précision virtuelle 23 sépare les fonctions pouvant potentiellement être passées en binary32.

La première fonction de ce classement (`__m_ewald_MOD_ewald`) est intéressante puisque qu'elle a une contribution importante sur le résultat `etotal` mais elle présente un intérêt limité pour le modèle physique. En effet, cette fonction calcule une constante (la sommation d'Ewald) qui est ajoutée à

7. En pratique nous avons repris les mêmes exécutables que l'analyse de filtrage.

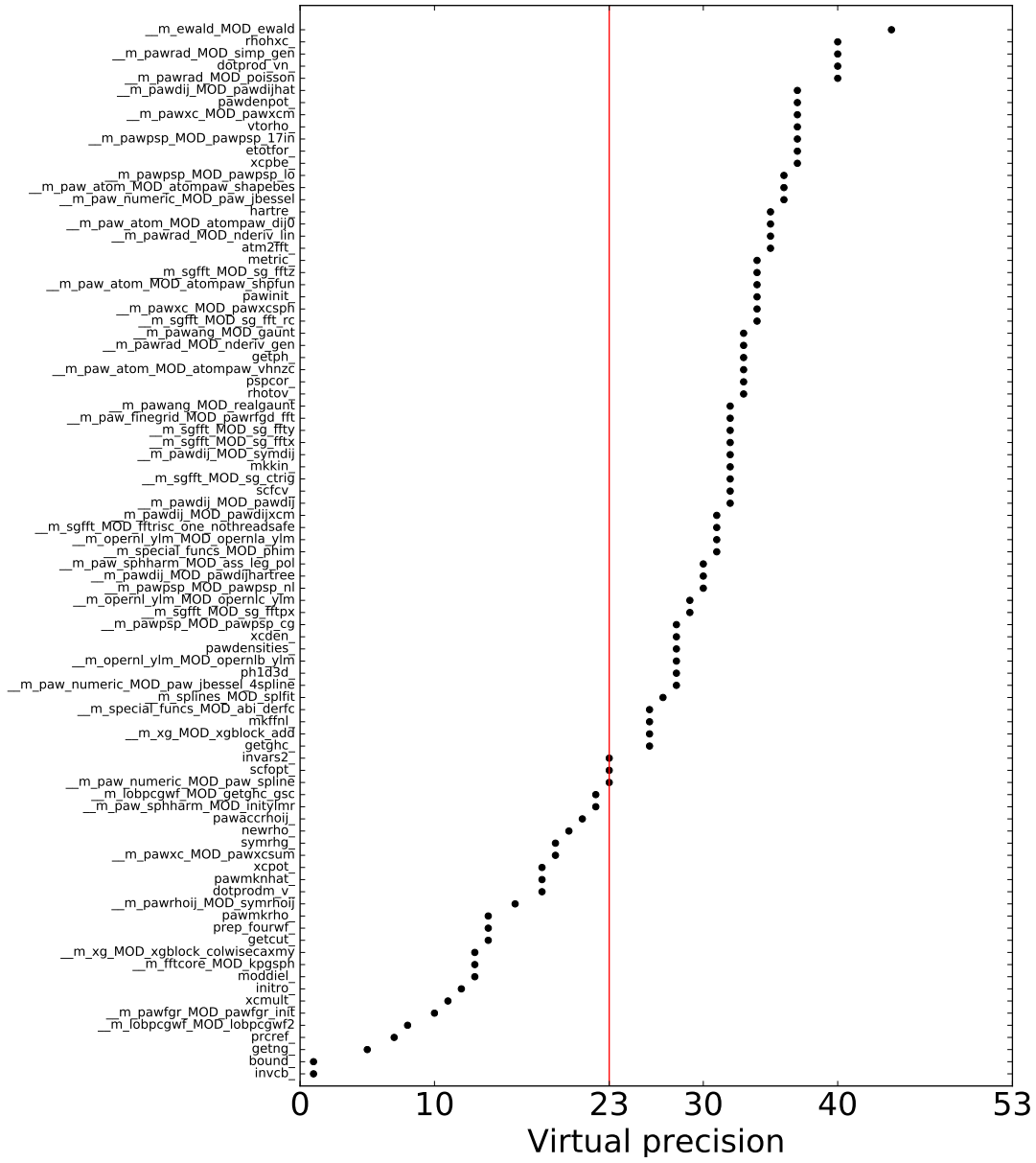


FIGURE 4.24 – Classification des 88 fonctions étiquetées *critiques*.

l'énergie totale du système. Or ce qui intéresse les physiciens c'est le potentiel d'activation qui correspond à la différence entre deux énergies d'une molécule et donc la constante doit disparaître avec la soustraction. Cela montre qu'il est important d'avoir un expert du domaine, en l'occurrence ici les physiciens, pour relier les calculs au modèle simulé.

La seconde fonction du classement `__m_pawrad_MOD_simp_gen` (notée `simp_gen` dans la suite) est intéressante puisque les physiciens d'ABINIT l'ont repéré indépendemment de nos travaux. Ils ont en effet identifié cette fonction comme ayant une contribution sur l'instabilité numérique du code. Sans a priori sur le code, notre approche a également su identifiée cette fonction comme particulièrement sensible au bruit numérique ce qui renforce le bien fondé de notre approche. Avec une importante contribution sur `etotal`, nous avons donc voulu étudier plus finement cette fonction ce qui fait l'objet de la section 4.10 suivante.

4.10 Étude détaillée de `simp_gen`

La fonction `simp_gen` est une fonction de calcul d'intégrales qui est utilisée à de nombreux endroits du code et qui fait parti des fonctions les plus sensibles d'ABINIT d'après notre méthode de classification (cf. section 4.9.2) Après une présentation de la méthode numérique, nous présenterons la qualité numérique de son résultat au cours du temps. Puis montrerons comment améliorer sa qualité numérique à l'aide d'EFTs. Enfin, nous terminerons avec une analyse de flot de données pour expliquer la perte de chiffres significatifs sur le résultat de `simp_gen`.

4.10.1 L'intégrale de Simpson

La fonction `simp_gen` calcul le résultat numérique d'une intégrale sur une grille 1-D généralisée à l'aide de la méthode de Simpson. Les méthodes d'intégration numérique consistent à diviser l'intervalle d'intégration en sous-intervalles puis à approximer localement la fonction f pour sommer la valeur numérique de chaque sous-intervalles. La méthode de Simpson utilise un polynôme quadratique pour interpoler la fonction f sur chacun des sous-ensemble du domaine.

$$\int_a^b f(x)dx = \frac{b-a}{6n} \sum_{i=0}^{n-1} (2f(x_i) + 4f(m_i))$$

avec m_i le milieu du segment $[x_i, x_{i+1}]$.

La méthode utilisée dans ABINIT permet de généraliser aux domaines non-cartésiens (polaire, logarithmique, ...).

Dans le code ABINIT, cette intégrale est exprimée par le code de la figure 4.25 dont la boucle principale est seulement présentée. Les différents paramètres sont `nn` qui représente le nombre de sous-intervalles du domaine sur lequel intégrer, `simp` l'accumulateur contenant la valeur numérique de l'intégral ; `f` un tableau contenant pour chaque indice i la valeur $f(i)$ de l'interpolation ; enfin `simfact` un tableau de même dimension que f qui contient les facteurs de chaque point.

On reconnaît une schéma similaire à celui du produit scalaire entre deux vecteurs $func$ et $simfact$. Cette similarité sera exploitée dans la section 4.10.3 pour améliorer la qualité numérique du résultat de `simp_gen`.

```
nn=int_meshsz
simp=zero
do ii=1,nn
 simp=simp+func(ii)*simfact(ii)
end do
```

FIGURE 4.25 – Coeur de boucle de la fonction `simp_gen`. Le calcul de l'intégrale peut se concevoir comme un produit scalaire entre la fonction à intégrer (`func`) et les facteurs d'intégration (`simfact`).

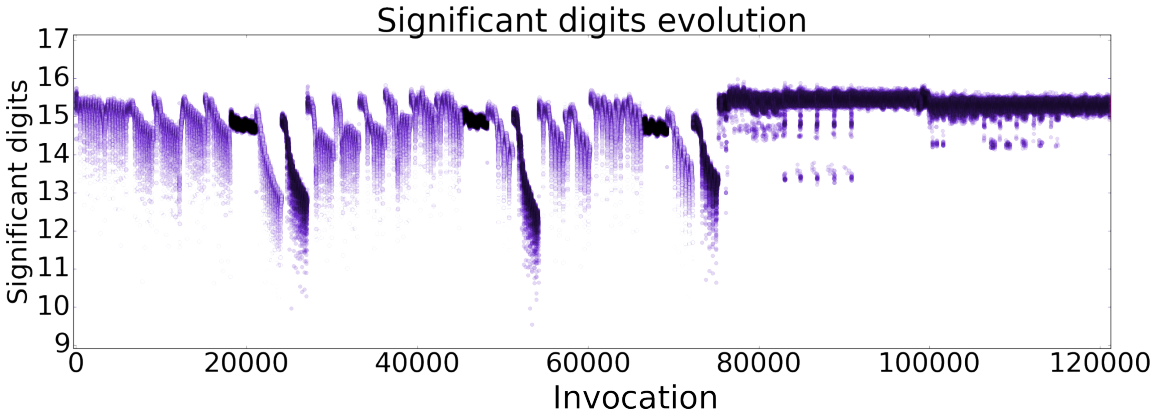


FIGURE 4.26 – Évolution du nombre de chiffres significatifs du résultat de la fonction `simp_gen` au cours du temps. On a jusqu'à 7 chiffres décimaux qui sont perdus suivant les invocations.

4.10.2 Vue globale de la qualité numérique

Nous avons voulu avoir dans un premier temps une estimation de la qualité numérique de `simp` au cours du temps. On a en effet vu en section 4.6 l'importance d'explorer la dimension temporelle des calculs (cf. figure 4.2). Cette première estimation va nous donné le profil général de la qualité des résultats de `simp_gen`. Nous avons pour cela compilé une version d'ABINIT n'instrumentant que la fonction `__m_pawrad_MOD_simp_gen` et ajouté une sonde pour la variable `simp`. Nous avons ensuite exécuté avec le backend `QUAD` en mode RR pour une précision virtuelle de 53 bits.

Le résultat du traçage est présenté sur la figure 4.26. La qualité des 120000 premières invocations de la fonction `simp_gen` est présentée sur cette figure. On remarque que différents profils apparaissent. Une bonne qualité des calculs avec environ 15 chiffres décimaux significatifs autour des invocations 20000, 50000 et 70000 et à partir de la 80000ème invocation. Des résultats plus contrastés pour les autres invocations.

Cependant, la lecture de la figure est difficile sans une compréhension du contexte. La figure 4.27 présente la figure 4.26 enrichie des contextes d'appels pour chaque point. Chaque couleur équivaut à un contexte d'appel et les points possédant la même couleur partagent le même contexte d'appels. Cela permet une identification rapide des différents profils de qualité numérique suivant le contexte d'appels. En effet, les points noirs et roses possèdent une meilleure qualité numérique que les points issus du contexte d'appels rouge.

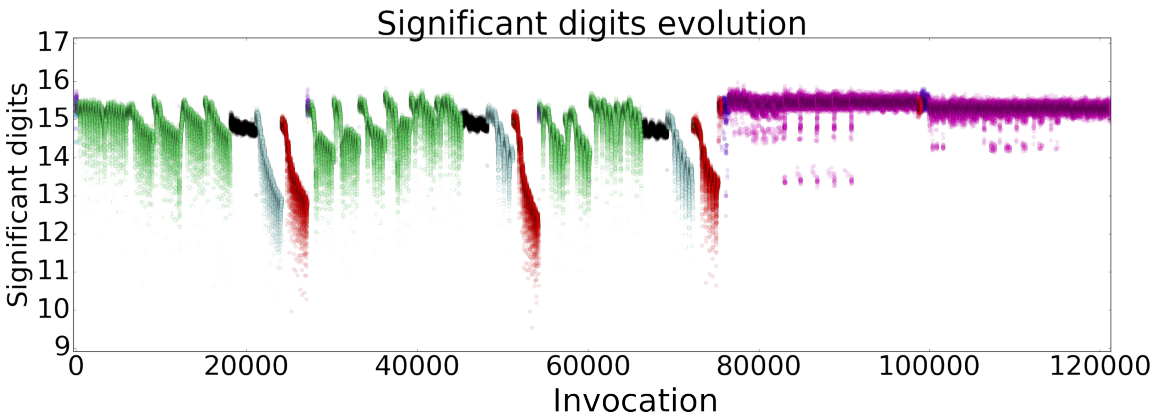


FIGURE 4.27 – Évolution du nombre de chiffres significatifs du résultat de la fonction `simp_gen` au cours du temps avec la coloration des contextes d'appels. Les points ayant la même couleur partagent le même contexte d'appels. On voit que la qualité du résultat dépend du contexte d'où la fonction a été appelée.

4.10.3 Amélioration de la qualité numérique

La figure 4.27 montre la qualité numérique du résultat de `simp_gen` au cours du temps. On voit que la qualité se dégrade sévèrement avec jusqu'à 7 chiffres décimaux significatifs de perdus. De plus, nous avons montré que cette fonction avait un impact significatif sur le calcul d'`etotal`. On souhaiterai donc améliorer la qualité numérique de cette fonction.

Nous avons montré dans la section 4.10.1 le lien entre la fonction `simp_gen` et le produit scalaire. Or, il existe une version dite compensée du produit scalaire.

Les méthodes compensées sont des algorithmes numériques permettant de compenser les erreurs numériques commises durant les calculs flottants. Ces méthodes compensées sont basées sur les Error Free Transformation (EFT) qui transforment le résultat d'une opération entre deux nombres flottants x et y en un résultat r plus un terme d'erreur e. En supposant qu'il n'y ait pas d'overflow ou d'underflow et en posant $\circ$ l'arrondi au plus près, alors :

$$\begin{aligned} &x, y, a, b \in \mathbb{F} \\ &x \diamond y = r + e \implies r = \circ(x \diamond y) \text{ et } |e| = |r - (x \diamond y)| \\ &\diamond \in \{+, -, \times\} \end{aligned}$$

Les deux blocs de base des EFTs sont `2Sum` [103] pour l'addition et `TwoProd` [51] pour la multiplication. Il existe une grande variété de méthodes compensées basées sur ces briques de bases dont une présentation détaillée est donnée dans [77]. Il existe également une méthode compensée pour le produit scalaire appelée `Dot2` [141] dont la définition est donnée sur la figure 4.28.

Dans `simp_gen` nous avons remplacé le produit scalaire naïf par sa version compensée. Nous avons ensuite tracé le résultat de l'exécution avec le backend `QUAD` et le mode RR pour une précision virtuelle de $t = 53$. A noté que le bruitage d'une méthode compensée est rendu possible par le fait qu'on utilise le mode RR. En effet, ce mode lorsqu'il est utilisé avec une précision virtuelle égale à la précision du format d'origine préserve les opérations exactes (cf.section 2.6.1). Or les EFTs

```
Dot2(x,y) =
  [p_1,s_1] = TwoProduct(x_1,y_1)
  for i=2:n
    [h_i,r_i] = TwoProduct(x_i,y_i)
    [p_i,q_i] = TwoSum(p_{i-1},h_i)
    s_i = ∘(s_{i-1}+(q_i+r_i))
  return ∘(p_n+s_n)
```

FIGURE 4.28 – Algorithme du `Dot2` en pseudo code. Calcul le produit scalaire compensé entre les vecteurs x et y de manière à ce que le résultat soit identique à un produit scalaire naïf utilisant le double de précision.

reposent sur des opérations exactes. L'utilisation du mode RR qui ne trouble pas les opérations exactes ne perturbe pas le fonctionnement de `Dot2`

La figure 4.29 présente l'évolution de la qualité numérique du résultat retourné par `simp_gen` au cours du temps avec l'utilisation du produit scalaire compensé `Dot2`. On voit que la qualité numérique de l'ensemble des points a été améliorée mis à part pour les points du contexte d'appels rouge. Les points faisant parti de ce contexte d'appels on vu leur qualité numérique être légèrement réhaussée mais restent néanmoins assujettis à des pertes de chiffres significatifs.

4.10.4 Analyse de flux de données en application

Pour comprendre pourquoi l'utilisation du `Dot2` n'a pas permis d'obtenir un nombre de chiffres significatifs à 16 sur le contexte rouge, nous avons analysé le flot de données du résultat de `simp_gen`. Le résultat de `simp_gen` est contenu dans la variable `simp_gen`. Nous avons effectué une taint analysis manuelle en suivant le chemin de contamination de `simp_gen`. Ce chemin est résumé sur la figure 4.30.

L'analyse du flot de donnée révèle que l'erreur introduite par MCA sur la valeur de la variable `simp_gen` lors des appels du contexte d'appels turquoise contamine le paramètre *func* des appels à `simp_gen` du contexte d'appels rouge.

Le premier appel à `simp_gen` dans la fonction `pawpsp_calc` (noté ♠) retourne le résultat `qq` entaché d'une erreur du au bruit MCA. Le second appel effectué dans la fonction `atompaw_shpfun` retourne le résultat `norm` qui est lui aussi entaché d'erreur et vient contaminer le vecteur `shapefunc`.

Ce vecteur `shapefunc` nommé `nhat` dans ♠ est multiplié par `qq` qui est entaché d'erreur. Ce vecteur est additionné à `tnvale` qui est passé à la fonction `pawpsp_cg` (noté ♣) et nommé `nr`. En suivant la trace de `nr`, on voit qu'il contamine les vecteurs `rnr` puis `ff`. Ce dernier est ensuite utilisé par la fonction `simp_gen` dans ♣. Le problème est que les valeurs de `ff` on été contaminées par du bruit MCA. L'erreur introduite sur les valeurs de `ff` détruit la compensation apportée par `Dot2` comme l'illustre la figure 4.31.

En suivant le flot des calculs, l'analyse de flot de données permet de comprendre la baisse de fidélité observée sur les points du contexte d'appels rouge et offre aux développeurs des pistes de réflexions. Ils ont ainsi des endroits du code qui interagissent entre eux sur lesquelles ils peuvent agir pour améliorer la qualité numérique de leur code.

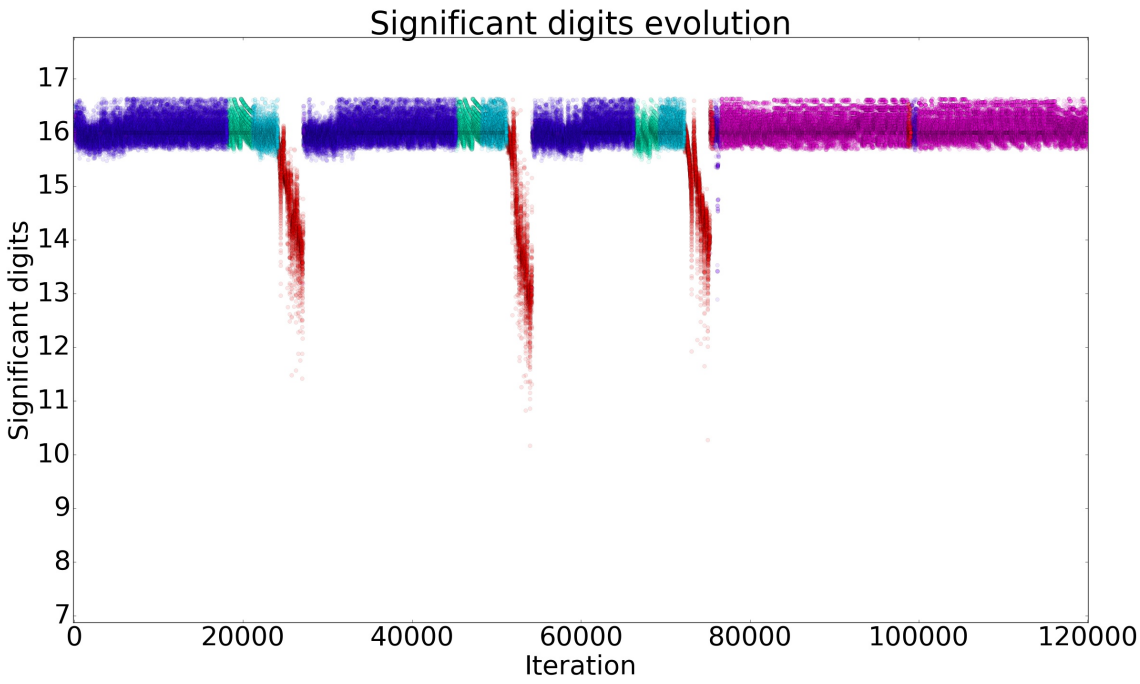


FIGURE 4.29 – Évolution du résultat de la fonction `simp_gen` au cours du temps avec l'utilisation du `Dot2`.

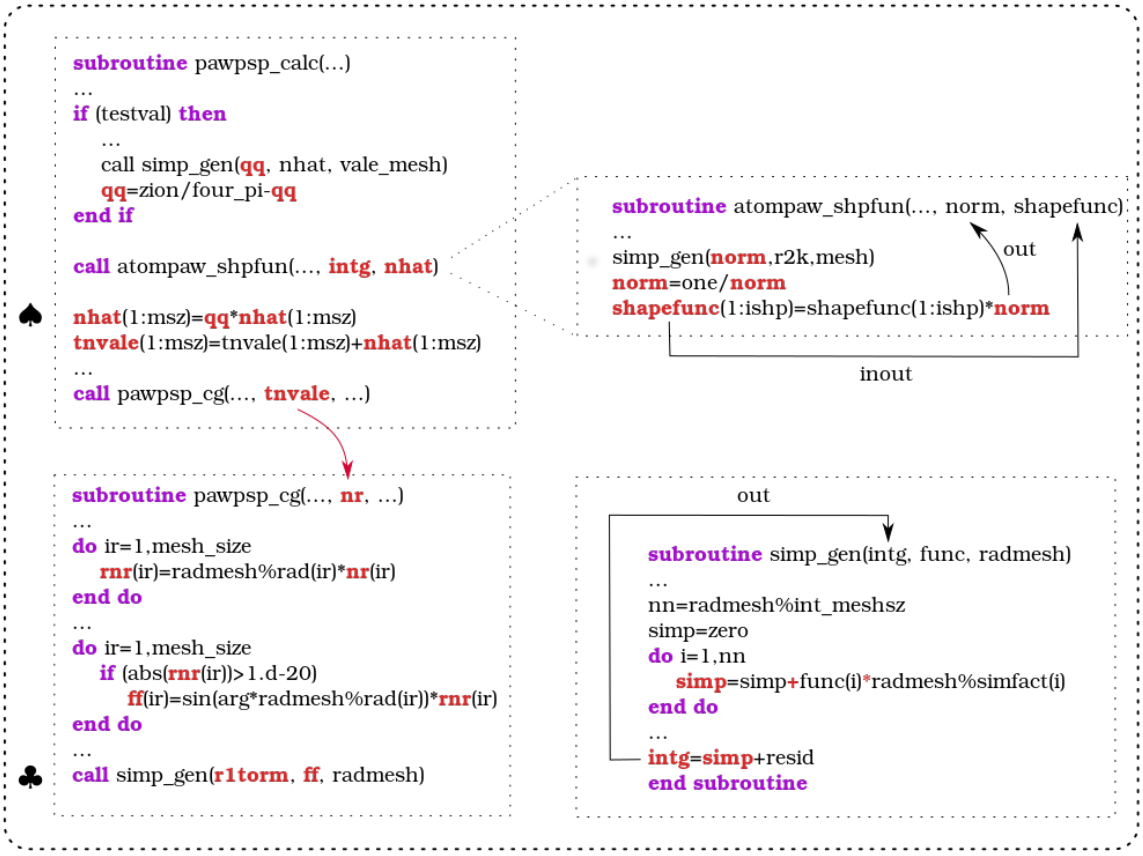


FIGURE 4.30 – Chemin de propagation de l'erreur.

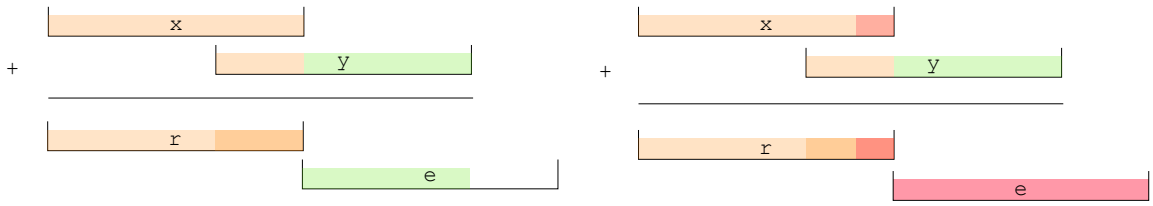


FIGURE 4.31 – Illustration de l'inefficacité des EFTs en présence de bruit sur les entrées. À gauche est présenté la compensation de l'algorithme `TwoSum`. L'erreur perdue dans l'addition entre x et y est contenu dans e. À droite est présenté la compensation de `TwoSum` avec du bruit sur les entrées symbolisé par la couleur rouge. Ces bits rouges étant différents d'une exécution à l'autre, le terme d'erreur e ne contient aucune information lorsque l'on regarde la moyenne des résultats. Le terme d'erreur e peut donc être considéré comme du bruit en moyenne.

4.11 Conclusion

Veritracer est un outil de débeuggage numérique reposant sur l'exploitation de traces numériques permettant de visualiser la qualité numérique des calculs au cours du temps. Il enrichit ces informations par des éléments de contextes qu'il puise dans une analyse des différents flux d'informations liées a l'exécution du programme. Enfin, il fournit un visualisation permettant d'identifier rapidement la qualité des résultats au cours du temps.

Ce chapitre a également montré que l'analyse de codes de calculs industriels fait apparaître des problèmes d'échelle et que la taille des espace de recherche rend les méthodes classiques impraticables. Pour pallier ces problèmes, nous avons développés des méthodes permettant d'identifier rapidement un ensemble de fonctions responsables d'erreurs numériques par raffinements itératifs pour réduire progressivement l'espace de recherche. Nous avons également fourni une hiérarchisation des fonctions à analyser qui est pertinente d'un point de vue développeur pour qui toutes les fonctions ne peuvent être étudiées. D'autre part, couplé avec ces méthodes de réductions de dimensionnalité, nous avons montré que Veritracer permettait de traquer et visualiser l'amélioration de la qualité numérique de la fonction `simp_gen`.

Chapitre 5

Optimisation et précision variable

L'optimisation des formats de calcul est un domaine qui connaît un regain d'attention. L'essor du deep learning pousse les constructeurs à concevoir des nouvelles architectures et formats (cf. 2.3). La question est de savoir où et comment utiliser ces innovations dans le contexte du HPC puisque les méthodes numériques utilisées par ces deux champs de recherche sont similaires. On y retrouve en effet les mêmes méthodes d'algèbre linéaire comme les méthodes itératives de type *Descente de Gradient* [20, 17] pour les problèmes d'optimisation.

Cependant, cet intérêt pour la précision mixte n'est pas nouveau puisque dès les années 60, Wilkinson discutait des avantages de la précision mixte pour améliorer la qualité numérique de la résolution de systèmes linéaires avec sa méthode du *raffinement itératif* [183]. Cette méthode consiste à augmenter la précision des calculs au fur et à mesure des itérations pour atteindre une meilleur exactitude. De plus, certains algorithme sont dit auto-correcteurs, c'est-à-dire qu'ils possèdent la propriété d'être résilients à une certaine quantité d'erreurs numériques durant leur exécution. L'exemple bien connu étant celui de la méthode de Newton-Raphson qui sera présentée en section 5.2. Cette méthode reste en effet convergente sous certaines conditions [105] malgré l'introduction d'erreurs. Les méthodes auto-correctrices sont très intéressantes pour les calculs massivement parallèles dans le HPC puisque la probabilité de bit-flip durant l'exécution d'un programme est proportionnelle au nombre de coeurs utilisés [21].

Si ces méthodes sont restées inexploitées pendant plusieurs années, c'est que l'accroissement de la vitesse de calcul a permis aux développeurs d'application d'utiliser du format binary64.

En effet, l'augmentation de la fréquence, de la taille des mémoires caches et des unités vectorielles a permis d'améliorer la performance des unités de calcul type CPU, conduisant, modulo l'adaptation à l'arrivée de chaque nouvelle génération de processeurs, à une amélioration de la performance des codes. Les architectures étaient plus performantes et donc les codes devenaient plus performants. Or on constate aujourd'hui un plafonnement de la performance mono-coeur dû aux limitations technologiques comme la finesse de gravure par exemple ou la consommation énergétique [61]. Pour conserver une performance croissante, les architectures évoluent naturellement vers du multi-coeur. Le principal vecteur de ralentissement avec les architectures parallèles vient du traitement des données, c'est-à-dire de leur accès en mémoire (cache, RAM, HDD, SSD) mais surtout de leur transfert à travers les interconnections. Or la vitesse d'échange dans les réseaux ainsi que le débit de transfert des données restent bien inférieurs aux débits de calcul fournis par les architectures modernes. Réduire le volume de communications entre les processeurs est donc un facteur crucial

de performance pour les applications aujourd'hui. C'est pourquoi la précision mixte connaît un vif regain d'attention, car elle permet d'accroître la densité de calcul tout en réduisant le volume des communications échangées.

Dans ce chapitre nous commencerons par présenter la méthode de réduction de format que nous avons développé durant la thèse. La contribution majeure de cette méthode présentée dans la section 5.1 par rapport aux approches existantes est double : d'une part notre méthode optimise la précision des calculs de manière temporelle, c'est-à-dire qu'elle cherche un format *sur mesure* pour chaque itération d'un schéma itératif, là où les approches standards se contentent de tester le même format sur toute la durée de l'exécution. D'autre part, les formats explorés par notre méthode sont plus nombreux que dans les approches de précisions mixtes classiques (cf. section 1.3) qui restent pour la plupart cantonnées aux binary32 et binary64. Le backend `VPREC` spécialement développé durant cette thèse permet de simuler n'importe quelle format de taille variable qui tient sur un binary64. Ce backend sera présenté en section 5.1.2.

Nous montrerons que notre méthode est bien adaptée aux schémas itératifs auto-résilients comme la méthode de Newton-Raphson que nous étudierons en détails dans la section 5.2. Puis nous terminerons ce chapitre avec l'application de notre méthode d'exploration sur YALES2, un code HPC de simulation de mécanique des fluides. Nous montrerons que notre méthode permet de réduire de près de moitié la taille des formats de calcul utilisés tout en préservant la convergence d'origine. Enfin nous montrerons les performances apportées par nos solution de formats réduits avec une version en précision mixte faute de matériel disposant d'unités de calcul flottant en précision variable.

5.1 Méthode de réduction de format

Les méthodes présentées en section 1.3 ont pour beaucoup exploré la dimension spatiale (cf. 3.3.1) mais peu l'aspect temporel (cf. 3.3.1) de l'optimisation. C'est pourtant tout un pan d'optimisation qui reste inexploré. En effet, une variable ne nécessite pas la même précision au cours du temps comme le montre l'exemple suivant :

k	x_{k+1}	s_k^{10}	s_k^2
0	0.0690266447076745	0.11	0.37
1	0.1230846130203958	0.21	0.70
2	0.1985746566605835	0.43	1.43
3	0.2732703639721015	0.84	2.79
4	0.3119369815109966	1.79	5.95
5	0.3181822938100336	3.40	11.3
6	0.31830983503924 71	6.79	22.6
7	0.3183098861837825	13.6	45.2
8	0.3183098861837907	15.6	51.8
9	0.3183098861837907	15.6	51.8

Cette table présente l'exécution de la méthode de Newton-Raphson pour calculer l'inverse de π en binary64. Pour chaque itération k, nous avons calculé le nombre de chiffres significatifs s_k^β en commun à l'aide de la formule 2.4.4, où β désigne la base. La colonne x_{k+1} indique la valeur de la variable ligne.7 sur le code 5.1 avec en rouge les chiffres non significatifs.

La couleur rouge souligne qu'une grande partie des chiffres de x_k sont faux dans les premières itérations et de manière duale que peu de chiffres sont significatifs. Pourtant ce code converge vers

```
1  double newton(double x0) {
2   double x_k;
3   double x_kp1 = x0;
4   double b = PI;
5   do {
6    x_k = x_kp1;
7    x_kp1 = x_k*(2-b*x_k);
8   } while (fabs((x_kp1-x_k)/x_k) >= 1e-15);
9   return x_kp1;
10 }
```

FIGURE 5.1 – Code C de la méthode de Newton-Raphson pour calculer l'inverse de π.

la solution attendue. On en déduit que la valeur des chiffres rouges n'a pas d'importance sur la convergence de la méthode et qu'ils pourraient être a priori remplacés par d'autres chiffres sans qu'il y ait de conséquences sur le résultat [1].

Nous avons donc l'intuition qu'un format binary64 n'est pas nécessaire pour arriver à un résultat juste. Il est en effet légitime de se demander qu'il faille 53 bits de précision pour calculer $\simeq 2$ bits justes à l'itération 3. Nous pensons donc qu'il est possible de trouver un format de calcul plus petit que binary64 pour chaque calcul de x_k tout en conservant la même qualité sur la solution obtenue. De plus le backend `VPREC` nous permet de simuler des formats à un grain fin, ouvrant la voie à d'autres formats que ceux proposés dans la norme IEEE-754.

Fort de ce constat, nous avons construit notre méthode d'optimisation pour qu'elle minimise la précision de calcul de chaque invocation à un grain fin. Nous avons également souhaité conserver les propriétés du calcul d'origine à savoir : trouver une solution proche de la solution IEEE-754 (critère de validité numérique, section 3.4.1) en un nombre d'itérations inférieur ou égal à l'exécution d'origine (critère de validité temporelle, cf. 3.4.2).

5.1.1 Principe de la méthode de réduction de format

La méthode d'optimisation de format de calculs proposée dans cette thèse s'appuie sur l'exploration temporelle des formats de calculs.

Pour la dimension temporelle, notre but est de trouver un format de calcul qui soit adapté à chaque itération du schéma. Comme nous l'avons vu avec l'exemple de Newton-Raphson, le nombre de chiffres significatifs d'une même variable évolue au cours du temps. Nous souhaitons ainsi fournir un format de calcul qui soit sur mesure pour chaque invocation. L'idée est d'optimiser plus finement que les Méthodes d'Optimisation Spatiale (MOS) présentées en section 1.3 en exploitant cette dimension temporelle. Les MOS sont contraints par le fait qu'elles optimisent le format d'une variable pour toute l'exécution du programme. Autrement dit, les MOS travaillent à gros grain sur l'ensemble des invocations en même temps tandis que nous travaillons à grain fin en analysant invocation par invocation ce qui permet d'épouser au mieux les besoins en précision des calculs.

1. Ce propos est à modérer notamment par la notion de bit contribuant évoquée en section 2.6.3. En effet, les bits contribuants illustrent le fait que les chiffres non-significatifs contiennent une part d'information sur la significance du résultat, malgré le qualificatif de non-significatifs.

Pour l'optimisation de format, notre but est de proposer un format sur mesure en donnant à chaque itération le nombre de bits suffisants pour garantir la convergence. Ce travail à grain fin (au niveau du bit) est en effet rendu possible par le backend `VPREC` qui permet de simuler n'importe quel format similaire aux formats IEEE-754 tenant sur un binary64 (64 bits), que nous allons présenter dans la section 5.1.2 qui suit. Contrairement aux approches dites de précision mixte (cf. 1.3) qui explorent uniquement les formats IEEE-754, notre méthode permet d'offrir des solutions plus fines qui répondent précisément aux besoins de chaque itération du schéma numérique analysé.

Nous souhaitons donc offrir des solutions de meilleur qualité que les approches MOS et précision mixte traditionnelles en explorant conjointement ces deux dimensions (temporelle et de format à grain fin). Nous souhaitons également que la solution trouvée par la méthode ne dégrade pas la qualité des résultats (contraintes numériques) ni la qualité de la convergence (contraintes de temporalité), c'est-à-dire le nombre d'étapes nécessaires pour atteindre la convergence. Ces critères sont fournis par l'utilisateur et peuvent par exemple correspondre aux résultats de l'exécution originale.

5.1.2 `VPREC`

Le backend `VPREC` a été développé durant cette thèse pour pouvoir simuler des formats de taille variable à une granularité fine. Nous avons vu en section 2.3.1 que l'optimisation de la précision de calcul est un moyen efficace pour gagner en performance. Il est néanmoins indispensable pour un développeur de savoir comment son code réagit aux changements de format de calcul avant de le modifier. On souhaite donc fournir aux développeurs d'application un moyen rapide de tester un type flottant sans se limiter aux formats IEEE-754 puisque nous avons vu que de nombreux formats alternatifs émergeaient ces dernières années (cf. section 2.3.1).

Le backend `VPREC` permet donc de répondre à ses besoins en permettant de simuler des formats de taille variable tenant sur du binary64. Le format simulé étant défini par la taille de son exposant (d) et la taille de sa mantisse (p). De plus, l'utilisation de `VPREC` à travers l'interface de Verificarlo permet de changer dynamiquement la taille de d et p durant l'exécution sans recompilations de la part du développeur.

Pour simuler des format flottants de taille variable sans avoir le surcoût lié à la librairie `MPFR`, le backend `VPREC` simule les formats qui peuvent être contenus dans un binary64 similairement à la norme IEEE-754.

Le nouveau format est défini par les deux paramètres (d, p). Le premier $d \in [0, 11]$ détermine la taille de l'exposant avec pour bornes $e_{max} = 2^{r-1} - 1$ et $e_{min} = 1 - e_{max}$. Le deuxième $p \in [1, 52]$ la taille de la mantisse. Par exemple, pour simuler du binary16 sur un binary32, on choisira $d = 5$ et $p = 10$ comme l'illustre la figure 5.2.

`VPREC` gère également les NaNs, $\pm Inf$ et les dénormalisés ainsi que l'arrondi au plus près.

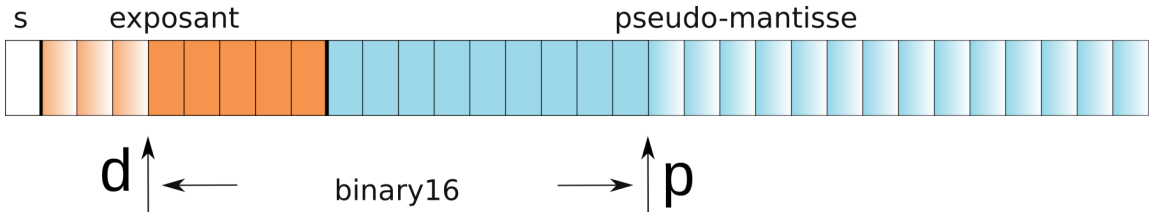


FIGURE 5.2 – Format VPREC simulant du binary16 dans un binary32. Les bits opaques représentent le nouvel exposant (d) en orange et la nouvelle mantisse (p) en bleu. Le signe reste inchangé.

VPREC dispose de quatre modes pour appliquer l'arrondi sur les opérandes de chaque opération flottante : soit $\diamond \in (+, -, \times, /)$ l'opération arithmétique IEEE-754 effectuée en précision d'origine (binary32, binary64) et $\circ_{d,p}$ l'arrondi au plus près en précision p et en dynamique d.

- **IEEE** : $IEEE(x \diamond y) = x \diamond y$,
  Les calculs sont effectués dans le format d'origine.
- **Inbound Precision** : $IB(x \diamond y) = \circ_{d,p}(x) \diamond \circ_{d,p}(y)$,
  il permet de borner la précision des opérandes.
- **Outbound Precision** : $OB(x \diamond y) = \circ_{d,p}(x \diamond y)$,
  il permet de borner la précision du résultat.
- **Full Precision Bounding** : $FULL(x \diamond y) = \circ_{d,p}(\circ_{d,p}(x) \diamond \circ_{d,p}(y))$,
  il permet de borner la précision en entrée et en sortie.

Le résultat de l'opération est stocké dans le format d'origine. Ceci permet d'utiliser des librairies externes sans intervention de l'utilisateur et de mécanismes complexes comme la shadow memory.

La figure 5.3 présente un exemple de comment simuler du binary16 à partir d'un format binary32 avec VPREC. On voit qu'il suffit d'utiliser les paramètres $d = 5$ et $p = 10$ pour simuler un binary16 sans avoir nécessairement à disposition l'unité flottante correspondante en matériel. De plus, comme les paramètres d et p sont modifiables durant l'exécution du programme en utilisant les variables d'environnement spécifiques, il est facile pour l'utilisateur de tester d'autre formats. Par exemple, en utilisant $d = 8$ et $p = 7$ l'utilisateur peut tester l'impact du bfloat16 sur son code sans recompiler celui-ci.

Pour comprendre le fonctionnement de VPREC, nous allons simuler le comportement de l'opération $(2.903225 + 2.903225) * 16384)$ en binary32 à l'origine avec du binary16. Cet exemple est détaillé sur la figure 5.3. On a présenté à gauche l'exécution originale et à droite l'exécution en simulant du binary16. On voit que la première opération effectue l'arrondi sans problèmes contrairement à la deuxième dont la dynamique est trop grande pour être représentée dans du binary16. Le résultat provoque donc un overflow et renvoi un $+Inf$.

A l'heure actuelle, nous ne pouvons pas fournir les détails d'implementation sur le backend VPREC car celui-ci est sous inspection d'Intel.

Performance

Les performances du backend VPREC sont bonnes avec un faible surcoût dû à l'utilisation des types natifs IEEE-754. Sur le test utilisé en figure 3.10 compilé avec les mêmes paramètres `-DSAMPLES=100000 -DITER=5000`, le code original IEEE-754 affiche un temps de 1,46 secondes contre 12,55 secondes en moyenne pour le backend VPREC soit un ralentissement de seulement 8,59.

	$1.01110011100111001110000 \times 2^1$
$+$	$1.01110011100111001110000 \times 2^1$
$\diamond_{8,23}$	$1.01000010000100001000000 \times 2^{\mathbf{2}}$
	$1.01000010000100001000000 \times 2^2$
	$1.01000010000100001000000 \times 2^2$
$*$	$1.00000000000000000000000 \times 2^{14}$
$\diamond_{8,23}$	$1.01000010001000010000000 \times 2^{\mathbf{16}}$
	$1.01000010001000010000000 \times 2^{16}$

	$1.01110011100111001110000 \times 2^1$
$+$	$1.01110011100111001110000 \times 2^1$
$\diamond_{5,10}$	$1.01000010000100001000000 \times 2^{\mathbf{2}}$
	1.0100001000×2^2
	$1.01000010000000000000000 \times 2^2$
$*$	$1.00000000000000000000000 \times 2^{14}$
$\diamond_{5,10}$	$1.01000010000000000000000 \times 2^{\mathbf{16}}$
	$+Inf$

FIGURE 5.3 – Exécution de l'opération $(2.903225 + 2.903225) * 16384)$ en binary32 à gauche et avec le backend `VPREC` à droite simulant du binary16 en prenant comme paramètres `p=10` et `d=5`. Contrairement à l'exécution en binary32, le calcul avec `VPREC` conduit à un overflow puisque le nombre maximal représentable avec un `d=5` est 2^{16}.

5.1.3 Algorithmes d'explorations

Nous avons décliné l'algorithme de recherche en trois versions :

- En avant : On cherche à diminuer la précision de chaque invocation en commençant par la première et en terminant par la dernière.
- En arrière : On cherche à diminuer la précision de chaque invocation en commençant par la dernière et en terminant par la première.
- Par morceaux : On cherche à diminuer la précision pour un ensemble d'invocations à la fois, en commençant par réduire l'ensemble des invocations dans un premier temps. On divise ensuite l'ensemble en deux morceaux puis on cherche récursivement sur les deux nouveaux ensembles crées et ce ainsi de suite.

Approches directionnelles

Les approches directionnelles explorent l'ensemble des invocations de manière incrémentale. L'idée, illustrée sur la figure 5.4 est de minimiser la précision de calcul de chaque itération soit, en commençant par la première invocation et en incrémentant jusqu'à la dernière, soit inversement, en commençant par la dernière puis en avançant de manière décrémentale jusqu'à la dernière. Les invocations qui n'ont pas encore été explorées sont exécutées dans le format d'origine. On peut également chercher à vouloir minimiser la précision de plusieurs invocations successives à la fois. On note ce nombre d qui par défaut vaut 1.

Méthode *en avant*. Le principe de la méthode *en avant* est de répartir le budget d'erreur sur les premières itérations. En effet, l'hypothèse sous-jacente est que les schémas itératifs requièrent en général peu de précision de calcul sur les premières itérations puis plus après. Pour mener à bien la convergence, on augmente la précision au fur et à mesure des itérations. La deuxième hypothèse est que la réduction de précision sur les premières itérations agisse comme un filtre passe-bas, c'est-à-dire qu'elle détruit les erreurs numériques de petites magnitudes qui ont pu s'accumuler et qu'elle améliore ainsi la convergence. Le potentiel problème avec cette méthode est qu'elle répartisse de façon inégalée le budget d'erreur en le concentrant sur les premières itérations.

Méthode en *en arrière*. Le principe de la méthode *en arrière* est de partir de la dernière invocation et de remonter le fil des invocations jusqu'à la première d'entre elles. Toujours suivant

l'hypothèse que la précision nécessaire croît au cours du temps, la méthode en arrière permet en principe de mieux répartir le budget d'erreur que la méthode *en avant*. L'idée étant qu'en concentrant la minimisation de la précision sur la dernière itération, puis l'avant dernière et ainsi de suite, la précision requise par les premières invocations soit plus élevée puisque les dernières invocations ont consommé une grande partie du budget d'erreur.

Interaction inter-invocations. La recherche invocation par invocation peut négliger un aspect important que sont les interactions entre invocations successives. En effet, les schémas itératifs sont construits sur des récursions du type :

$$x_{k+1} = f(x_k)$$

voire avec des ordres de récurrence plus élevés. Il est donc intéressant de prendre en compte ces dépendances lors de la minimisation de la précision comme on le montrera dans la section 5.2. On introduit donc le paramètre D (par défaut à 1) qui définit le nombre minimal d'invocations à explorer en même temps.

Approche par morceaux

L'approche par morceaux est une exploration de type Divide & Conquer qui consiste à optimiser la précision de plusieurs invocations à la fois. L'ensemble des invocations optimisées constitue un morceau. Une fois que la précision optimale est trouvée pour un morceau, celui ci est divisé en plusieurs morceaux pour lesquels on cherche à optimiser récursivement chaque sous-partie jusqu'à atteindre des morceaux de taille minimale. L'approche par morceaux est née d'une synthèse des quatre principes suivants :

1. Les solutions qui évoluent lentement dans le temps sont préférables aux solutions évoluant de manière chaotique. Le postulat étant que les architectures qui disposent d'un mécanisme de changement dynamique de précision ont un surcoût pour modifier les circuits de calcul. Il est donc préférable de minimiser les changements de précision.
2. La réduction de précision doit se répartir de manière homogène sur l'ensemble des invocations. Comme nous le montrerons en section 5.2, l'idée est qu'il est préférable que chaque invocation ait une précision réduite plutôt qu'il y ait des contrastes élevés entre régions.
3. Les premières itérations des schémas itératifs nécessitent une précision de calcul plus réduite que les dernières itérations. Une autre façon d'aborder ce principe est de considérer que les premières itérations peuvent absorber plus d'erreurs que les dernières itérations. Cette hypothèse se prête bien aux algorithmes résilients comme Newton-Raphson mais moins à des schémas non auto-correcteurs comme celui de Goldschmidt (cf.section 5.2.4).
4. Les interactions (cf. section 5.1.3) entre invocations peuvent être sur des distances de plusieurs itérations et avec des corrélations différentes. Cette multi-résolution n'est pas prise en compte dans les méthodes unidirectionnelles puisque la distance d'interaction (D) est fixe durant la recherche. En prenant des tailles de morceaux de plus en plus fins, on peut capturer différentes résolutions d'interactions entre les invocations.

Le principe de l'approche par morceaux est présenté sur la figure 5.5.

Complexité des méthodes

On souhaite avoir une estimation au pire cas du temps mis par les méthodes de recherche pour trouver une solution. Notre mesure de complexité s'intéresse donc au nombre d'exécutions

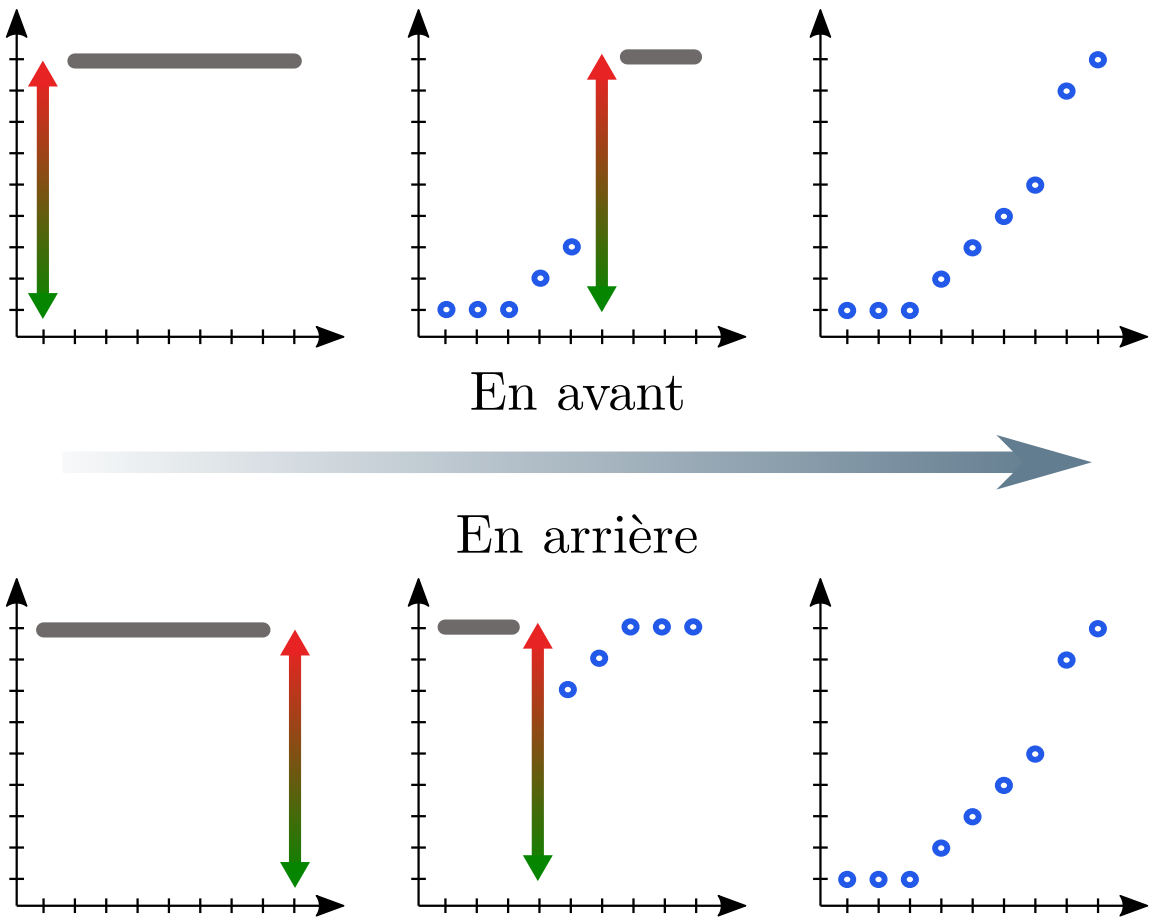


FIGURE 5.4 – Illustration des méthodes *en avant* et *en arrière*. La flèche en dégradé représente le sens d'exécution de l'exploration. Les ordonnées et les abscisses de chaque repère représentent respectivement la précision et l'invocation. La flèche verticale en dégradé vert/rouge représente la recherche de précision pour l'invocation correspondante. La barre grise à la précision maximale signifie que les invocations qui n'ont pas encore été explorées sont exécutées à la précision d'origine. Chaque point bleu représente la solution de la recherche dichotomique de la précision pour l'invocation correspondante.

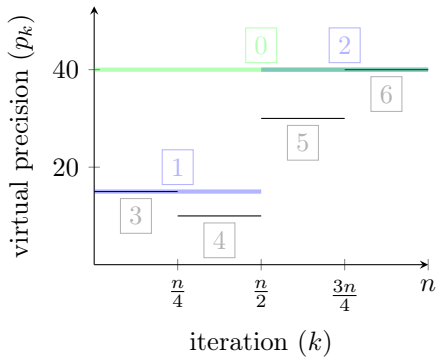


FIGURE 5.5 – L'approche par morceaux consiste à diviser récursivement par dichotomie l'ensemble des invocations à explorer. Chaque ensemble d'invocations représente un intervalle d'itération sur lequel diminuer la précision. Ici sont présentées les trois premières étapes en vert, bleu et noir. Les intervalles sont découpés jusqu'à ce qu'ils soient de taille D.

du programme d'origine nécessaire pour mener à bien la recherche. La complexité des méthodes dépend de trois paramètre :

- N : le nombre maximal d'invocations autorisé par la contrainte de validité temporelle.
- D : le nombre minimal d'invocations pour chaque recherche de précision.
- T : le nombre de précision virtuelle que l'on souhaite explorer. Par défaut on le prend égal à la précision d'origine plus un.

On pose n comme le ratio $\lceil \frac{N}{D} \rceil$.

Cela borne donc la complexité des méthodes directionnelles à :

$$\mathcal{O}\left(n \log_2(T)\right).$$

Celle de la méthode par morceaux à :

$$\mathcal{O}\left(2n \log_2(n) \log_2(T)\right).$$

5.2 Exemple sur la méthode de Newton-Raphson

La méthode de Newton-Raphson est une méthode itérative pour trouver une racine d'une fonction. Elle doit son nom à Isaac Newton et Joseph Raphson. Pour plus d'histoire sur la méthode, voir [184]. Pour une fonction f réelle différentiable, on souhaite trouver une racine de f notée $x^\star$ telle que :

$$f(x^\star) = 0.$$

5.2.1 Équation

L'idée de la méthode est de suivre le gradient de la fonction pour se diriger vers l'endroit où elle s'annule en utilisant l'équation de la tangente ou développement limité d'ordre 1 (cf. formule **??**).

Ainsi pour le cas réel, si $f : \mathbb{R} \to \mathbb{R}$, $f \in \mathcal{C}^1(\mathbb{R})$ alors l'équation de la tangente au point x_0 s'écrit :

$$y = f(x_0) + f'(x_0)(x - x_0).$$

On cherche le point qui coupe la tangente en 0 :

$$\begin{aligned} 0 &= f(x_0) + f'(x_0)(x - x_0) \\ -f'(x_0)(x - x_0) &= f(x_0) \\ (x - x_0) &= -\frac{f(x_0)}{f'(x_0)} \\ x &= x_0 - \frac{f(x_0)}{f'(x_0)}. \end{aligned}$$

On obtient donc le schéma itératif $\mathcal{NR}$ suivant avec x_0 comme solution de départ :

$$\mathcal{NR}(x_k) = x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Comme toute méthode itérative, un critère d'arrêt est nécessaire pour terminer la résolution. Celui ci est généralement pris comme l'erreur relative entre deux itérés successifs x_k et x_{k+1} avec un seuil prit aux alentours de la précision machine. Cela signifie que la distance entre deux solutions successives n'est plus représentable dans le format de calcul. On dit qu'un schéma itératif a convergé lorsqu'il a atteint le critère d'arrêt aussi appelé critère de convergence. Le code de la figure 5.1 présente le code associé dans le cas du calcul de l'inverse de π.

5.2.2 Convergence de la méthode

Une propriété intéressante des méthodes itératives est la vitesse de convergence. Celle-ci mesure la rapidité avec laquelle la méthode va donner la solution et donne une indication sur le nombre d'étapes nécessaires pour atteindre la convergence. Nous allons montrer le résultat classique [110, 89] de la convergence quadratique de la méthode de Newton-Raphson, c'est-à-dire que l'erreur $\epsilon = |x_k - x|$ est divisée par deux à chaque étape ou bien que le nombre de chiffre en commun entre x_k et x est doublé à chaque itération. La vitesse de convergence est donnée par la définition 5.2.1.

Définition 5.2.1 (Vitesse de convergence)**.** Une suite $(u_k)_{k\in\mathbb{N}}$ est dite de convergence p si limite la suivante existe ($\lambda < \infty$) :

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^\star|}{|x_k - x^\star|^p} = \lambda.$$

On suppose que f a un racine simple. On part du développement limité à l'ordre 1 de $f \in \mathcal{C}(\mathbb{R})^2$

au voisinage de $x^\star$:

$$\begin{aligned} f(x) &= f(x^\star) + f'(x)(x - x^\star) + \frac{f''}{2}(\xi)(x - x^\star)^2 \\ f(x) &= f'(x)(x - x^\star) + \frac{f''(\xi)}{2}(x - x^\star)^2 \\ \frac{f(x)}{f'(x)} &= (x - x^\star) + \frac{f''(\xi)}{2f'(x)}(x - x^\star)^2 \\ \frac{f(x)}{f'(x)} - (x - x^\star) &= \frac{f''(\xi)}{2f'(x)}(x - x^\star)^2 \\ x - \frac{f(x)}{f'(x)} - x^\star &= -\frac{f''(\xi)}{2f'(x)}(x - x^\star)^2. \end{aligned}$$

On reconnaît ici le schéma $\mathcal{NR}$

$$\mathcal{NR}(x) = \frac{f''(\xi}{2f'(x)})(x - x^\star)^2.$$

donc

$$x - \frac{f(x)}{f'(x)} - x^\star = -\frac{f''(\xi)}{2f'(x)}(x - x^\star)^2.$$

Or $f \in \mathcal{C}(\mathbb{R})^\in$ donc $f'(x)$ et $f''(x)$ sont continues sur $\mathbb{R}$ elles admettent donc des extrema locaux au voisinage de ξ donc $\sup_{x \in \mathcal{V}(\xi)} \frac{f''(\xi)}{2f'(x)} = M$ existe ce qui implique que :

$$\begin{aligned} |\mathcal{NR}(x) - x^\star| &<= |M||x - x^\star|^2 \\ \frac{|\mathcal{NR}(x) - x^\star|}{|x - x^\star|^2} &<= |M|. \end{aligned}$$

D'après la définition 5.2.1 on en déduit que la méthode de Newton-Raphson a une convergence quadratique.

5.2.3 Calcul de la racine carrée de 2

On souhaite calculer la racine carrée de 2 à partir de la méthode de Newton-Raphson. Cela revient à trouver la solution de la fonction f suivante :

$$\begin{aligned} x = \sqrt{2} &\implies x^2 = 2 \\ &\iff x^2 - 2 = 0 \qquad\qquad \iff f(x) = 0 \end{aligned}$$

```c
double erreur_relative(double x, double y) {
  return fabs((x-y)/y);
}

double newton(double x0) {
  double x_k;
  double x_kp1 = x0;
  double b = 0.5;
  do {
   x_k = x_kp1;
   x_kp1 = 0.5*x_k-1/x_k;
  } while (erreur_relative(x_kp1, x_k) >= 1e-15);
  return x_kp1;
}
```

FIGURE 5.6 – Code C du calcul de $\sqrt{2}$ par la méthode de Newton-Raphson.

avec $f'(x) = 2x$. On obtient donc le schéma $\mathcal{N}(x)$ suivant :

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$
$$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k}$$
$$x_{k+1} = \frac{x_k}{2} - \frac{1}{x_k}.$$

Lorsque l'on exécute le programme associé présenté en figure 5.6 en binary64, on obtient les valeurs des x_k suivants sur la table 5.1. Nous voyons que la méthode suit bien une convergence quadratique puisque le nombre de chiffres significatifs est au minimum doublé à chaque itération.

Nous avons ensuite appliqué notre méthode d'exploration en utilisant les trois méthodes présentées en section 5.1.1 avec les critères suivants : On pose $x_\infty = \max_k\{x_k\}$.

- $c_t : x_\infty \leqslant n_{IEEE}.\epsilon_t$
- $c_n : \left|\frac{x_\infty - x^\star}{x^\star}\right| \leqslant \epsilon_n$

avec n_{IEEE} le nombre d'itérations IEEE-754, $x^\star = \sqrt{2}$, $\epsilon_t = 1$ et $\epsilon_n = 10^{-15}$ La fonction `erreur_relative` n'est pas bruitée avec Verificarlo pour garder le critère d'arrêt original. En effet, les critères d'arrêts sont des fonctions sensibles d'un point de vue numérique car il s'agit souvent de calculs de distance entre deux éléments proches (voisins d'un maillage par exemple cf. 2.5.3) ce qui est sujet aux erreurs d'annulations.

La figure 5.7 compare les trois méthodes d'exploration. Pour chacune d'entre elles, on présente à gauche la solution trouvée par l'algorithme, c'est-à-dire les précisions virtuelles trouvées pour chaque itération k. La ligne noire située à l'ordonnée 24 sépare la précision du binary32 de celle du

k	x_{k+1}	s_k^2	s_k^{10}
0	0.5000000000000000	0.629	0.189
1	2.2500000000000000	0.758	0.228
1	1.5694444444444444	3.187	0.959
2	1.42189036381514 26	7.525	2.265
3	1.41423428594007 34	16.05	4.834
4	1.41421356252493 21	33.11	9.969
5	1.41421356237309 50	53.11	15.98

TABLE 5.1 – Exécution de la méthode de Newton-Raphson pour calculer $\sqrt{2}$ en binary64. Pour chaque itération k, nous avons le nombre de chiffres significatifs s_k^β en commun calculé à l'aide de la formule 2.4.4, où β désigne la base. La colonne x_{k+1} indique la valeur de la variable `x_xkp1` à la ligne.7 sur la figure 5.6 avec en rouge les chiffres non significatifs. On voit que s double au minimum à chaque itération k car la méthode a convergence quadratique.

binary64. Elle indique le nombre potentiel d'invocations pouvant être exécutées en précision mixte. Nous avons implémenté une version utilisant de la précision mixte pour valider la solution trouvée par notre méthode que nous présenterons dans la section 5.2.3. À droite se trouve le graphique présentant la vitesse de convergence de l'exécution originale au format IEEE-754 (en vert), celle de l'exécution réalisée avec `VPREC` en utilisant les précisions virtuelles données sur le graphique de gauche et la vitesse de convergence théorique attendue de la méthode (en rouge). Cette dernière donne une borne sur le nombre minimal de chiffre significatifs attendus à l'itération k.

On constate que pour chacune des méthodes, les solutions proposées avec le backend `VPREC` suivent le même profil de convergence que la solution IEEE-754 et sont au dessus de la vitesse de convergence attendue ce qui atteste de la qualité de la solution trouvée.

Influence du seuil de convergence

On s'est intéressé à l'influence du seuil de convergence ϵ_n du critère de validité numérique sur la qualité des solutions trouvées. La figure 5.8 montre les résultats sur les trois méthodes en faisant varier $\epsilon_n \in \{10^{-5}, 10^{-10}, 10^{-15}\}$. Pour la méthode par morceaux, on voit que les solutions trouvées par `VPREC` sur la figure 5.8(a) requièrent moins de précision lorsque ϵ_n est plus faible. Les vitesses de convergences sont également plus faibles et même en dessous de la vitesse théorique.

La variation du paramètre ϵ_n n'influe pas sur les solutions trouvées par les méthodes directionnelles. Dans ces modes d'exploration, nous avons utilisé une petite fenêtre d'exploration puisque nous avons choisi d'optimiser invocation par invocation. Or le fait d'optimiser plusieurs invocations simultanément permet de capturer des liens entre les calculs. Dans notre cas, le calcul du résidu dans le critère d'arrêt est pris entre deux x_k successifs. Puisque on optimise ces x_k séparément, la précision trouvée pour chacun d'entre eux est différente ce qui implique un arrondi par `VPREC` différent. Or comme le montre la figure 5.9, avoir le même arrondi pour deux x_k successifs permet d'atteindre plus rapidement le critère de convergence.

Pour rendre compte de ce principe, nous avons exécuté les méthodes de recherches en forçant l'exploration à analyser deux invocations au minimum. Les résultats sont présentés sur la figure 5.10. On voit maintenant que les méthodes *en avant* et *en arrière* trouvent également des solutions ayant des précisions plus faibles. Le choix de la méthode a un impact sur la qualité des solutions puisque le partitionnement des invocations n'est pas le même. En effet, pour :

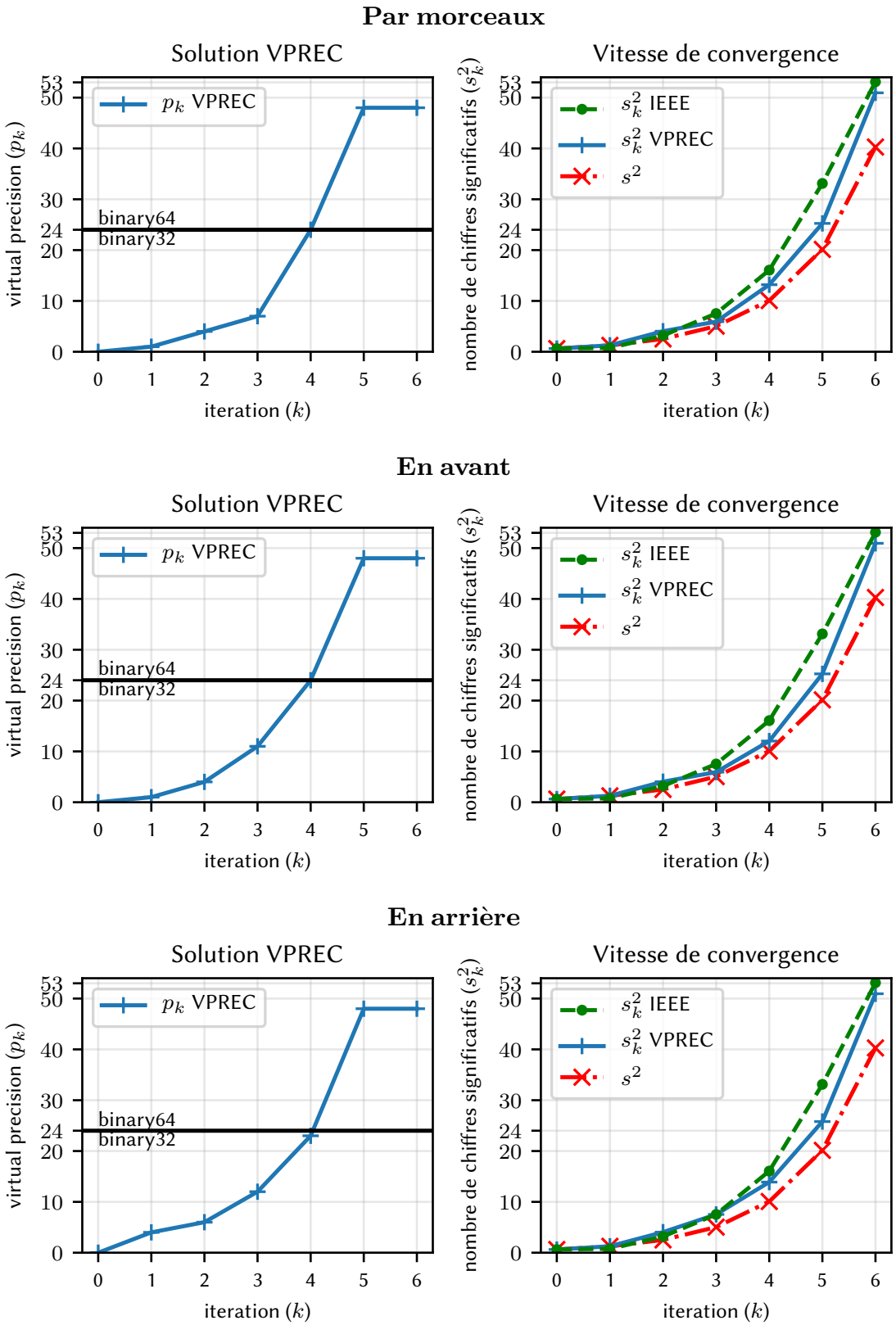
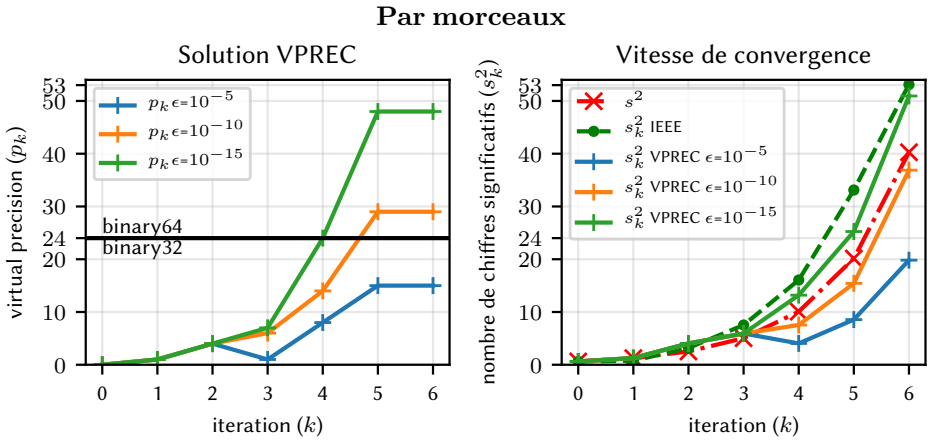
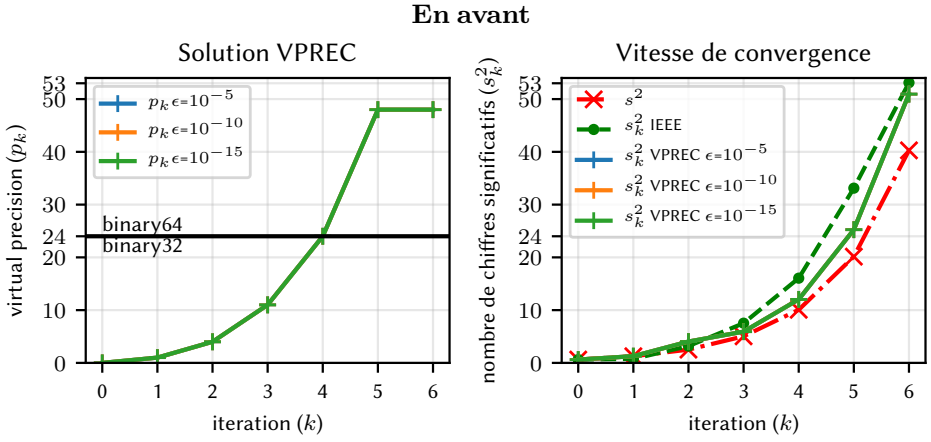


FIGURE 5.7 – Comparaison des explorations pour NR sur $\sqrt{2}$. Est présentée à gauche la solution trouvée par l'algorithme, c'est-à-dire les précisions virtuelles trouvées pour chaque invocation x_k. À droite sont présentées les vitesses de convergence associées. On voit que les trois méthodes d'explorations arrivent à baisser la précision de calcul tout en conservant un profil de convergence similaire à l'exécution en IEEE-754.

(a)



(b)



(c)

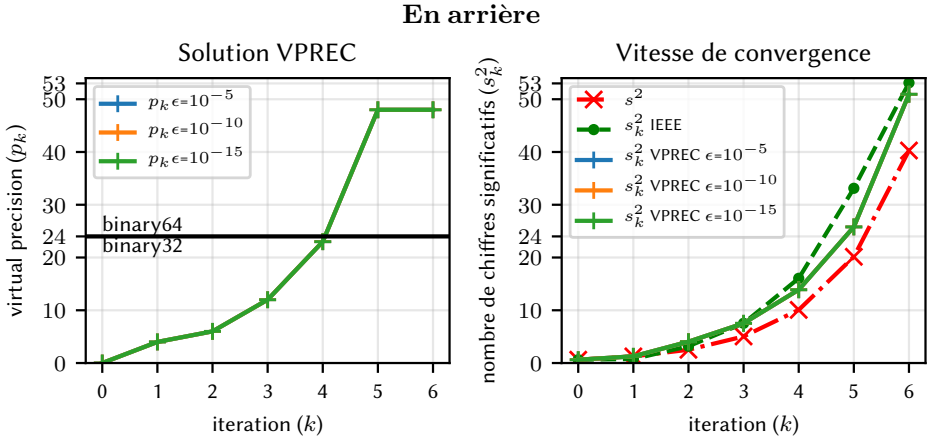


FIGURE 5.8 – Comparaison des résultats des explorations en faisant varier le seuil de tolérance (ϵ). On a gauche la solution trouvée par l'exploration, c'est-à-dire les précisions virtuelles trouvées pour chaque invocation x_k. À droite sont présentées les vitesses de convergence associées. On voit que les trois méthodes d'explorations arrivent à baisser la précision de calcul tout en conservant un profil de convergence similaire à l'exécution en IEEE-754. La variation de ϵ a une influence sur les solutions obtenues avec l'approche par morceaux mais n'en a pas sur les méthodes directionnelles. L'explication est que les méthodes directionnelles ne prennent pas en compte l'interaction entre deux x_k successifs comme le montre la figure 5.9.

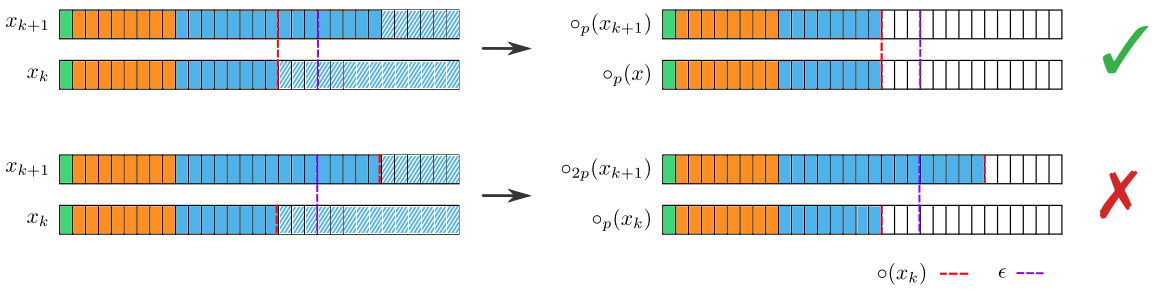


FIGURE 5.9 – Illustration de l'effet de l'arrondi VPREC sur le calcul du résidu. Les cases vertes, oranges et bleues représentent respectivement les bits de signe, d'exposant et de mantisse. Des couleurs opaques signifies que les bits en x_k et x_{k+1} sont différents et des couleurs transparentes que les bits sont égaux. La couleur blanche signifie zéro. Les pointillés rouges représentent l'arrondi au plus près à la précision p effectué par VPREC, et ceux violets le critère d'arrêt. On voit sur la ligne du haut qu'utiliser la même précision pour arrondir les deux invocations permet d'atteindre le critère de convergence. En revanche, l'utilisation du double de la précision d'arrondi utilisée à l'itération précédente ne permet pas d'atteindre la convergence.

- En avant : [0,1]→[2,3]→[4,5]→[6]
- En arrière : [5,6]→[3,4]→[1,2]→[0]
- Par morceaux [2] : [0,1]→[2,3]→[4,5]→[6]

Cela influe notamment sur la précision de la dernière itération puisque l'algorithme d'exploration arrive à trouver une solution utilisant une itération de moins que la version originale (6 contre 7) pour la recherche *en arrière*. Ceci se traduit par une précision $p_6 = 0$.

Enfin dernier point sur l'importance du critère de validité temporelle dans la réduction de l'exploration, qui permet de montrer le caractère auto-correcteur de l'algorithme de Newton-Raphson. La figure 5.11 montre une recherche par morceaux avec une contrainte temporelle très relâchée ($\epsilon_t = 100$). Pour rappel, les itérations supplémentaires autorisées sont exécutées en précision d'origine (ici binary64). La précision de ces itérations est indiquée en pointillé orange sur la figure de gauche, et la configuration trouvée en bleu. On observe que la quasi-totalité des précisions trouvées sont faibles (< 5 bits). Cependant, lorsqu'on observe la vitesse de convergence associée, on s'aperçoit que la méthode converge bien vers la solution de manière quadratique. Celle ci démarre plus tard ($k \simeq 6$), c'est-à-dire quand le calcul repart dans le format d'origine. La solution trouvée est donc plus faible en moyenne si on regarde la précision moyenne par itération (25 contre 53 pour celle d'origine) mais en réalité, il s'avère que nous avons uniquement retardé la convergence de la méthode.

2. On affiche ici les ensembles de taille minimale et non les ensembles intermédiaires.

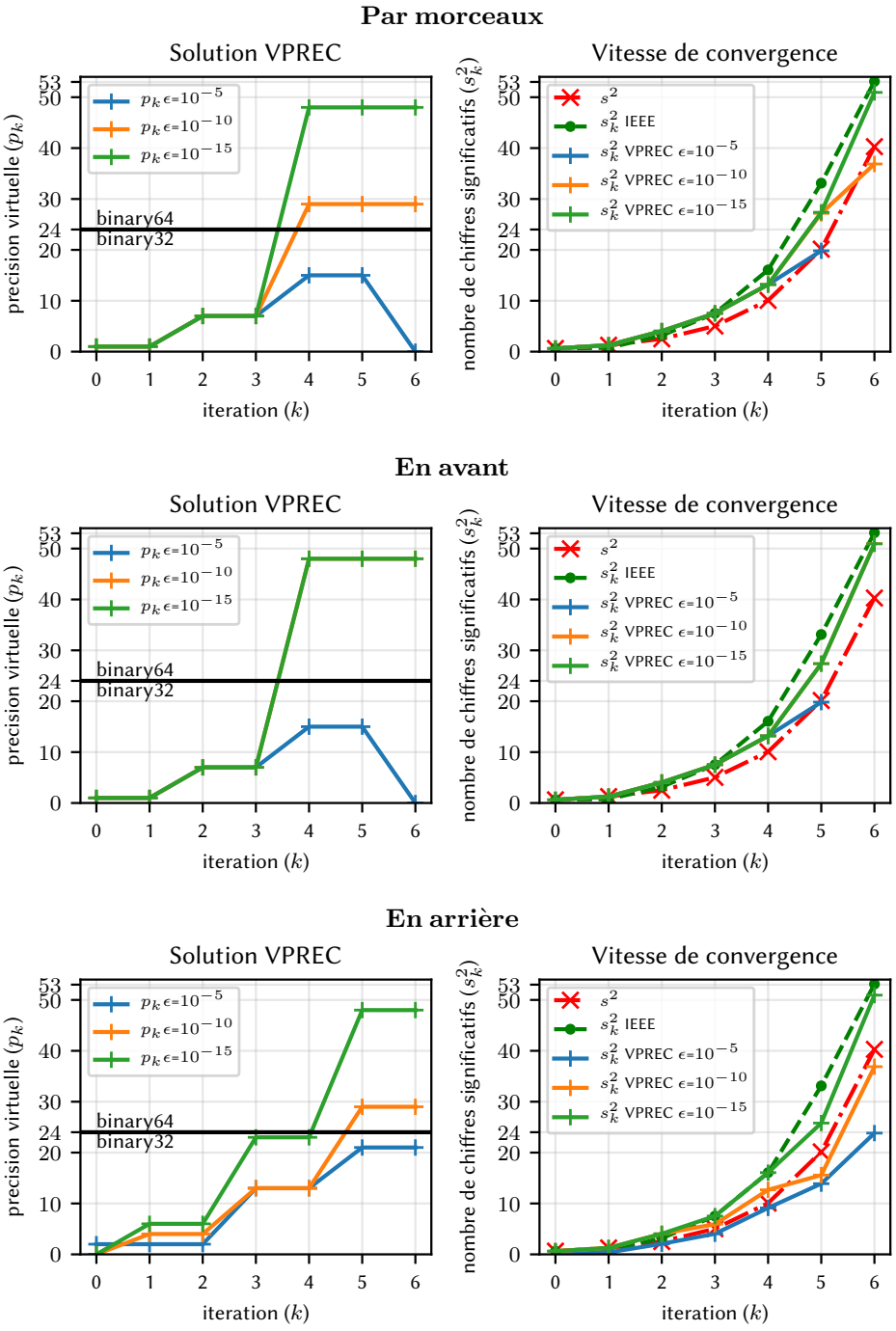


FIGURE 5.10 – Comparaison des résultats des explorations en faisant varier le seuil de tolérance (ϵ) avec une distance inter-itération de 2. On a gauche la solution trouvée par l'exploration, c'est-à-dire les précisions virtuelles trouvées pour chaque invocation x_k. À droite sont présentées les vitesses de convergence associées. On voit que les trois méthodes d'exploration arrivent à baisser la précision de calcul tout en conservant un profil de convergence similaire à l'exécution en IEEE-754. Optimiser la précision de deux invocations à la fois permet de prendre en compte les interactions. Les méthodes directionnelles arrivent ainsi à trouver des solutions plus réduites en faisant varier ϵ contrairement à ce qui a été présenté sur la figure 5.8.

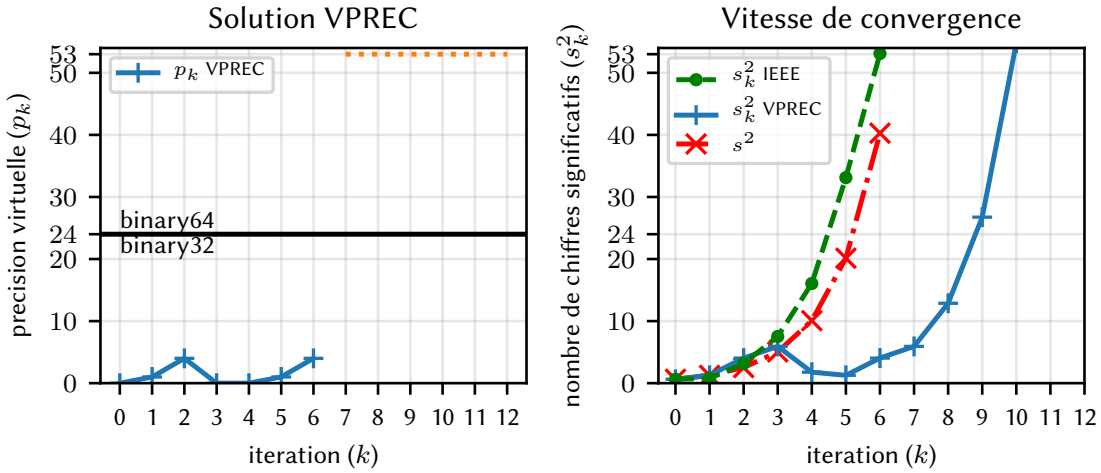


FIGURE 5.11 – Exploration par morceaux sans contrainte de temps pour la méthode de Newton-Raphson sur le calcul de $\sqrt{2}$. Les itérations effectuées après le nombre d'itérations de l'exécution originale ($n_{IEEE} = 6$) utilisent la précision d'origine (binary64). La précision de ces itérations est indiquée en pointillé orange sur la figure de gauche, et la configuration trouvée en bleu. La méthode converge bien vers la solution de manière quadratique mais démarre plus tard quand le calcul utilise le format binary64 d'origine. On a une précision moyenne par itération (25) plus faible mais la convergence de la méthode a uniquement été reportée.

5.2.4 Goldschmidt

Pour montrer que notre méthode d'exploration rencontre certaines difficultés avec les algorithmes non auto-correcteurs, nous l'avons testé sur la méthode de Goldschmidt. Cette méthode qui repose sur la méthode de Newton-Rapshon permet de calculer la racine carrée et son inverse avec le même schéma. Ils existent plusieurs variantes [105, 125] dont nous présentons la suivante : si

$$\begin{aligned} r_k &= \tfrac{1}{2} - x_k h_k \\ x_{k+1} &= x_k + x_k r_k \\ h_{k+1} &= h_k + h_k r_k \end{aligned}$$

alors

$$\begin{aligned} \lim_{k\to\infty} x_k &= \sqrt{S} \\ \lim_{k\to\infty} 2h_k &= \frac{1}{\sqrt{S}} \end{aligned}$$

avec $y_0 \simeq \frac{1}{\sqrt{S}}, x_0 = 2y_0, h_0 = y_0/2$.

Il a été montré [62] que ce schéma n'est pas résilient aux erreurs numériques. En appliquant notre algorithme d'exploration, on voit sur la figure 5.12 que la précision nécessaire reste relativement haute (< 45) et identique pour l'ensemble des invocations et des trois modes d'exploration.

La méthode de Goldschmidt montre qu'il est plus difficile d'optimiser les méthodes non résilientes à l'erreur numérique. Contrairement à Newton-Raphson, toutes les itérations de Goldschmidt requièrent plus ou moins la même précision élevée. Néanmoins `VPREC` montre qu'il est possible de baisser la précision de quelques bits sur l'ensemble des itérations, chose impossible avec une approche en précision mixte.

Validation en précision mixte

Pour montrer que les solutions trouvées par notre algorithme d'exploration sont correctes, nous avons implémenté une version en précision mixte. La gestion des binary16 n'étant pas encore supportée sur les architectures intel utilisées avec clang, nous n'avons pas pu tester nos solutions avec ce type flottant. Nous avons donc utilisé les types binary32 et binary64. L'exécution de la version mixte est présentée sur la figure 5.13. On observe que les deux versions en précision mixte et en précision originale ont le même profil de convergence.

5.3 Robustesse des résultats

Le backend `VPREC` tout comme les opérations suivant la norme IEEE-754 sont déterministes. Or, la solution trouvée par notre algorithme d'exploration est liée à l'environnement de calcul utilisé. De plus, comme le design de l'algorithme d'exploration a été pensé pour minimiser au maximum la précision utilisée, cette nouvelle version du code en précision réduite est peut être plus sensible aux erreurs d'arrondi.

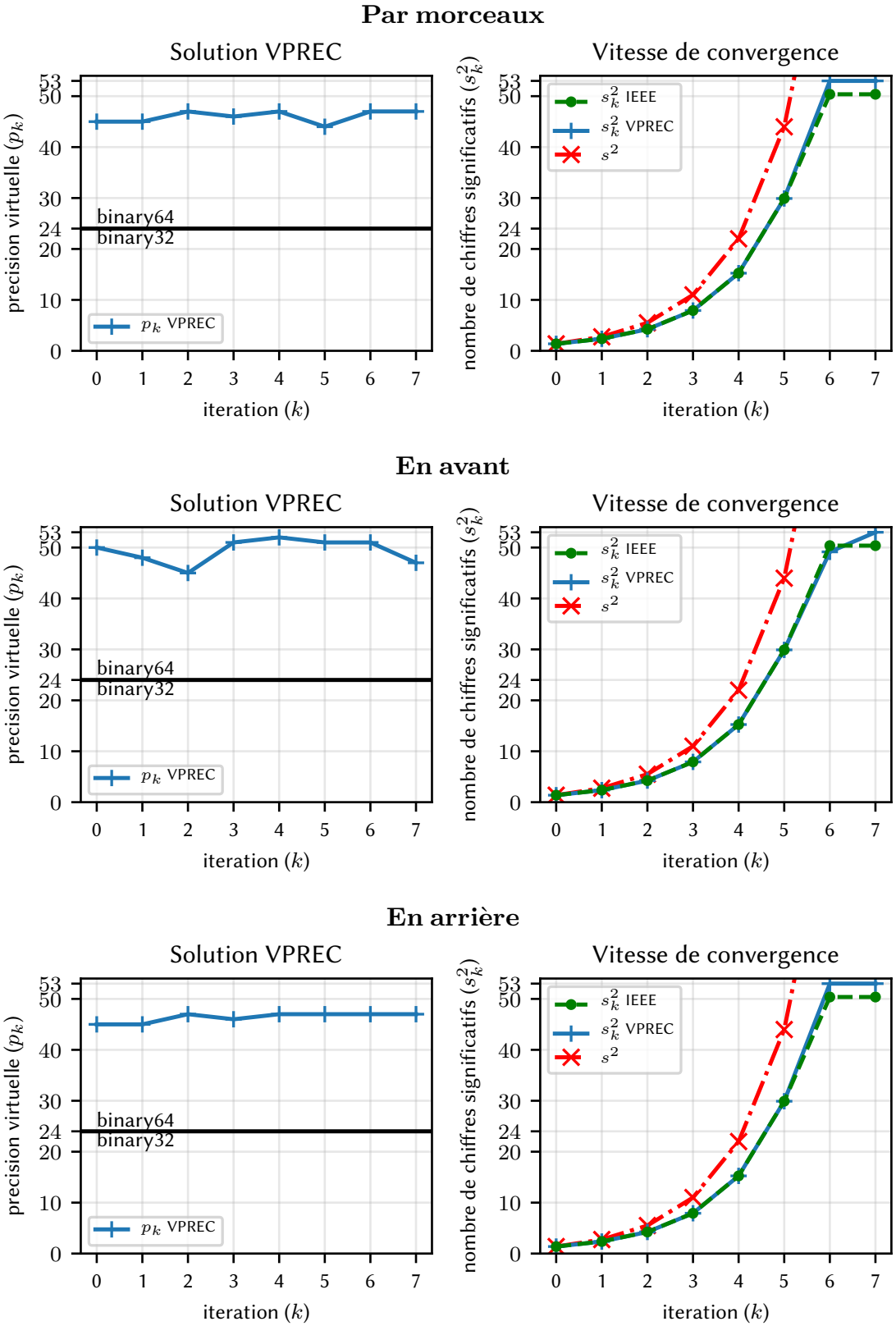


FIGURE 5.12 – Comparaison des explorations pour la méthode de Goldschidt sur $\sqrt{2}$. Est présentée à gauche la solution trouvée par l'algorithme, c'est-à-dire les précisions virtuelles trouvées pour chaque invocation x_k. À droite sont présentées les vitesses de convergence associées. On voit que les trois méthodes d'exploration n'arrivent pas à baisser la précision de calcul. Ceci est dû au fait que la méthode de Goldschmidt n'est pas une méthode auto-correctrice.

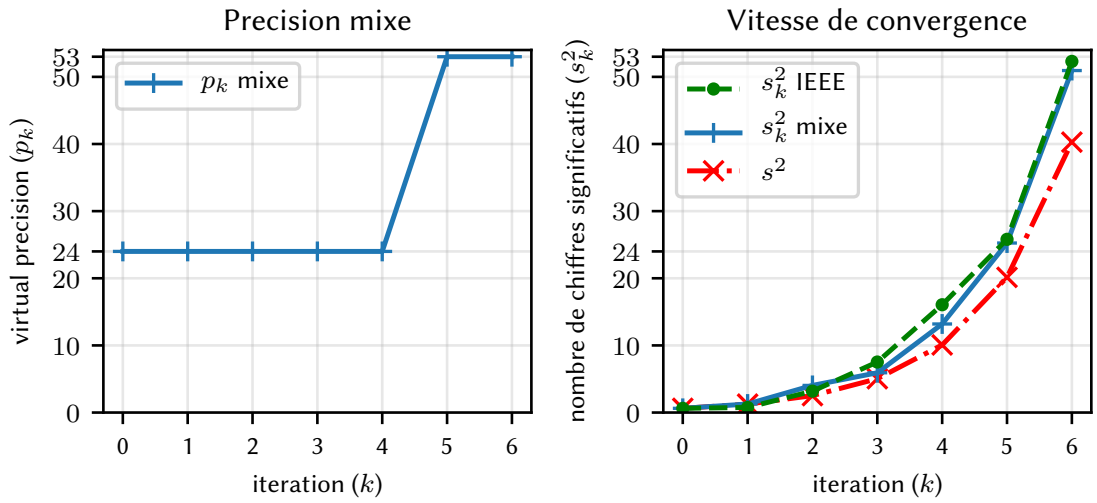


FIGURE 5.13 – Validation de la solution trouvée avec l'approche par morceaux sur la figure 5.7 avec une version en précision mixte. On voit que l'essentiel des itérations sont exécutées en binary32 tout en suivant la même vitesse de convergence que le code originale en binary64.

5.3.1 Validation avec MCA

Pour estimer la sensibilité aux erreurs d'arrondi de la solution trouvée, on utilise le backend MCA. On exécute ainsi chaque invocation à la précision virtuelle donnée par la solution de l'exploration, non plus avec le backend `VPREC` mais avec MCA. On s'assure ensuite que l'algorithme converge toujours malgré la présence de bruit. Cependant la question de la taille des échantillons reste en suspens. Pour y répondre nous utilisons les intervalles de confiance présentés en section 2.6.3.

Pour rappel, ce cadre théorique nous donne une borne précise sur le nombre réel de chiffres significatifs (s) d'un résultat d'un calcul effectué en arithmétique stochastique par rapport à une référence x_{ref}. En fonction de la taille n de l'échantillon MCA pris, c'est-à-dire du nombre de fois que le programme a été exécuté en utilisant le backend MCA, la théorie nous donne la probabilité d'avoir une borne sur s avec un niveau de confiance associé. Il existe deux formules : une pour le cas où la distribution des résultats calculés avec MCA suit une loi normale et une formule si ce n'est pas le cas.

5.3.2 Exemple sur le calcul de $\sqrt{2}$ par la méthode de Newton-Raphson

Pour savoir quelle formule prendre, nous avons testé la normalité de la variable aléatoire $Z = x_{ref} - X$ où X est la variable aléatoire modélisant le résultat du calcul de Newton-Raphson en arithmétique de Monte-Carlo. Nous avons pris $n = 299$ échantillons et $x_{ref} = \sqrt{2}$ comme valeur de référence. On utilise la backend MCA en mode RR.

Le diagramme quantile-quantile de la figure 5.14 montre que les résultats ne suivent pas la loi normale. Ceux-ci auraient du en effet être alignés sur la droite rouge si c'était le cas. Nous avons également utilisé le test de Shapiro-Wilk [163] qui teste l'hypothèse nulle selon laquelle un

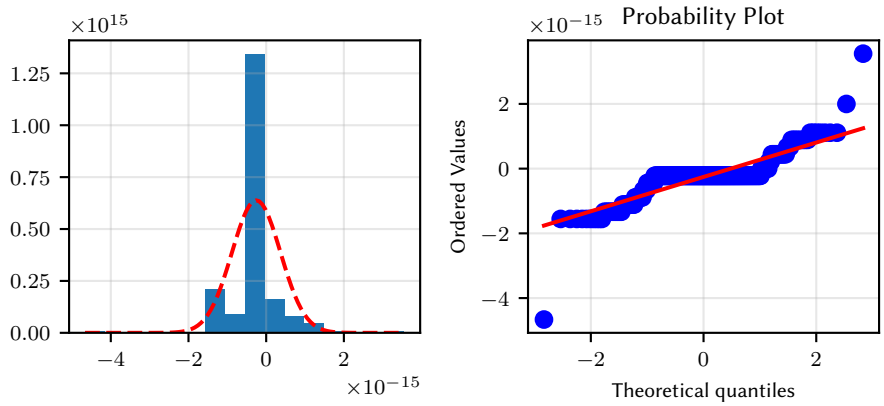


FIGURE 5.14 – A droite sont tracés les quantiles des résultats de la méthode de Newton-Raphson en présence de bruit MCA. Les distributions correspondent à la variable aléatoire $Z = \sqrt{2} - X$ modélisant la différence entre la solution de référence ($\sqrt{2}$) et le résultat d'une exécution de l'algorithme de Newton-Raphson avec MCA. La précision virtuelle de chaque itération étant celle de la solution par morceaux sur la figure 5.7. On voit que les résultats ne suivent pas une loi normale. De plus, le test de Shapiro-Wilk rejette l'hypothèse de normalité.

échantillon est distribué selon une loi normale. Le test rejette la normalité avec W=0.73 et p=1.18e-21. On utilise donc la formule pour le cas d'une distribution non-normale basée sur un estimateur de Bernoulli. Pour rappel, cet estimateur est basé sur le nombre de succès (n_s) au test suivant :

$$|Z_i| \leqslant 2^{-k}$$

où Z_i est le résultat de la i-ème réalisation MCA. Pour un k donné, la probabilité p_k d'avoir k bits significatifs est donnée par $\frac{n_s}{n}$ (le taux de succès).

La figure 5.15 montre la fonction de répartition de la variable Z. Pour chaque k est associé la probabilité que le résultat (X) de la méthode de Newton-Raphson et la valeur de référence x_{ref} est k bits en commun. Cette probabilité est calculée à partir de l'échantillon en bleu et la borne inférieur de l'intervalle de confiance à 95% est donnée par les pointillés oranges. On voit qu'à partir du 48-ème bit la probabilité d'être significatif passe sous la barre des 100%. Cela signifie que notre résultat a 47 bits significatifs avec une probabilité de 99% et un niveau de confiance à 95%.

On s'est également intéressé à la distribution du nombre d'itérations effectuées par les réalisations MCA, dont les résultats sont présentés sur la figure 5.16. On constate que jusqu'à deux itérations supplémentaires sont nécessaires pour mener à bien la convergence. Les barres sur le graphique de gauche présente le nombre de réalisations ayant eu besoin de k itérations pour converger. À droite est présentée la vitesse de convergence associée à chaque barre. Le nombre de chiffres significatifs est calculé à partir de la formule de Bernoulli. Si on s'autorise deux itérations supplémentaires bruitées, on peut estimer que la solution trouvée par notre algorithme d'exploration à une probabilité de convergence en présence d'erreurs d'arrondi de 99% avec une confiance à 95%.

Cette analyse confirme que la solution `VPREC` est robuste aux erreurs d'arrondi ; néanmoins selon la quantité de bruit introduite la méthode de Newton-Raphson converge entre 7 et 9 itérations.

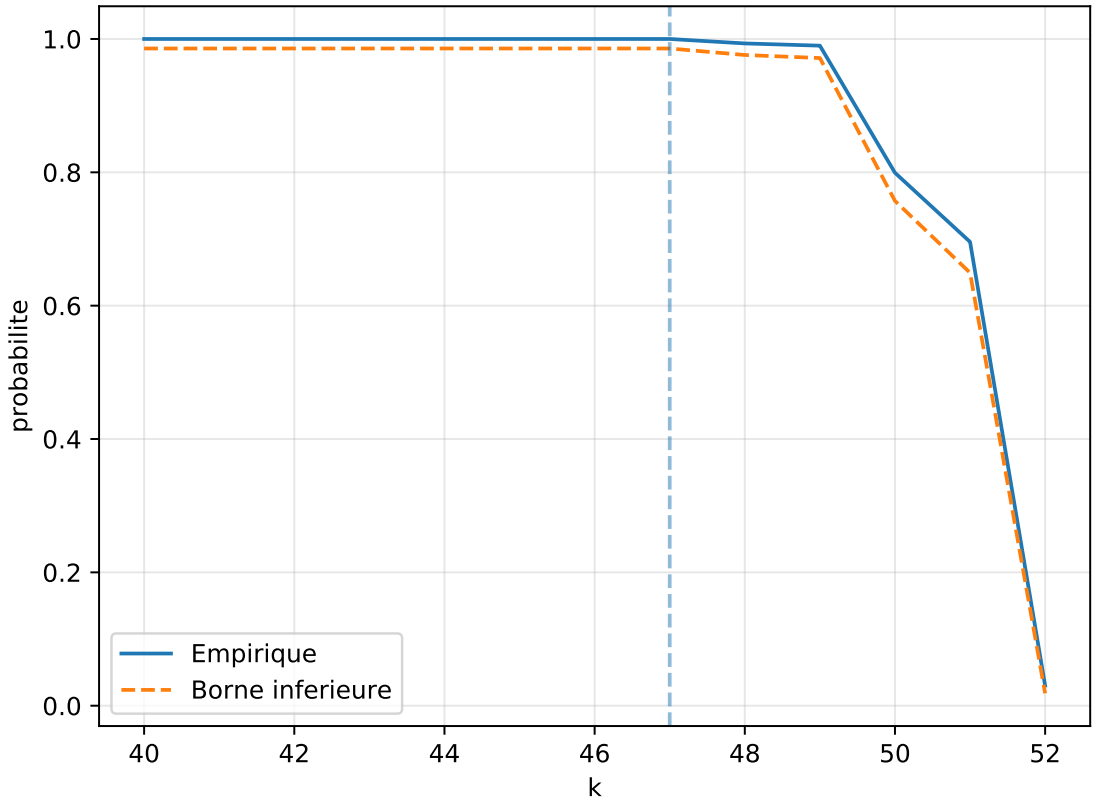


FIGURE 5.15 – Probabilité d'avoir k chiffres significatifs à l'aide de l'estimateur de Bernoulli calculé en bleu à partir des résultats des 299 exécutions MCA. En pointillé orange est affiché la borne inférieure de l'intervalle de confiance à 95%. On voit qu'à partir du 48-ème bit la probabilité d'être significatif passe sous la barre des 100%. Notre résultat à 47 bits significatifs exacts avec une probabilité de 99% et un niveau de confiance à 95%.

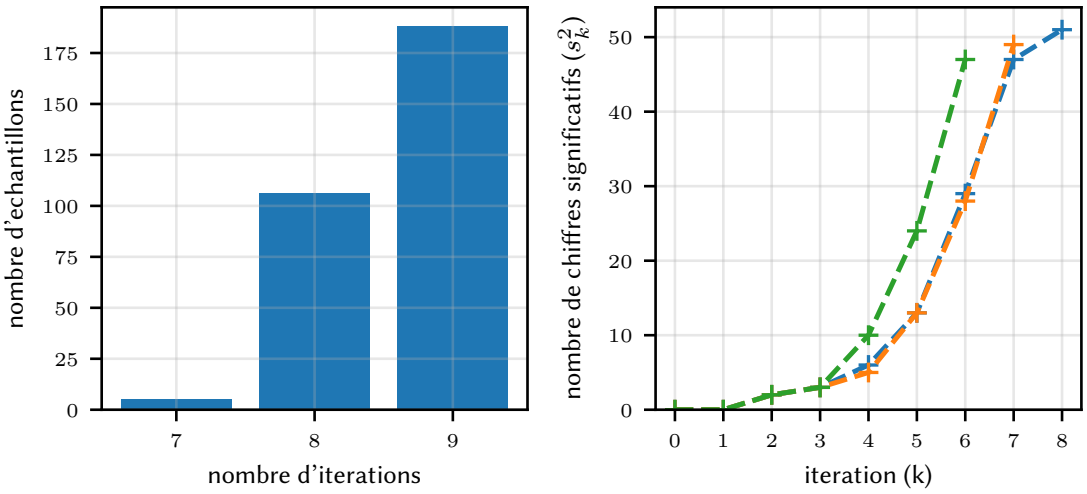


FIGURE 5.16 – À gauche, la distribution du nombre d'itérations nécessaires à la convergence en présence de bruit MCA pour 299 réalisations de Newton-Raphson. Plus de 90% des exécutions ont nécessité 1 à 2 itérations supplémentaires pour converger. À droite, le nombre de chiffres significatifs pour chaque itération k calculé à l'aide de l'estimateur de Bernoulli.

5.4 Optimisation d'un code de calcul industriel

Nous allons dans cette section présenter les résultats obtenus sur l'optimisation du code de calcul industriel YALES2. Nous commencerons par présenter YALES2 et détaillerons sa structure algorithmique. Puis nous analyserons les résultats des explorations d'optimisation sur ce code. Enfin nous terminerons par présenter les performances obtenues lorsqu'on utilise une version en précision mixte.

5.4.1 YALES2

YALES2 est une bibliothèque de calcul de Dynamique des Fluides parallèle qui vise à résoudre les équations de Navier-Stokes instationnaires avec une approximation à faible nombre de Mach pour les écoulements multiphasés et réactifs [133]. Elle gère efficacement des maillages non structurés de plusieurs milliards d'éléments, permettant la simulation numérique directe de configurations de laboratoires et d'essais industriels. Une méthode de projection [28, 149] renforce la contrainte de conservation de masse sur le débit grâce à la résolution d'une équation de Poisson à chaque pas de temps. Dans le contexte du calcul haute performance, cela est généralement réalisé avec les méthodes de Krylov et nous nous concentrons ici sur le DPCG [139, 122, 123] mis en œuvre dans YALES2.

Le cas d'utilisation représentatif sur lequel nous avons étudié YALES2 est le brûleur PRECCINSTA [112, 9]. C'est un brûleur à l'échelle laboratoire utilisé pour valider les solveurs CFD de combustion dérivée d'une technologie industrielle conçu par SAFRAN Helicopter Engines [112], représentatif d'une véritable chambre de combustion de turbine à gaz aéronautique. Il est com-

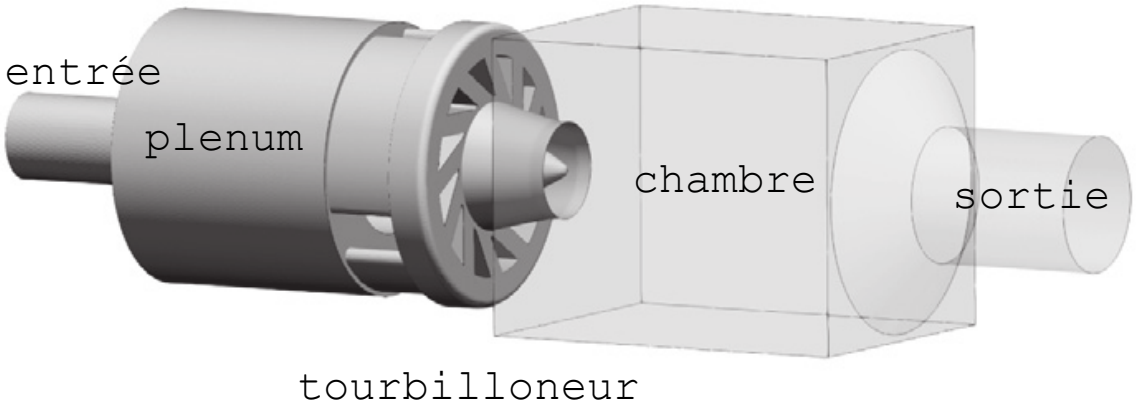


FIGURE 5.17 – Description du brûleur PRECCINSTA tirée de l'article [10]. Le combustible (mélange air-méthane) arrive dans le plenum où il est mélangé à l'aide du tourbilloneur puis injecté dans la chambre de combustion. La figure 5.18 montre la simulation de l'évolution de la combustion au cours du temps avec YALES2.

posé d'une chambre de combustion tourbillonnaire avec un mélange méthane-air dont le détail est présenté sur la figure 5.17. Sur la figure 5.18, on peut voir l'évolution de la combustion dans PRECCINSTA telle que simulée par YALES2.

Méthode du Deflated Preconditionned Conjugate Gradient (DPCG)

Dans cet algorithme présenté sur la figure 5.19, une grille grossière est construite à partir du maillage fin en fusionnant un nombre fixe d'éléments en supercellules (cette procédure est conceptuellement similaire à l'approche multigrille [55]). Le principe général de la déflation est le suivant : la grille grossière est utilisée pour faire converger les modes propres de basse fréquence de la solution qui représentent les interactions à longue portée de l'équation de Poisson. Numériquement, l'opérateur déflaté est résolu en $iterdef$ itérations en utilisant la méthode PCG usuelle de sorte que le critère de convergence soit satisfait. La solution est ensuite étendue et injectée dans la boucle PCG principale de la grille fine. Tout ce processus est répété jusqu'à ce que le maximum du résidu ($maxnorm$) de la grille fine soit en dessous d'un seuil. Dans les deux solveurs Conjugate Gradient (CG) (sur les grilles grossières et fines), le système est préconditionné par l'inverse de la diagonale. Le DPCG de YALES2 a été étudié en profondeur dans la thèse de Malandain [122].

5.4.2 Exploration

Dans toutes les expériences, nous avons limité le nombre total d'itérations effectuées ($\Sigma iterdef_k$) par l'algorithme à moins de 1% d'itérations supplémentaires par rapport à la version originale. Nous utilisons 3 tailles de maillage différentes pour PRECCINSTA : 1,75 million, 40 millions et 870 millions d'éléments tétraédriques. Pour toutes les configurations, la taille des supercellules dans la grille grossière a été fixée à 500 éléments. Pour réduire l'espace de recherche, nous avons considéré deux ensembles de fonctions. Le premier ensemble, nommé *all*, contient toutes les fonctions

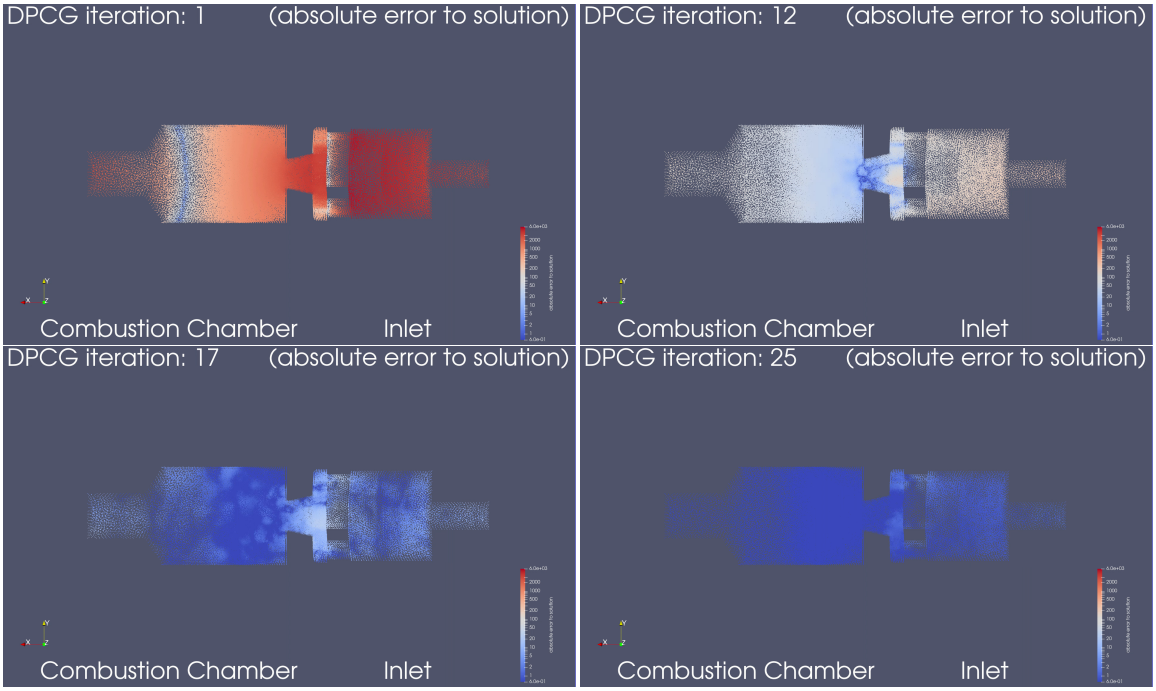


FIGURE 5.18 – Évolution de la combustion dans le brûleur PRECCINSTA avec à droite l'injection du combustible dans la chambre de combustion à gauche, simulée avec YALES2. Le code couleur indique la distance à la solution en erreur absolue. Le bleu foncé signifie qu'on est proche de la solution. Chaque image correspond à une itération du DPCG sur la grille fine.

Require : A, b, M^{-1}, W
initialization
for k = 0 ,1 ,... until required convergence **do**
$\alpha_{k+1} = \frac{r_k^T w_k}{p_k^T A p_k}$
$x_{k+1} = x_k + \alpha_{k+1} p_k$
$r_{k+1} = r_k - \alpha_{k+1} A p_k$
Solve $\hat{A} d_{k+1} = W^T (A M^{-1} - I) r_{k+1}$
$w_{k+1} = M^{-1} r_{k+1} - W d_{k+1}$
$\beta_{k+1} = \frac{r_k^T w_{k+1}}{r_k^T w_k}$
$p_{k+1} = w_{k+1} + \beta_{k+1} p_k$
end for

FIGURE 5.19 – Algorithme du DPCG. En vert sont soulignées les étapes effectuées sur la grille fine et en bleu celles effectuées sur la grille grossière.

utilisées par YALES2. Le second ensemble, appelé *deflated*, est l'ensemble des fonctions utilisées pour résoudre la déflation de l'opérateur sur la grille grossière.

Dans les deux explorations suivantes, nous présentons l'exploration sur le maillage de 1,75M. Nous avons fixé statiquement la plage des exposants à 8 bits dans l'exploration avec `VPREC` ce qui nous permettra de vérifier s'il est possible d'utiliser la dynamique d'un binary32 sans obtenir d'overflow.

Exploration sur l'espace *all*

Les deux figures 5.20 de gauche montrent les résultats des explorations sur la grille grossière et la grille fine en même temps. Sur les deux graphiques, l'axe des x représente le nombre d'itérations sur la grille fine. Sur le graphique du haut, l'axe des y à droite représente sur la même échelle la norme *maxnorm* du résidu entre deux itérations successives et le critère de convergence *convcrit* de l'opérateur déflaté. L'axe y à gauche représente *iterdef*, le nombre d'itérations sur l'opérateur déflaté. Sur le graphique du bas, l'axe des y représente la précision virtuelle utilisée pour calculer l'itération donnée sur l'axe des x.

Pour l'exploration *all*, nous pouvons tout d'abord noter sur le graphique du haut que notre solution conserve le même profil de convergence que l'exécution de référence IEEE (en transparent). On voit en effet que le résidu de la norme *maxnorm* est similaire pour notre solution et la référence IEEE, de même que pour le nombre d'itérations sur la grille grossière (*iterdef*). On remarque également que la précision requise augmente avec les itérations. Ceci est attendu puisque le solveur a besoin de plus en plus de précision pour converger à mesure qu'il affine la solution. Étonnamment, la précision requise passe de 34 bits à 21 bits à l'itération 50.

Point intéressant sur le graphique du bas, nous observons que la précision requise est proche

de celle d'un binary32 (représentée par la bande jaune). En effet, la précision maximale requise par notre algorithme est de 33 bits, soit 20 bits de moins que de disponible dans un binary64. En moyenne on obtient une réduction de 43% comparé à la version originale. Soit une réduction de 49% de la précision Avec un support matériel de formats flottants variable, nous aurions pu mesurer les gains en performances apportées par notre solution, ce qui aurait démontré l'avantage de notre méthode comparé aux approches en précision mixte classique qui n'aurait pas pu optimiser ce code.

Exploration sur l'espace $deflated$

Les deux figures 5.20 de droite montrent le résultat de l'exploration sur la grille grossière. Sur les deux graphiques, l'axe des x représente le nombre d'itérations sur la grille fine. Sur le graphique du haut, l'axe des y à droite représente, sur la même échelle, la norme *maxnorm* du résidu entre deux itérations successives et le critère de convergence *convcrit* de l'opérateur déflaté. L'axe y gauche représente $iterdef$, le nombre d'itérations sur l'opérateur déflaté. Sur le graphique du bas, l'axe des y représente la précision virtuelle utilisée pour calculer l'itération donnée sur l'axe des x.

Pour l'exploration *deflated*, le graphique du haut montre que notre solution conserve le même profil de convergence que l'exécution de référence IEEE (en transparent). On voit en effet que le résidu de la norme *maxnorm* est similaire pour notre solution et la référence IEEE. Cependant notre réduction de précision a impacté la convergence sur la grille grossière puisque la courbe des *iterdef* est légèrement inférieur à celle de la référence IEEE avec au total 3549 itérations sur la grille grossière contre 3577 pour la version IEEE. Notre solution a donc légèrement améliorée la convergence.

Concernant la précision requise, la figure du bas montre qu'il faut moins de 23 bits de précision pour l'opérateur déflaté avec une précision moyenne de 18 bits. Soit une réduction de près de 60%. On peut même noter que la dernière itération requière 0 bit de précision signifiant que le calcul sur la grille grossière n'est pas nécessaire. Par conséquent, l'opérateur déflaté peut être calculé avec du binary32, ce que nous vérifions avec une implémentation en précision mixte détaillée dans la section 5.4.5.

5.4.3 Accélération de la convergence

L'algorithme d'exploration permet de trouver des solutions qui requièrent moins d'itérations sur la grille fine comme le montre l'exemple de la figure 5.21. En effet, contrairement à l'exécution originale en binary64 qui nécessitait 100 itérations, la solution trouvée converge au bout de 65 itérations, soit 25 itérations de moins.

Néanmoins, lorsque nous analysons les résultats de plus près, on s'aperçoit que le nombre d'itérations sur la grille grossière est pratiquement le même pour la version originale et la solution trouvée avec `VPREC`. On observe que le nombre d'itérations sur la grille grossière effectuées aux itérations 30 à 34 sur la grille fine à atteint un plafond. Ce calcul supplémentaire intervenu en amont de la convergence a permis au DPCG de converger plus tôt que dans la version binary64. On constate que la version binary64 peine à converger à partir de la 65ème itération et n'effectue que très peu d'itérations sur la grille grossière.

Si la solution trouvée par notre algorithme n'a pas réduit le nombre de calculs au total, elle a néanmoins permis au solveur de ne pas stagner sur les dernières itérations tout en utilisant une précision moyenne plus basse.

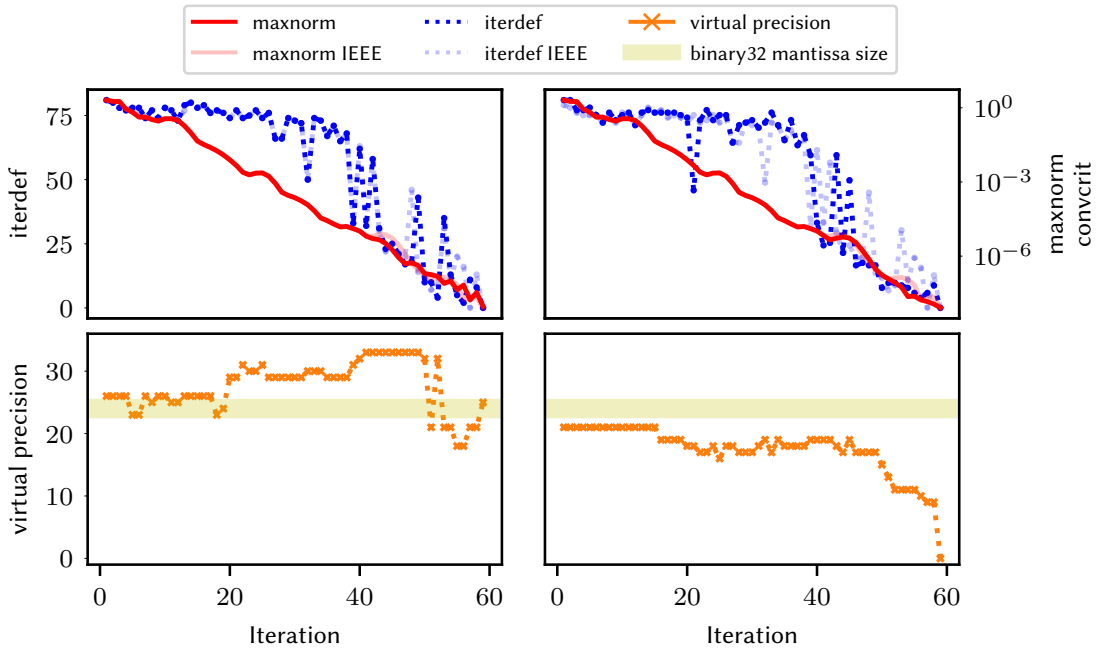


FIGURE 5.20 – Recherche de la précision minimale sur le DPCG de YALES2 sur le code entier (à gauche) et sur la partie déflatée (à droite). Sur les deux graphiques, nous pouvons voir que notre solution de précision réduite suit le profil de convergence de la solution IEEE-754 de référence.

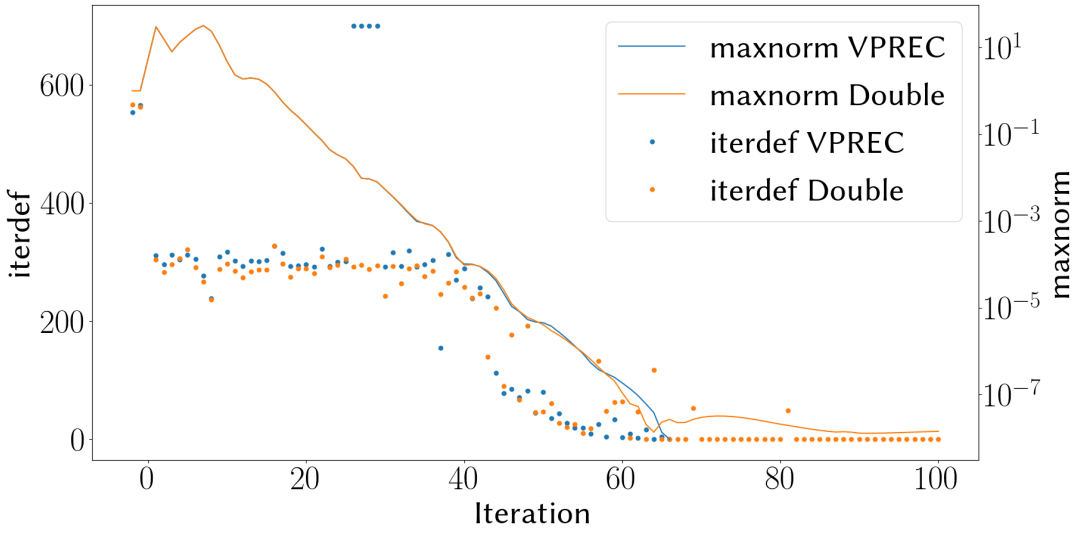


FIGURE 5.21 – Exploration sur PRECCINSTA avec 850 millions d'éléments dans le maillage sur 112 coeur MPI. La solution trouvée par l'algorithme d'exploration réduit de 25 itérations (65 contre 100) la convergence. Le nombre total d'itérations sur la grille grossière reste néanmoins le même entre les deux versions.

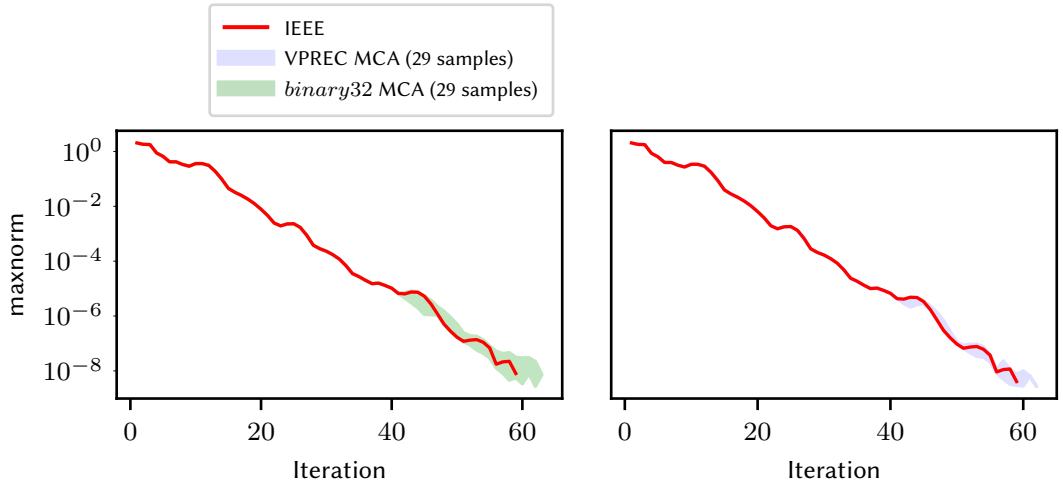


FIGURE 5.22 – Résilience des configurations VPREC et binary32. En rouge, la convergence de *maxnorm* pour la solution de référence IEEE-754. L'enveloppe bleue montre les 29 réalisations MCA de la configuration obtenue avec VPREC trouvé en section 5.4.2. L'enveloppe verte montre les 29 réalisations MCA pour la configuration binary32. Toutes les réalisations convergent, ce qui montre bien la résilience des deux configurations.

5.4.4 Validation de la solution aux erreurs d'arrondi

Dans les sections 5.4.2 et 5.4.2 précédentes, nous avons démontré que le DPCG de YALES2 converge avec un format de calcul réduit. Comme nous l'avons expliqué en section 5.3 sur l'exemple de Newton-Raphson, ce résultat n'est valable qu'avec le mode d'arrondi particulier utilisé par VPREC et est sensible aux données d'entrée. Dans une configuration réaliste, de petites erreurs d'arrondi peuvent se produire lors de l'exécution d'opérations flottantes dans un environnement logiciel et matériel différent. Nous avons donc utilisé le backend MCA comme deuxième étape dans notre analyse pour trouver une configuration qui résiste aux erreurs d'arrondi. La loi de probabilité du résultat calculée avec YALES2 ne nous étant pas connue, nous avons modélisé ce processus comme une épreuve de Bernoulli.

Nous avons pris 29 réalisation MCA pour simuler l'effet des erreurs d'arrondi. Si l'un des échantillons ne converge pas, nous concluons que la solution n'est pas robuste aux erreurs d'arrondi. Par contre, si les 29 réalisations convergent, nous concluons, grâce aux intervalles de confiance présentés en section 2.6.3, que la probabilité de convergence en présence d'erreurs d'arrondi est de 90% avec un niveau de confiance de 0,95.

La figure 5.22 montre que la solution trouvée dans la section 5.4.2 est robuste et converge pour toutes les réalisations. Comme la solution est proche du format binary32, notre objectif est d'obtenir une configuration binary32 robuste : la configuration binary32 à précision constante représentée par l'enveloppe rouge clair de la figure 5.22 converge en 57 à 63 itérations en présence d'erreurs d'arrondi pour l'ensemble de l'échantillon. Ceci démontre qu'il est possible de réécrire avec une certaine confiance l'opérateur de la grille grossière du DPCG en binary32.

5.4.5 Évaluation en précision mixte

Nous avons évalué la version en précision mixte sur les trois différentes grilles de PRECCINSTA avec un critère de convergence à 10^{-9}. Nous avons pour cela utilisé le cluster CRIANN constitué de 366 nœuds bisocket Intel Xeon E5-2680 et d'une interconnexion Intel Omnipath. Les statistiques ont été recueillies en utilisant l'interface Intel IPM pour Intel MPI.

Comme prévu par notre méthodologie, le calcul converge et toutes les versions satisfont aux contraintes de validité (temporelle et numérique). Cependant, nous avons remarqué que les expériences de plus grandes tailles nécessitent des itérations initiales en binary64 supplémentaires sur la grille déflatée. Par exemple, respectivement deux et quatre itérations supplémentaires en binary64 sont nécessaires pour les mailles 40M et 870M. Ceci est cohérent avec les observations de Cools et al. [35] sur l'importance d'être précis dans la première itération d'une récurrence d'un CG.

Nous avons également remarqué que dans ces cas plus gros, il est nécessaire de passer du binary32 au binary64 pour la grille déflatée lorsque le critère de convergence déflaté est difficile à atteindre avec du binary32 $\sim 10^{-8}$. On retrouve ici le principe du raffinement itératif.

Cet effet n'est pas apparu dans le cas plus petit de 500 éléments par groupe. Notre hypothèse est que la différence de granularité entre les deux niveaux de grille est plus grande sur le petit maillage et donc les petites erreurs sur l'itération de la grille grossière ont moins d'impact sur les itérations de la grille fine [122, 123].

Nous avons mesuré une réduction de 28 à 31% du volume de communication. Le gain d'énergie peut être estimé comme étant linéairement lié à ce gain de volume avec le modèle simple proposé dans [3].

Puisque DPCG est principalement limité par la latence des communications, le gain de performance est limité lorsque le nombre de processeurs augmente pour une taille donnée, passant de 28% d'accélération à 2% de ralentissement lors d'expériences critiques de mise à l'échelle forte. Cependant, selon ces résultats et l'utilisation du code par l'utilisateur final, l'accélération prévue pour l'utilisation quotidienne sera de l'ordre de 10%.

5.5 Conclusion

L'optimisation de la précision de calcul est un problème difficile. Jusqu'à présent les méthodes d'optimisations proposées dans la littérature abordaient la chose principalement selon l'angle spatial. Nous avons démontré que la prise en compte de la dimension temporelle apporte un réel bénéfice sur la qualité des solutions trouvées. Nous avons en effet montré qu'il était possible de réduire de près de moitié la précision des calculs sur l'ensemble de l'application YALES2 et de près de 60% ceux de l'opérateur déflaté de son DPCG. D'autre part, l'optimisation de la précision à grain fin (au niveau du bit) apporté par `VPREC` permet d'exhiber des solutions hors de portée pour les recherches en précision mixte classiques.

Nous avons démontré la viabilité de notre algorithme d'exploration de recherche pour trouver des solutions réduisant la précision de manière conséquente sur des espaces de recherche très grands. Notre approche avec MCA a également montré que l'on pouvait obtenir une estimation précise de la résilience aux erreurs d'arrondi des solutions trouvées par notre algorithme et ceux dans un cadre mathématique solide par l'intermédiaire des intervalles de confiance.

Enfin, l'utilisation de notre méthode sur YALES2 a démontré son passage à l'échelle. L'utilisation de la précision mixte nous a permis de valider la solution trouvée par notre méthode d'exploration sur des maillages allant jusqu'à 850 millions d'éléments et exécutés sur un nombre de processus MPI

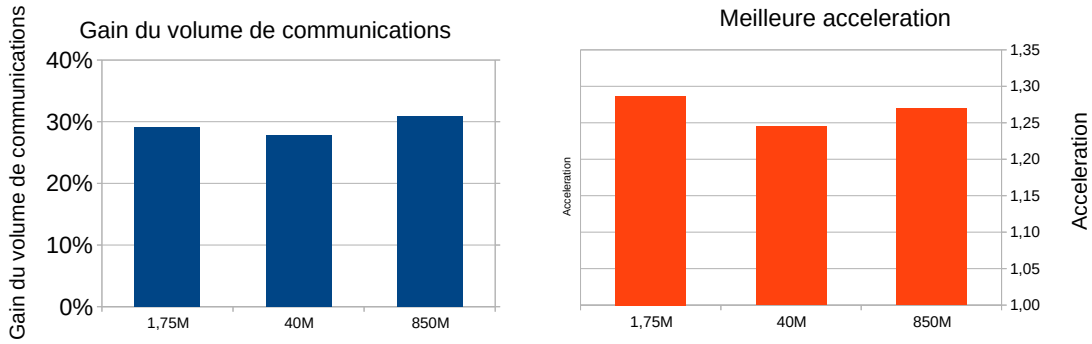


FIGURE 5.23 – Pour chaque taille de maillage sont présentés les gains obtenus sur le volume de communications échangée entre les processus MPI et la meilleure accélération obtenue parmi les différents nombres de processeurs pris. La version de YALES2 avec l'opérateur déflatée en précision mixte a permis de réduire de près de 30% le volume des communications. D'autre part on observe des vitesses d'exécutions jusqu'à $1,30\times$. Les facteurs d'accélérations observés sont principalement dûs à la réduction des communications car la vitesse d'exécution est dominée par la latence des communications entre noeuds MPI.

jusqu'à 560 coeurs. Cette version mixte a montré un gain net moyen de la performance du code en réduisant le volume de communications MPI de 30% en moyenne et en augmentant la vitesse d'exécution d'un facteur jusqu'à $1,30\times$.

Chapitre 6

Conclusion et Perspectives

Cette thèse s'est attachée à développer des outils de débogage numérique et d'optimisation de formats de calcul flottant dans un contexte HPC. L'accent a été mis sur l'ergonomie et la facilité d'utilisation pour l'utilisateur non expert du domaine de l'arithmétique flottante ainsi que sur le passage à l'échelle des outils proposés. Si le parallèle est osé, le but a été d'offrir, à la manière d'un GDB accessible aux non-experts de la compilation, des outils accessibles aux développeurs pour déboguer et optimiser les calculs flottants dans un contexte HPC. Nous sommes encore loin du niveau de sophistication d'un GDB mais souhaitons pousser ces travaux pour qu'ils en suivent les mêmes pas.

6.1 Réponses aux problématiques

Nous allons présenter en détail les réponses proposées aux problématiques posées en introduction dans la section 1.2. Pour chaque problématique, les réponses apportées se résument ainsi :

1. **Comment estimer la justesse d'un code de calcul ?**
   Une chaîne d'outils pour analyser la qualité numérique des codes au cours du temps à l'aide du modèle MCA ainsi que des méthodologies pour appliquer efficacement le modèle MCA afin de localiser des instabilités numériques.
2. **Comment optimiser la précision d'un code de calcul ?**
   `VPREC`, un backend simulant n'importe quel format similaire à IEEE-754 tenant sur un binary64 et une méthode d'optimisation des formats de calcul flottant à grain fin selon la dimension temporelle.
3. **Comment passer à l'échelle les méthodes d'analyse ?**
   La réponse apportée par cette thèse a été transverse aux deux problématiques précédentes.
   Pour la problématique 1.2, cette thèse a optimisé le backend `MPFR` de Verificarlo d'un facteur $\times 2,5$, développé le backend `BITMASK` offrant un compromis entre surcoût et qualité du bruit numérique introduit, proposé deux méthodes d'identification d'instabilités numériques en mode boîte noire et démontré leur mise à l'échelle sur le code de chimie quantique ABINIT.
   Pour la problématique 1.2 le changement de format du backend `VPREC` est dynamique, paramétrable et nécessite une seule passe de compilation. Cette thèse a également démontré le

passage à l'échelle de la méthode d'optimisation temporelle de format à grain fin sur le code de dynamique des fluides YALES2.

Enfin Veritracer a été pensé pour limiter les surcoût liés à la sortie des contextes d'appels en maintenant une pile des entrées et sorties de fonctions. De plus un soin a été apportée pour réduire la taille des traces numériques en utilisant un format binaire qui permet de paralléliser la lecture des traces pour améliorer la vitesse de traitement.

6.1.1 Comment évaluer la justesse des codes ?

Dans le chapitre 2 nous avons posé les définitions pour désambiguïser la polysémie des mots précision, juste ou exact dans la vie courante. En plus de poser les notions de métrologie et de quantification d'erreurs pour le calcul flottant utiles pour la thèse, l'objectif de ce chapitre a été de poser une base de travail commune avec les personnes extérieures au domaine de l'arithmétique flottante.

Dans le chapitre 3 nous avons montré la pertinence du modèle MCA pour estimer la qualité numérique du résultat d'un calcul flottant à l'aide de la notion de chiffres significatifs. De plus, nous avons montré qu'il était possible d'avoir des bornes rigoureuses sur ce résultat grâce aux travaux théoriques récents sur les intervalles de confiance.

Dans le chapitre 4 nous avons montré que les informations fournies par le modèle MCA au travers de l'outil Verificarlo pouvai être améliorées en analysant la qualité numérique des calculs au cours du temps. Nous avons pour cela développé la chaîne d'outils Veritracer qui permet de visualiser la qualité des calculs au cours du temps sur la base de traces générées automatiquement durant l'exécution du programme cible. Ces traces sont de plus enrichies d'informations sur le contexte des calculs, comme le nom de la variable, la fonction appelante, l'endroit du calcul dans le code source ainsi que le contexte d'appels.

Dans le code de chimie quantique ABINIT, Veritracer a permis de visualiser la qualité de la fonction `simp_gen` qui calcule la valeur numérique d'une intégrale par la méthode de Simpson. Nous avons montré qu'il était possible d'améliorer la qualité de `simp_gen` grâce à l'algorithme compensé `Dot2` en la visualisant avec Veritracer. Nous avons, de plus, montré le bénéfice apporté par les contextes d'appels et l'analyse du flot de données pour comprendre les pertes de chiffres significatifs de `simp_gen` même en utilisant `Dot2`.

Nous avons également développé deux méthodologies (filtrage et classification) pour identifier rapidement les fonctions qui contribuent directement à l'erreur globale d'un code. Nous avons pour cela développé la notion de degré de sensibilité pour comparer la contribution à l'erreur globale de deux fonctions. Cette relation nous semble pertinente pour aider le développeur d'application à hiérarchiser ses priorités d'actions. Enfin, nous avons appliqué ces deux méthodes sur le code ABINIT, ce qui nous a permis de révéler des fonctions ayant un impacte direct sur le principal résultat physique calculé par ABINIT.

6.1.2 Comment optimiser les formats de calcul ?

Dans le chapitre 5 nous avons présenté le backend `VPREC` permettant de simuler à grain fin (au niveau du bit) des formats de calcul similaires à ceux de la norme IEEE-754. Ce backend est paramétrable par l'utilisateur et permet de changer dynamiquement le format de calcul d'un code instrumenté par Verificarlo ce qui permet notamment de tester rapidement différents formats sans recompilation de la part du développeur.

Nous avons montré l'utilité du backend VPREC pour optimiser le format des calculs d'une application. Nous avons développé une heuristique de recherche déclinée en plusieurs versions permettant de trouver automatiquement, à partir de critères utilisateur, un format de calcul réduit pour les applications basées sur des schémas itératifs. La contribution majeure de notre méthode est l'optimisation du format de calcul selon la dimension temporelle, dimension peu inexploitée jusqu'à présent. Pour une instance du code donnée, cette heuristique a été validée sur la méthode de Newton-Raphson où nous avons montré qu'elle permettait de réduire la précision de calcul tout en préservant la vitesse de convergence.

Nous avons également montré l'efficacité de cette méthode sur YALES2, un code HPC de simulation de dynamique des fluides. YALES2 est basé sur un DPCG dont nous avons montré qu'il était possible de réduire la précision de calcul de l'opérateur déflaté de près de 60% et d'un peu moins de la moitié pour l'application globale. Nous avons montré que les solutions trouvées par notre algorithme de recherche étaient robustes numériquement grâce à la méthode MCA. Cela nous a permis de monter qu'il était possible de réécrire l'opérateur déflaté en précision mixte avec un certain niveau de confiance. Enfin, cette réécriture nous a permis de montrer des gains de performances sur YALES2 avec une réduction de jusqu'à 28% des communications entre processeurs et un facteur d'accélération jusqu'à $1,30\times$, le tout pour des maillages de 1,75 à 860 millions d'éléments sur 1 à 560 coeurs MPI.

6.1.3 Comment passer à l'échelle les méthodes d'analyse ?

Dans le chapitre 3, il a été montré comment nous avons optimisé le backend MPFR pour multiplier ses performances par $\times 2,5$. Nous avons également présenté le backend BITMASK et montré qu'il offrait un bon compromis entre surcoût d'exécution et qualité du bruit numérique injecté.

Ce backend BITMASK a montré son utilité dans le chapitre 4 où il a permis une évaluation rapide du degré de sensibilité (cf.def 4.9.2) des fonctions d'ABINIT. Dans ce même chapitre nous avons montré le passage à l'échelle des méthodes de *filtrage* et de *classification* pour révéler les fonctions ayant un impacte sur le résultat d'intérêt d'ABINIT. Nous avons également montré que Veritracer pouvait être utilisé sur le code ABINIT avec un léger ralentissement de $\times 1,18$ en moyenne.

Dans le chapitre 5 nous avons montré la mise à l'échelle de la méthode d'exploration pour réduire les formats de calcul sur le code YALES2. Nous avons montré que notre méthode pouvait être employée sur une exécution en mémoire distribuée allant de 1 jusqu'à 560 coeurs et sur des données allant de 1,75 à 850 millions d'éléments par maillage.

6.2 Perspectives

6.2.1 À court terme

À court terme, les perspectives de cette thèse sont :

- Étendre les fonctionnalités du backend VPREC pour permettre de relâcher les contraintes de la norme IEEE-754. Il nous semble intéressant de proposer une option similaire aux Denormals As Zero (DAZ)/Flush To Zero (FTZ) de l'extension SSE des architectures Intel qui permet de considérer les nombres dénormalisés comme des 0. En effet, leur traitement est coûteux en matériel [93] et les ignorer peut faire gagner plusieurs cycles d'instructions dans des sections critiques de codes.

- Modifier la chaîne de compilation de Verificarlo pour tirer parti du Link Time Optimization (LTO) de LLVM. Le LTO permet d'appliquer des optimisations durant la phase de linkage. Les backends seraient alors optimisés avec l'application d'origine ce qui réduirait les indirections.
- Explorer d'autres générateurs de nombres aléatoires (RNG). Le RNG utilisé dans Verificarlo est en effet un générateur de très bonne qualité mais néanmoins coûteux par rapport au temps d'exécution global des backends. Il nous semble intéressant d'explorer des générateurs disposant d'une moins bonne qualité, comme par exemple d'une période plus courte, mais qui sont plus rapides d'exécution.
- Implémenter une version vectorisée des backends pour les instructions vectorielles. L'implémentation actuelle de Verificarlo pour les instruction vectorielles consiste à les dépaqueter, appliquer séquentiellement le bruit sur chacun des éléments et repaquer les éléments dans le vecteur d'origine. Ce paquetage/dépaquetage peut être la source d'un certain ralentissement pour les applications utilisant intensivement la vectorisation. Une version vectorielle des backends permettrait alors une accélération notable de la vitesse d'exécution, en particulier pour les applications utilisant intensivement la vectorisation.
- Coupler les solutions données par l'heuristique de recherche sur l'optimisation des formats de calculs avec des modèles de prédictions de performances. Nous avons pensé qu'il serait, entre autres, judicieux de coupler ces informations avec un modèle de prédiction de volume de communication, un simulateur de mémoires caches ou de vectorisation pour estimer les potentiels facteurs d'accélération.

6.2.2 À plus long terme

Analyse de sensibilité

Pour une fonction $f(x) = y$ donnée, l'analyse de sensibilité cherche à étudier les effets sur la sortie y d'une variation sur les entrées x. Deux types de questions se posent alors : premièrement sur les aspects qualitatifs, on souhaite savoir si deux paramètres ont un effet conjoint sur la sortie. Secondement, sur les aspects quantitatifs, on souhaite pouvoir comparer les effets de deux paramètres. Nous reprenons le formalisme introduit dans [175] pour une meilleure compréhension du problème. Soit

$$\begin{aligned} f : \mathbb{R}^n &\to \mathbb{R} \\ x &\mapsto y \end{aligned}$$

la fonction implémentée par le programme. On cherche

$$f : D = D_1 \times \dots D_2 \times \dots D_n$$

la fonction multivariée avec D_i l'intervalle de variation du *facteur* x_i à expliquer. L'idée de l'analyse de sensibilité globale est de considérer f comme une somme de fonctions $f_{\mathfrak{u}}$ ne dépendant que des $x_i \in \mathfrak{u}$, $\mathfrak{u} \subseteq \mathcal{P}(\{1, \dots, n\})$:

$$f(x) = f_{\varnothing} + \sum_{i=1}^{n} f_i(x_i) + \sum_{i=1}^{n} \sum_{\substack{i=1 \\ j \neq i}}^{n} f_{ij}(x_i, x_j) + \dots f_{1\dots n}(x_1, x_2, \dots, x_n).$$

Cette décomposition produit des *effets principaux* $f_i(x_i)$ et des sommes d'interactions $f_{\mathfrak{u}}(\mathbf{x}_{\mathfrak{u}})$, $|\mathfrak{u}| \leqslant 2$ [1] ou *effets secondaires*.

Un parallèle peut être dressé entre les méthodes de filtrage et de classification développées dans le chapitre 4. En effet, la méthode de filtrage peut être vue comme une estimation des effets principaux non nuls en prenant comme facteurs les fonctions d'un programme perturbées avec MCA. La méthode de classification esst alors vue comme un moyen de comparer les effets non nuls. En poursuivant l'analogie, nous aimerions également estimer les effets secondaires. Néanmoins, lorsque le programme comporte un grand nombre de fonctions, on retombe sur des problèmes d'explosion combinatoires évoqués en section 4.9.1.

Plusieurs solutions existent pour éviter une explosion combinatoire du nombre de facteurs à analyser : ne regarder qu'un certains nombres d'interactions entre facteurs, ce qui revient à limiter la taille de $\mathfrak{u}$, ou partitionner l'application pour dégager des groupes de fonctions sans liens de calcul, diminuant ainsi le nombre de facteurs à analyser. De nombreux travaux ont été menés pour partitionner les applications HPC en morceaux indépendants [148, 150, 22] appelés *codelets* afin d'en caractériser les performances. Le partitionnement d'application est une autre perspective que nous souhaiterions explorer.

Approche par codelets

Un *codelet* est un morceau d'un code rejouable indépendemment de l'application originale. L'approche par *codelets* consiste à partitionner une application en *codelets* indépendants à l'aide d'outils comme CERE [22]. Pour pouvoir modifier, compiler et exécuter les *codelets* indépendemment du code d'origine, CERE capture la mémoire qu'une région (*codelet*) utilise durant l'exécution orignale du code. CERE travaille à la granularité d'une page mémoire en ne sauvegardant que les pages touchées, ce qui permet de limiter la taille des sauvegardes. CERE supporte également la capture d'applications parallèles multi-threadées avec une capture parallèle que j'ai implémenté durant mon stage de fin de Master [150]. CERE rejoue un *codelet* en chargeant sa mémoire dans la configuration initiale au moment de la capture ainsi que l'état de la mémoire cache ce qui permet d'obtenir des mesures précises de performance lors du *replay*.

L'approche par codelets s'est montrée très puissante pour caractériser les performances d'un code HPC [150]. Dans le cadre de l'analyse numérique de code, nous voyons trois intérêts à pouvoir rejouer un noyau de calcul indépendamment de l'application originale : 1) faire des analyses de sensibilité sur des régions plus petites que l'application originale, 2) essayer de caractériser *numériquement* un noyau de manière analogue à la caractérisation de performances 3) coupler spatialité et temporalité pour optimiser les formats de calcul d'une région du code.

- **Analyse de sensibilité sur une région**
  L'intérêt des *codelets* est qu'ils représentent une petite section de code au sens qu'ils sont plus rapides à exécuter et qu'ils délimitent une zone restreinte. De plus, ils peuvent être rejoués de manière indépendante. En couplant l'approche par *codelet* avec le modèle MCA, on peut alors rapidement tester les effets de plusieurs combinaisons de facteurs là où le test sur l'application entière prendrait trop de temps.
- **Caractérisation de noyaux de calcul**
  De manière analogue à la caractérisation de performance, il serait intéressant de pouvoir caractériser *numériquement* une séquence de calculs pour par exemple prédire l'impact d'op-

1. $\mathbf{x}_{\mathfrak{u}} = Px, P = (\delta_{ij}\chi_{\mathfrak{u}}(i))_{1\leqslant i\leqslant n, 1\leqslant j\leqslant n}$, avec δ le delta de kronecker et $\chi_{\mathfrak{u}}$ la fonction caractéristique de $\mathfrak{u}$.

timisations de compilation sur la qualité numérique. Les métriques à choisir pour caractériser *numériquement* un code restent pour nous une question ouverte.

- **Couplage spatialité/temporalité**
  La méthode développée dans le chapitre 5 s'est focalisée sur la dimension temporelle de l'optimisation, laissant de côté la dimension spatiale. Or le partitionnement d'application fournit un découpage spatial que nous trouvons utile d'exploiter. En effet, nous pensons qu'appliquer notre méthode de recherche temporelle sur chacun des codelets permettrait d'améliorer la qualité des solutions trouvées, en proposant une solution adaptée pour la région de l'espace couverte par le codelet. Cette perspective ouvre sur des problématiques d'estimation des effets des baisses de précision sur l'erreur globale et rejoint les problématiques d'analyse de sensibilité évoquées précédemment puisqu'un *codelet* rejoue seulement une partie de l'application.

Projet Interflop

Nous avons montré en section 1.3 les outils existants pour analyser les calculs flottants dans des codes. Les approches développées par ces outils s'intersectionnent parfois avec Verificarlo comme l'arrondi aléatoire implémenté par Verificarlo, CADNA et Verrou mais sont souvent complémentaires. Nous pouvons citer comme exemple l'instrumentation binaire de Verrou avec Valgrind ou l'analyse synchrone pour détecter dynamiquement les branchements instables de CADNA. Chaque outil possède des forces et des faiblesses et nous pensons qu'il est judicieux de mutualiser les efforts pour que chaque outil tire parti des avantages des outils complémentaires.

À ce titre, le projet Interflop s'inscrit dans cet effort en proposant une interface commune entre frontends (modèles d'exécution) et backends (modèles de calcul) afin de proposer une interopérabilité entre les outils. Le projet Interflop regroupe ANEO, CEA, EDF, Intel, Sorbonne Université, TriScale Innov, UPVD, et UVSQ. Un premier prototype interfaçant les outils Verrou et Verificarlo est en phase de finalisation à l'heure où sont écrites ces lignes. Le projet est disponible sur github : `github.com/interflop`.

Acronymes

ASCII American Standard Code for Information Interchange. 90
ASIC Application-Specific Integrated Circuit. 28, 29

BLAS Basic Linear Algebra Subprograms. 149, *Glossary :* BLAS

CADNA Control of Accuracy and Debugging for Numerical Applications. 11, 12
CESTAC Contrôle et Estimation Stochastique des Arrondis de Calculs. 11, *Glossary :* CESTAC
CG Conjugate Gradient. 134, 141
CPU Central Processing Unit. 6, 28, 29, 60, 110, 153
CSV Comma Separated Values. 78, 91, 92

DAZ Denormals As Zero. 145
DFT Density Functional Theory. 96, *Glossary :* DFT
DPCG Deflated Preconditionned Conjugate Gradient. 133–138, 140, 141, 145
DSA Discrete Stochastic Arithmetic. 11

EFT Error Free Transformation. 105, *Glossary :* EFT

FORTRAN FORmula TRANSlator. 96, 97
FPGA FieldProgrammable Gate Array. 6, 29
FTZ Flush To Zero. 145

GCC GNU C Compiler. 52, 60, 61, 153
GDB GNU DeBugger. 143
GNU GNU's Not UNIX. 12, 79, 87, *Glossary :* GNU
GPGPU General Purpose Graphics Processing Unit. 28
GPU Graphics Processing Unit. 6, 28, 153

HDD Hard Drive Disk. 110
HPC High Performance Computing. 2, 6, 7, 15, 16, 28, 143, *Glossary :* HPC
HPCG High-Performance Conjugate Gradient. *Glossary :* HPCG

IA-32 Intel Architecture,32-bit. *Glossary :* IA-32
IDE Integrated Development Environment. 94

LLVM Low Level Virtual Machine. 9, 12–14, 51, 52, 80, 81, 83, 85, 86, 146, 154
LTO Link Time Optimization. 146

MAC Multiply ACcumulator. 28, 29
MCA Monte Carlo Arithmetic. 11, 12, 15, 76, 77, *Glossary :* MCA
MPI Message Passing Interface. 15, 142, 145

NaN Not-a-Number. *Glossary :* NaN

PB Precision Bounding. 45, *Glossary :* PB

RAM Random Access Memory. 110
RR Random Rounding. 45, *Glossary :* RR

SIMD Single Instruction Multiple Data. 28, 36, 156
SMT Satisfiability Modulo Theory. 8, 9, *Glossary :* SMT
SSA Static Single Assignment. 80
SSD Solid State Drive. 110

TLS Thread Local Storage. 57
TPU Tensor Processor Unit. 2, 28, 29

ulp Unit in the Last Place. 30, *Glossary :* ulp

Glossaire

`etotal` Nom de la variable dans l'application ABINIT contenant l'énergie totale (exprimée en Hartree) du système physique étudié. C'est le résultat d'intérêt majeur calculé par ABINIT.. 97, 100, 101, 103, 105

Agnostique Qualificatif qui exprime l'indépendance vis à vis d'un contexte. Par exemple, un compilateur est dit architecture agnostique s'il peut compiler du code sur n'importe quelle architecture. . 79, 80

AND ET logique avec pour table de vérité **??**. 62

Annulation Perte de chiffres significatifs lors d'une soustraction entre deux nombres flottants. On parle d'annulation catastrophique lorsque les deux nombre sont très proches et font remonter des erreurs d'arrondi des derniers chiffres de la mantisse. Voir définition 2.4.2. 11, 12, 26, 45–47, 73, 121, 155

Backend Implémentation d'un modèle de calcul dans Verificarlo. Chaque backend doit implémenter un certain nombre d'opérations arithmétiques et logiques pour s'interfacer avec Verificarlo. 14, 15, 50, 51, 53, 57, 60, 62, 64, 66, 67, 73, 74, 100, 101, 104, 105, 111–115, 122, 128, 130, 140, 143–146

Backtrace Contexte d'appels d'une fonction. Présenté sous forme de pile d'appels en empilant la fonction appelante à chaque appel.. 87, 91–93

Backward error Pour un schéma numérique f donné, l'erreur inverse est la distance entre le problème exact x^* et sa discrétisation en précision finie x. Si y^* est la solution exacte, alors l'erreur inverse est le δ_x tel que $f(x + \delta_x) = y^*$. 7

Basic block Un *basic block* est une suite d'instructions ayant les propriétés suivantes : un seul point d'entrée et un seul point de sortie ce qui impose que les instructions de branchements soient à la dernière instruction du *basic block*. Les instructions à l'intérieur d'un *basic block* sont donc toujours exécutées dans le même ordre. 81–83

Benchmark Test de performance servant à mesurer la performance d'un code de calcul ou une architecture. Différentes métriques existent : Flops, énergie, mémoire, entrés/sorties, ... 3, 153, 154

Bit Valeur binaire avec deux valeurs possibles {0,1}. 90, 98

Bit-flip Inversement incontrôlé d'un bit en matériel du à un phénomène physique aléatoire comme un rayon cosmique. . 110

BLAS Librairie d'algèbre linaire regroupant les opérations de base (produit scalaire, produit matrice-vecteur, produit matrice-matrice, ... 40

Branch and Bound La méthode de séparation (*branch*) et d'évaluation (*bound*) pour un problème d'optimisation consiste à construire un arbre de décision représentant les solutions possibles puis d'évaluer le coût d'une branche. Si l'on arrive à montrer que cette branche est trop coûteuse, on coupe la branche ce qui réduit le nombre de possibilités à évaluer. 10

CESTAC Arithmétique stochastique créée par J. Vignes en 1974 qui propose d'introduire un arrondi aléatoire sur les résultats des calculs flottants pour modéliser les erreurs d'arrondi.. 11, 41, 42, 149

Clang Compilateur frontend pour les langages `C`,`C++`,`Ojective-C` et `Ojective-C++` qui utilise l'infrastructure LLVM comme back end. 33, 51, 84, 86, 128

Convolution Un produit de convolution entre deux fonctions est le produit qui $\forall f, g \in L^2, (f * g)(x) = \int_{-\infty}^{+\infty} f(x-t)g(t)dt$. 156

Couverture de code La couverture de code mesure l'ensemble des flots d'exécution qu'un programme peut prendre. La couverture de code sert notamment à démontrer qu'un chemin d'exécution ne sera jamais pris (code mort) ou à vérifier qu'une batterie de tests teste bien l'ensemble des exécutions possibles d'un code.. 88, 100, 153

Criticité La criticité est la détermination et la hiérarchisation du degré d'importance et de la disponibilité d'un processus informatique. (Wikipédia). 97

Désassembler Le désassemblage consiste à décoder les instructions assembleurs binaires en mnéomnique, c'est-à-dire des instructions assembleurs lisibles par un humain.. 79

Deep learning Les méthodes d'apprentissage par réseaux de neurones profonds sont des algorithmes combinant plusieurs couches de neurones. Ils surpassent les méthodes d'apprentissages plus conventionnelles d'où leur intérêt croissant.. 6, 7, 26, 110

DFT La théorie de la fonctionnelle de la densité est une théorie physique qui permet l'étude de la densité électronique, c'est-à-dire la probabilité de trouver un électron dans un point de l'espace. . 96, 149

Diagramme quantile-quantile Le diagramme quantile-quantile ou diagramme Q-Q est un graphe permettant de comparer la position de certains quantile dans la population observée avec leur position dans la population théorique.. 130

Divide & Conquer Méthode algorithmique qui consiste à résoudre un problème en le divisant en sous-problèmes plus faciles à résoudre et combiner les solutions pour obtenir la solution du problème initial. 116

Dénormalisé Un dénormalisé est un nombre flottant dont le premier chiffre significatif est 0. Depuis le norme IEEE 754-2008 on parle également de sous-normal. Voir définition 2.1.2. 13, 60, 61, 113, 145

EFT Ensemble d'algorithmes permettant de calculer exactement avec des nombres flottants . 40, 103, 105, 108, 149

Erreur relative Distance entre deux nombres, normalisé par l'un des deux. Voir définition 2.4.2. 153

Exposant Si $x = m \times \beta^e$ alors m est sa mantisse, e son exposant et β sa base. . 13, 154

Flops Unité de mesure pour la performance d'un programme qui donne le nombre de calculs flottants par seconde. 3, 151

Folding Fonction d'ordre supérieur utilisée dans le paradigme de la programmation fonctionnelle pour agréger les éléments d'un type récursif . 37, 38

Fork-Join Modèle de programmation parallèle de type maître-esclave où l'exécution séquentielle du maître est rythmé par des sections parallèles. Celles-ci débutent par le *fork* où plusieurs tâches esclaves effectuent du travail en parallèle puis s'achèvent par le *join* où le maître attend la fin de chacun des esclaves pour reprendre le cours de l'exécution. 155

Forward error Pour un schéma numérique f donné, l'erreur directe est la distance entre une solution y calculée avec une précision finie et la solution exacte y^* calculée par f. L'erreur directe est la distance : $\delta_y = |y - y^*|$. 7

Gcov Gcov est un outil d'analyse de couverture de code et de profilage qui fait partie de GCC.. 12, 88

GNU GNU est un système d'exploitation libre créé en 1983 par Richard Stallman, maintenu par le projet GNU. Il reprend les concepts et le fonctionnement d'UNIX (Wikipedia). 52, 79, 149

Grandeur scalaire Grandeur dont la valeur ne dépend que d'un point, n'est représenté que par un seul nombre, à la différence d'une grandeur vectorielle. Formellement, un scalaire est un tenseur d'ordre 0. 70

Grandeur vectorielle Un vecteur est un objet généralisant plusieurs notions provenant de la géométrie (couples de points, translations, etc.), de l'algèbre ('solution" d'un système d'équations à plusieurs inconnues), ou de la physique (forces, vitesses, accélérations, etc.). Formellement, un vecteur est un tenseur d'ordre 1. (Wikipedia). 70, 153

Guards digits Chiffres de précision supplémentaires permettant d'accroître la précision du résultat d'une soustraction flottante. Par exemple, si $x, y \in \mathbb{F}_{\beta,p}$ et $x - y$ est calculée en utilisant une précision de $p + 1$ chiffres alors l'erreur relative est inférieur à 2ulp (cf. Théorème 2 [71, p.10]). 45

Hétérogénéité Qualificatif d'une architecture regroupant plusieurs unités de calculs de nature différentes. Par exemple, un cluster constitué de CPUs et de GPUs sera dit hétérogène par opposition à une architecture uniquement constituée de CPUs. 3

Hartree L'énergie de Hartree (Ha) est la valeur absolue de l'énergie potentielle électrique de l'atome d'hydrogène au repos. Un Hartree équivaut à $Ha = 4,35974434 \times 10^{-18} Joules$. 97, 98, 151

HPC Le Calcul à Haute Performance regroupe l'ensemble des activités ayant pour objectifs la maximisation des performances d'un code donné pour une architecture visée. Les performances recherchées peuvent être : le temps, la mémoire ou l'énergie.. 2, 3, 6, 7, 50, 73, 75, 96, 110, 143, 147, 154

HPCG Le High-Performance Conjugate Gradient benchmark est un benchmark pour les supercalculateurs se voulant plus proche des applications réellement exécutées, en terme de calculs et d'accès mémoire, que ceux du LINPACK . 3, 149

IA-32 Version 32-bit du jeu d'instructions assembleur x86 conçu par Intel. Aussi appelé i386. 33, 35, 150

Inlining Technique qui consiste à remplacer un appel de fonction par le corps de la fonction ce qui permet d'augmenter les performances pour des fonctions avec peu d'instructions. 33, 35

Interpretation Dans la logique du premier ordre, une interprétation est une fonction qui associe une valeur de vérité à chaque variable propositionnelle. 156

Interprocédurale L'analyse interprocédurale étudie les interactions entre procédures (fonctions) d'un code. Elle se distingue de l'analyse intraprocédurale qui analyse à l'intérieur d'une fonction.. 80, 154

Intraprocédurale L'analyse intraprocédurale analyse l'intérieur d'une fonction. Elle se distingue de l'analyse interprocédurale qui analyse à l'extérieur d'une fonction.. 88, 154

intrinsèque Un intrinsèque est une fonction intégrée au compilateur. Ces intrinsèques sont notamment utilisées pour s'abstraire des instructions assembleurs spécifiques à une architecture comme un appel à des instructions vectorielles par exemple.. 83

Invocation Appel ou exécution d'une section spatiale numéroté suivant l'ordre d'apparition. Voir définition 3.3.2 . 15, 70, 75, 91, 92, 112, 115–118, 122, 128, 130

Linkage Phase d'édition des liens où les symboles définis dans les fichiers objets sont liés dans un même fichier (linkage statique) ou vers des fichiers séparés (linkage dynamique) résolues durant l'exécution. . 146

LINPACK benchmark pour le calcul HPC dédié à l'algèbre linéaire. `netlib.org/benchmark/hpl/`. 3, 153

llvm-cov Outil de la suite LLVM qui permet de faire de la couverture de code.. 88

Logique du premier ordre Une formule logique de premier ordre est une formule logique faisant uniquement intervenir des variables, des fonctions, des connecteurs logiques, des quantificateurs et des prédicats. 9, 154, 156

Méthode itérative En analyse numérique, les méthodes itératives concernent l'ensemble des schémas itératifs dont le nombre d'étapes nécessaires à la terminaison du calcul n'est pas connu à l'avance contrairement aux méthodes dites directes.. 110, 118, 119

Mantisse Si $x = m \times \beta^e$ alors m est sa mantisse, e son exposant et β sa base. . 13, 60, 61, 151, 152

MapReduce Modèle de programmation distribuée basé sur les concepts de `map` et `reduce` empruntés à la programmation fonctionnelle. Le `map` distribue le travail aux unités de calcul et le `reduce` remonte l'information calculée. 37

MAQAO Modular Assembler Quality Analyzer and Optimizer. Outil d'analyse de performance des codes au niveau binaire. 57

MCA Arithmétique stochastique théorisée par Stott Parker en 1997. Modélise les valeurs flottantes inexactes par une valeur exacte plus une variable aléatoire modélisant un bruit. Voir section 2.6.2.. 11, 46, 150, 155

Multi-échelle Un code de simulation multi-échelle inclut plusieurs échelles de résolution spatiale ou temporelle dans sa simulation. Par exemple, une simulation d'écoulement d'air autour d'un avion aura une résolution très fine autour des ailes pour capturer les comportements fins des turbulences et plus grossier là ou l'écoulement est plus laminaire. 4

Multi-physique Un code multi-physique modélise plusieurs disciplines physiques au sein d'une même simulation. Par exemple, les vibrations d'une aile sont le fruit de l'interaction entre les déformations de l'aile (contraintes mécaniques) et l'écoulement de l'air (mécanique des fluides). Le couplage entre deux physiques est parfois nécessaire lorsque l'interaction n'est pas négligeable. 4

NaN Nombre flottant n'ayant pas de sens mathématique dans les nombres réels $\mathbb{R}$. Par exemple $\sqrt{-1}$ ou $0/0$. 26, 113, 150

Obfusqué L'obfusquage est une technique de sécurité informatique qui consiste à rendre un code illisible pour une personne externe si celle-ci n'a pas la clef pour le déchiffrer.. 79

Octet Un octet représente 8 bits.. 90

OpenMP Interface de programmation pour le calcul parallèle à mémoire partagée basé sur le modèle fork-join . 37

OR OU logique avec pour table de vérité **??**. 62

Ordre de récurrence L'ordre de récursion d'une suite est l'écart entre le plus petit et le plus grand terme de récurrence. Voir definition **??**. 116

Overflow Un overflow ou dépassement de capacité vers $\pm\infty$ survient lorsque la valeur absolue du nombre flottant est plus grande que le plus grand nombre représentable dans un format donné. 115

Padding Technique de bourrage consitant à ajouter des 0 pour aligner un mot memoire. 90

PB Mode de bruitage de MCA permettant de détecter les annulations. Voir section 2.6.1. 45–47, 150

Processus Programme en cours d'exécution sur une machine. Il dispose d'une mémoire qui lui est propre contrairement au thread. 71

Quantile Valeur qui divise un jeu de données en parts égales. Par exemple, l'ensemble $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ à pour 1er, 2nd et 3ème quartile respectivement 2,5 ,5 et 7,5.. 152

Race condition Situation de compétition ou deux threads essaient d'accéder à une ressource partagée. Sans mécanisme d'exclusion mutuelle, cette situation produit des bugs difficiles à reproduire. 156

Racine Solutions x de l'équation $f(x) = 0$ pour une fonction f donnée.. 118

RR Mode d'arrondi aléatoire de MCA. Voir section 2.6.1. 45, 46, 150

sémantique La sémantique d'un programme est le comportement attendu d'un programme, c'est-à-dire ce que le programme calcule réellement. Elle décrit tout les états possibles du programme.. 9

Satisfiabilité Une formule logique propositionnelle satisfiable est une formule dont une instance des variable rend la formule vraie. 9

Section spatiale Section de code contenant plusieurs unités spatiales. La taille d'une unité dépend de la résolution spatiale utilisée (fonction, ligne source, instructions, ...).. 154

Shadow memory Technique qui consiste à exécuter les opérations d'un programme en parallèle de celui ci dans une mémoire séparée. La shadow memory permet de manipuler une copie des données sans affecter l'exécution du programme originale. . 12, 13, 89, 114

Source à source Une transformation source à source est une transformation qui transforme un code source un autre code source. Le code source résultant peut être exprimé à l'aide d'un autre langage que celui de départ ou bien posséder une sémantique différente.. 10

Taint analysis La taint analysis est une technique issue de la sécurité informatique qui consiste à colorer des variables et propager les couleurs à travers l'exécution pour observer les influences. Une taint policy régie la façon dont sont propagées les couleurs.. 12, 89, 106

Thread Fil d'exécution ou processus léger. Les threads d'un même processus partagent la même mémoire. 57, 92, 155, 156

Thread-safe Désigne la propriété d'une fonction pouvant être exécutée simultanément par plusieurs threads sans situation de compétition (race condition). 57

Top500 Classement des 500 supercalculateurs les plus puissants de la planète. `top500.org`. 3

Type composé Type qui regroupe plusieurs champs comme le type `struct` en langage `C`. 11, 81

Type natif Type de base fourni par un langage de programmation comme les entiers, les nombres flottants, les caractères. 53, 60, 81, 114

Type récursif Type de donnée construit récursivement dont la définition fait appel à lui-même comme par exemple les structures d'arbre ou de liste. Exemple figure **??**. . 153

ULP Unité à la dernière place d'un nombre flottant mesurant la précision minimale d'un format flottant. Par exemple pour un format flottant $\mathbb{F}_p$ donné, deux nombres flottants consécutifs sont distant d'un $ulp = 2^{-p+1}$. Voir définition 2.4.3. 30, 43, 150, 153

UNIX Famille de systèmes d'exploitation multitâche et multi-utilisateur dérivé du Unix d'origine créé par AT&T. Il repose sur un interpréteur ou superviseur (le shell) et de nombreux petits utilitaires, accomplissant chacun une action spécifique, commutables entre eux (mécanisme de "redirection") et appelés depuis la ligne de commande (Wikipedia). 153

Unum Format de calcul flottant alternatif aux flottants issues de la norme IEEE-754.. 26

Valeur de vérité Valeurs qui disent si une formule est vrai ou fausse. Par exemple les booléens . 154

Valide Dans la logique du premier ordre une formule valide est une formule vrai pour toute interpretation de la formule.. 10

Vectorisation Technique de parallélisation de type SIMD consistant à calculer une même opération sur différentes données en même temps. 53

VNNI Instructions assembleurs du jeu d'instructions AVX-512 pour accélérer le calcul des réseaux de neurones à convolution. Ces instructions calcul la convolution entre deux vecteurs de binary16 dans un accumulateur binary32.. 8

XOR OU exclusif logique avec pour table de vérité **??**. 61–63, 65

Bibliographie

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow : A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200. ACM, 1998.

[3] Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J Higham, and Enrique S Quintana-Ortí. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation : Practice and Experience*, page e4460, 2017.

[4] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. Cascade Lake : Next generation Intel Xeon scalable processor. *IEEE Micro*, 39(2) :29–36, 2019.

[5] Jean-Claude Bajard, Dominique Michelucci, Jean-Michel Moreau, and Jean-Michel Muller. Introduction to the Special Issue "Real Numbers and Computers". In *The Journal of Universal Computer Science*, pages 436–438. Springer, 1996.

[6] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. In *ACM Sigplan Notices*, volume 48, pages 549–560. ACM, 2013.

[7] Joan Batlle, J Martı, Pere Ridao, and Josep Amat. A new FPGA/DSP-based parallel architecture for real-time image processing. *Real-Time Imaging*, 8(5) :345–356, 2002.

[8] Heiko Becker, Eva Darulova, Magnus O Myreen, and Zachary Tatlock. Icing : Supporting fast-math style optimizations in a verified compiler. In *International Conference on Computer Aided Verification*, pages 155–173. Springer, 2019.

[9] P Benard, G Lartigue, V Moureau, and R Mercier. Large-eddy simulation of the lean-premixed PRECCINSTA burner with wall heat loss. *Proceedings of the Combustion Institute*, 2018.

[10] Pierre Bénard, Ghislain Lartigue, Vincent Moureau, and Renaud Mercier. Large-eddy simulation of the lean-premixed PRECCINSTA burner with wall heat loss. *Proceedings of the Combustion Institute*, 37(4) :5233–5243, 2019.

[11] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Notices*, volume 47, pages 453–462. ACM, 2012.

[12] James Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 759–767. IEEE, 2005.

[13] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, Lyon, École normale supérieure (sciences), 2004.

[14] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution : a comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4) :423–456, 2013.

[15] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2) :135–163, 2015.

[16] Sylvie Boldo and Guillaume Melquiond. Flocq : A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.

[17] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[18] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[19] Bryan Buck and Jeffrey K Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4) :317–329, 2000.

[20] Christopher Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Gregory N Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine learning (ICML-05)*, pages 89–96, 2005.

[21] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience : 2014 update. *Supercomputing frontiers and innovations*, 1(1) :5–28, 2014.

[22] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. CERE : LLVM-based Codelet Extractor and REplayer for piecewise benchmarking and optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1) :6, 2015.

[23] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[24] Yohan Chatelain, Pablo De Oliveira Castro, Eric Petit, David Defour, Jordan Bieder, and Marc Torrent. Veritracer : Context-enriched tracer for floating-point arithmetic analysis. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 61–68. IEEE, 2018.

[25] Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. Automatic exploration of reduced floating-point representations in iterative methods. In *European Conference on Parallel Processing*, pages 481–494. Springer, 2019.

[26] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO : Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019.

[27] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. In *ACM Sigplan Notices*, volume 49, pages 43–52. ACM, 2014.

[28] Alexandre Joel Chorin. Numerical solution of the Navier-Stokes equations. *Mathematics of computation*, 22(104) :745–762, 1968.

[29] David V Chudnovsky and Gregory V Chudnovsky. Approximations and complex multiplication according to ramanujan. In *Pi : A Source Book*, pages 596–622. Springer, 2004.

[30] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2) :8–20, 2018.

[31] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT 5 SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[32] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Full-speed deterministic bit-accurate parallel floating-point summation on multi-and many-core architectures. *HAL-CCSD, Tech. Rep. hal-00949355*, 2014.

[33] IEEE Computer Society. Standards Committee and American National Standards Institute. *IEEE standard for binary floating-point arithmetic*, volume 754. IEEE, 1985.

[34] IEEE Standards Committee et al. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, page 517, 2008.

[35] Siegfried Cools, Emrullah Fatih Yetkin, Emmanuel Agullo, Luc Giraud, and Wim Vanroose. Analysis of rounding error accumulation in Conjugate Gradients to improve the maximal attainable accuracy of pipelined CG. Research Report RR-8849, Inria Bordeaux Sud-Ouest, January 2016.

[36] Martyn J Corden and David Kreitzer. Consistency of floating-point results using the Intel compiler or why doesn't my application always give the same answer. Technical report, Technical report, Intel Corporation, Software Solutions Group, 2009.

[37] Marius A Cornea-Hasegan, Roger A Golliver, and Peter Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*, pages 96–105. IEEE, 1999.

[38] Matthieu Courbariaux, Jean-Pierre David, and Yoshua Bengio. Low precision storage for deep learning. *arXiv preprint arXiv :1412.7024*, 2014.

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[40] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4) :451–490, 1991.

[41] Nasrine Damouche and Matthieu Martel. Mixed precision tuning with Salsa. In *PECCS*, pages 185–194, 2018.

[42] Eva Darulova, Einar Horn, and Saksham Sharma. Sound mixed-precision optimization with rewriting. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 208–219. IEEE, 2018.

[43] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Acm Sigplan Notices*, volume 49, pages 235–248. ACM, 2014.

[44] Florent De Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. Posits : the good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, page 6. ACM, 2019.

[45] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1318–1322. ACM, 2006.

[46] Luiz Henrique De Figueiredo and Jorge Stolfi. Affine arithmetic : concepts and applications. *Numerical Algorithms*, 37(1-4) :147–158, 2004.

[47] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[48] Jeffrey Dean and Sanjay Ghemawat. MapReduce : simplified data processing on large clusters. *Communications of the ACM*, 51(1) :107–113, 2008.

[49] David Defour. *Contribution au calcul sur GPU : considérations arithmétiques et architecturales*. Habilitation à Diriger des Recherches, Université de Perpignan, 2014.

[50] David Defour. FP-ANR : A representation format to handle floating-point cancellation at run-time. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 76–83. IEEE, 2018.

[51] Theodorus Jozef Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3) :224–242, 1971.

[52] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 53–69. Springer, 2009.

[53] James Demmel and Hong Diep Nguyen. Fast reproducible floating-point summation. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 163–172. IEEE, 2013.

[54] James Demmel and Hong Diep Nguyen. Numerical reproducibility and accuracy at exascale. In *2013 IEEE 21st Symposium on Computer Arithmetic*, pages 235–237. IEEE, 2013.

[55] J.E Dendy. Black box multigrid. *Journal of Computational Physics*, 48(3) :366 – 386, 1982.

[56] C. Denis, P. d. O. Castro, and E. Petit. Verificarlo : Checking floating point accuracy through monte carlo arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 55–62, July 2016.

[57] Jérémie Detrey. *Arithmétiques réelles sur FPGA : virgule fixe, virgule flottante et système logarithmique*. PhD thesis, Lyon, École normale supérieure (sciences), 2007.

[58] Jérémie Detrey, Florent de Dinechin, and ENS-Lyon LIP. Opérateurs trigonométriques en virgule flottante sur FPGA. In *RenPar*, volume 17, pages 96–105, 2006.

[59] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, William Jalby, et al. Maqao : Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, volume 200, 2005.

[60] J Dongarra, H Meuer, and Erich Strohmaier. Top 500 supercomputers. *website, November*, 2008.

[61] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[62] Guy Even, Peter-M Seidel, and Warren E Ferguson. A parametric error analysis of Goldschmidt's division algorithm. *Journal of Computer and System Sciences*, 70(1) :118–139, 2005.

[63] François Févotte and Bruno Lathuiliere. VERROU : a CESTAC evaluation without recompilation. *SCAN 2016*, page 47, 2016.

[64] George E Forsythe. Reprint of a note on rounding-off errors. *SIAM Review*, 1(1) :66, 1959.

[65] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard, Version 3.1*. High Performance Computing Center Stuttgart (HLRS), 2015.

[66] Robert Fourer, David M Gay, and Brian W Kernighan. AMPL : A modeling language for mathematical programming. 1993.

[67] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2) :13, 2007.

[68] Michael Frechtling and Philip HW Leong. Mcalib : Measuring sensitivity to rounding error with monte carlo programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2) :5, 2015.

[69] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[70] GNU. GNU Debugger. gnu.org/software/gdb, 2019.

[71] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1) :5–48, 1991.

[72] Max Goldstein. Significance arithmetic on a digital computer. *Communications of the ACM*, 6(3) :111–117, 1963.

[73] Xavier Gonze, François Jollet, et al. Recent developments in the ABINIT software package. *Computer Physics Communications*, 205 :106–131, 2016.

[74] Eric Goubault. Static analysis by abstract interpretation of numerical programs and systems, and fluctuat. In *International Static Analysis Symposium*, pages 1–3. Springer, 2013.

[75] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuiliere. PROMISE : floating-point precision tuning with stochastic arithmetic. In *Proceedings of the 17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*, pages 98–99, 2016.

[76] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. Auto-tuning for floating-point precision with discrete stochastic arithmetic. *Journal of Computational Science*, 36 :101017, 2019.

[77] Stef Graillat, Valérie Ménissier-Morain, et al. Error-free transformations in real and complex floating point arithmetic. In *Proceedings of the International Symposium on Nonlinear Theory and its Applications*, pages 341–344. Citeseer, 2007.

[78] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755 : ADOL-C : a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2) :131–167, 1996.

[79] William Gropp, William D Gropp, Argonne Distinguished Fellow Emeritus Ewing Lusk, Ewing Lusk, and Anthony Skjellum. *Using MPI : portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[80] William Gropp and Ewing Lusk. User's guide for mpich, a portable implementation of MPI, 1996.

[81] Posit Working Group. Posit standard documentation. In *Posit Standard Documentation*, 2018.

[82] John L Gustafson and Isaac T Yonemoto. Beating floating point at its own game : Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2) :71–86, 2017.

[83] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU Tensor Cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 47 :1–47 :11, Piscataway, NJ, USA, 2018. IEEE Press.

[84] John Harrison. Floating point verification in HOL Light : the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/tang.html`.

[85] John Harrison. A machine-checked theory of floating point arithmetic. In *International Conference on Theorem Proving in Higher Order Logics*, pages 113–130. Springer, 1999.

[86] John Harrison. HOL light : An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[87] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool : Principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3) :20, 2013.

[88] Martin C Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with FPGA-based computing. *Computer*, 40(3) :50–57, 2007.

[89] Nicholas J Higham. *Accuracy and stability of numerical algorithms*, volume 80. Siam, 2002.

[90] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 63–68. IEEE, 2017.

[91] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant : a tutorial : version 7.2. 2002.

[92] Roman Iakymchuk, Sylvain Collange, David Defour, and Stef Graillat. ExBLAS : Reproducible and accurate BLAS library. 2015.

[93] Intel. x87 and SSE FP assits, 2008.

[94] Intel Corporation. *Intel64 and IA-32 Architectures Software Developer's Manual-Volume 1 : Basic Architecture*, 2019.

[95] International Electrotechnical Commission (IEC), International Federation of Clinical Chemistry and Laboratory Medicine (IFCC), International Laboratory Accreditation Cooperation (ILAC), International Union of Pure and Applied Chemistry (IUPAC), and International Union of Pure and Applied Physics (IUPAP). *ISO and OIML : The international vocabulary of metrology-basic and general concepts and associated terms (VIM)*. Bureau International des Poids et Mesures (BIPM), third edition, 2012.

[96] Arnault Ioualalen and Matthieu Martel. Sardana : an automatic tool for numerical accuracy optimization. 2012.

[97] Fabienne Jézéquel and Jean-Marie Chesneaux. CADNA : a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12) :933–955, 2008.

[98] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[99] William Kahan. Pracniques : further remarks on reducing truncation errors. *Communications of the ACM*, 8(1) :40, 1965.

[100] William Kahan. A logarithm too clever by half, 2004.

[101] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of bfloat16 for deep learning training. *arXiv preprint arXiv :1905.12322*, 2019.

[102] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and C-L Wang. Heterogeneous computing : Challenges and opportunities. *Computer*, 26(6) :18–27, 1993.

[103] Donald E Knuth. *Art of computer programming, volume 2 : Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[104] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint : An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*, pages 1742–1752, 2017.

[105] EV Krishnamurthy. On optimal ierative schemes for high-speed division. *IEEE Transactions on Computers*, 100(3) :227–231, 1970.

[106] Ulrich Kulisch and Van Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91(3) :307–313, 2011.

[107] HT Kung. Let's design algorithms for VLSI systems. 1979.

[108] Michael O Lam, Jeffrey K Hollingsworth, Bronis R de Supinski, and Matthew P LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378. ACM, 2013.

[109] Michael O Lam, Jeffrey K Hollingsworth, and GW Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3) :146–155, 2013.

[110] Peter Lancaster. Error analysis for the Newton-Raphson method. *Numerische Mathematik*, 9(1) :55–68, 1966.

[111] Philippe Langlois, Rafife Nheili, and Christophe Denis. Recovering numerical reproducibility in hydrodynamic simulations. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 63–70. IEEE, 2016.

[112] G Lartigue, U Meier, and C Bérat. Experimental and numerical investigation of self-excited combustion oscillations in a scaled gas turbine combustor. *Applied thermal engineering*, 24(11-12) :1583–1592, 2004.

[113] Chris Lattner and Vikram Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[114] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. 1977.

[115] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. Make it real : effective floating-point reasoning via exact arithmetic. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 117. European Design and Automation Association, 2014.

[116] Xavier Leroy et al. The Compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 53, 2012.

[117] Google LLC. Using bfloat16 with tensorflow models. https://cloud.google.com/tpu/docs/bfloat16, 2019.

[118] LLVM. DragonEgg - using as a GCC backend, 2014.

[119] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[120] Commisseriat l'énergie atomique et aux énergies alternatives (CEA). Manual europlexus user's. *URL http ://europlexus. jrc. ec. europa. eu/public/manual_ html/index. html*, 2016.

[121] Victor Magron, George Constantinides, and Alastair Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Transactions on Mathematical Software (TOMS)*, 43(4) :34, 2017.

[122] Mathias Malandain. *Massively parallel simulation of low-Mach number turbulent flows*. Theses, INSA de Rouen, January 2013.

[123] Mathias Malandain, Nicolas Maheu, and Vincent Moureau. Optimization of the deflated Conjugate Gradient algorithm for the solving of elliptic equations on massively parallel machines. *Journal of Computational Physics*, 238 :32 – 47, 2013.

[124] Naraig Manjikian and Tarek Abdelrahman. Array data layout for the reduction of cache conflicts. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, pages 1–8, 1995.

[125] Peter Markstein. Software division and square root using Goldschmidt's algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers (RNC'6)*, volume 123, pages 146–157, 2004.

[126] Clive Max Maxfield. Chapter 17 - Application-Specific Integrated Circuits (asics). In Clive Max Maxfield, editor, *Bebop to the Boolean Boogie (Third Edition)*, pages 235 – 249. Newnes, Boston, third edition edition, 2009.

[127] Harshitha Menon, Michael Lam, D Kuffour, Markus Schordan, S Llyod, Kathryn Mohror, and Jeff Hittinger. ADAPT : Algorithmic differentiation for floating-point precision tuning. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[128] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv :1710.03740*, 2017.

[129] Sparsh Mittal and Jeffrey S Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4) :69, 2015.

[130] John Monaghan, Luc Trouche, and Jonathan M Borwein. *Tools and mathematics*. Springer, 2016.

[131] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3) :12, 2008.

[132] Ramon E Moore. *Methods and applications of interval analysis*. SIAM, 1979.

[133] V Moureau, P Domingo, and L Vervisch. Design of a massively parallel CFD code for complex geometries. *Comptes Rendus Mécanique*, (339) :141–148, 2011.

[134] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2010.

[135] Zainalabedin Navabi. *VHDL : Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[136] Nedialko S Nedialkov, Vladik Kreinovich, and Scott A Starks. Interval arithmetic, affine arithmetic, taylor series methods : why, what next ? *Numerical Algorithms*, 37(1-4) :325–336, 2004.

[137] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[138] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[139] Roy A Nicolaides. Deflation of conjugate gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 24(2) :355–365, 1987.

[140] Andres Nötzli and Fraser Brown. LifeJacket : verifying precise floating-point optimizations in LLVM. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 24–29. ACM, 2016.

[141] Takeshi Ogita, Siegfried M Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6) :1955–1988, 2005.

[142] Paul Osmialowski. How the flang frontend works. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC 2017)*, 2017.

[143] Gabriele Paganelli and Wolfgang Ahrendt. Verifying (in-) stability in floating-point programs by increasing precision, using SMT solving. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 209–216. IEEE, 2013.

[144] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *ACM SIGPLAN Notices*, volume 50, pages 1–11. ACM, 2015.

[145] D Stott Parker, Brad Pierce, and Paul R Eggert. Monte carlo arithmetic : how to gamble with floating point and win. *Computing in Science & Engineering*, 2(4) :58, 2000.

[146] Douglass Stott Parker. *Monte Carlo Arithmetic : exploiting randomness in floating-point arithmetic*. University of California (Los Angeles). Computer Science Department, 1997.

[147] Kristin Persson. Materials data on BaTiO3 (sg :99) by Materials Project, 7 2014. An optional note.

[148] Eric Petit. *Vers un partitionnement automatique d'applications en codelets spéculatifs pour les systèmes hétérogènes à mémoires distribuées*. PhD thesis, 2009.

[149] C. D. Pierce and P. Moin. Progress-variable approach for large-eddy simulation of non-premixed turbulent combustion. *Journal of Fluid Mechanics*, 504 :73–97, 2004.

[150] Mihail Popov, Chadi Akel, William Jalby, and Pablo de Oliveira Castro. Piecewise holistic autotuning of compiler and runtime parameters. In *European Conference on Parallel Processing*, pages 238–250. Springer, 2016.

[151] Jon Porter. Google employee calculates pi to record 31 trillion digits : But remember, only 40 or so of them are actually useful. Web site, March 2019.

[152] Michael M Resch and Edgar Gabriel. Supercomputers in grids. In *Cloud, Grid and High Performance Computing : Emerging Applications*, pages 1–9. IGI Global, 2011.

[153] Nathalie Revol. Introduction à l'arithmétique par intervalles. Rapport de recherche RR-4297, INRIA, 2001.

[154] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1074–1085. ACM, 2016.

[155] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. Precimonious : Tuning assistant for floating-point precision. In *SC'13 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[156] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error with herbgrind. *arXiv preprint arXiv :1705.10416*, 2017.

[157] Gilbert Saporta. *Probabilités, analyse des données et statistique*. Editions Technip, 2006.

[158] Geof Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H Ahn. FLiT : Cross-platform floating-point result-consistency tester and workload. In *2017 IEEE international symposium on workload characterization (IISWC)*, pages 229–238. IEEE, 2017.

[159] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Notices*, volume 49, pages 53–64. ACM, 2014.

[160] Kirk Schloegel, George Karypis, and Vipin Kumar. *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

[161] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[162] Ariel Shamir. A survey on mesh segmentation techniques. 27(6) :1539–1556, 2008.

[163] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4) :591–611, 1965.

[164] James E Smith. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, volume 10, pages 112–119. IEEE Computer Society Press, 1982.

[165] Steven W Smith et al. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.

[166] Devan Sohier, Pablo De Oliveira Castro, François Févotte, Bruno Lathuilière, Eric Petit, and Olivier Jamond. Confidence intervals for stochastic arithmetic. *arXiv preprint arXiv :1807.09655*, 2018.

[167] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1) :20, 2018.

[168] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S Păsăreanu. CORAL : solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*, pages 359–374. Springer, 2011.

[169] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The ARM scalable vector extension. *IEEE Micro*, 37(2) :26–39, 2017.

[170] Bjorn Stevens and Sandrine Bony. What are climate models missing ? *Science*, 340(6136) :1053–1054, 2013.

[171] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. A transprecision floating-point platform for ultra-low power computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1051–1056. IEEE, 2018.

[172] Alexandre F Tenca and Milos D Ercegovac. A variable long-precision arithmetic unit design for reconfigurable coprocessor architectures. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No. 98TB100251)*, pages 216–225. IEEE, 1998.

[173] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[174] Arnaud Tisserand, Pierre Marchal, and Christian Piguet. An on-line arithmetic based FPGA for low-power custom computing. In *International Workshop on Field Programmable Logic and Applications*, pages 264–273. Springer, 1999.

[175] Jean-Yves Tissot. *Sur la décomposition ANOVA et l'estimation des indices de Sobol'. Application à un modèle d'écosystème marin.* PhD thesis, Grenoble, 2012.

[176] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Muñoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 516–537. Springer, 2018.

[177] Honey Durga Tiwari, Ganzorig Gankhuyag, Chan Mo Kim, and Yong Beom Cho. Multiplier design based on ancient Indian Vedic Mathematics. In *2008 International SoC Design Conference*, volume 2, pages II–65. IEEE, 2008.

[178] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3) :384 – 393, 1975.

[179] Michael L Van De Vanter, DE Post, and Mary E Zosel. HPC needs a tool strategy. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 55–59. ACM, 2005.

[180] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. Towards automatic significance analysis for approximate computing. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 182–193. IEEE, 2016.

[181] J Vignes and M La Porte. Error analysis in computing, information processing 74, 1974.

[182] Nathan Whitehead and Alex Fit-Florea. Precision & performance : Floating point and IEEE-754 compliance for nvidia GPUs. *rn (A+ B)*, 21(1) :18749–19424, 2011.

[183] James Hardy Wilkinson. *Rounding errors in algebraic processes.* Courier Corporation, 1994.

[184] Tjalling J Ypma. Historical development of the Newton–Raphson method. *SIAM review*, 37(4) :531–551, 1995.

[185] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why ? In *ACM SIGSOFT Software engineering notes*, volume 24, pages 253–267. Springer-Verlag, 1999.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Outils de débugage et d'optimisation des calculs flottants dans le contexte HPC

Mots clés : HPC, virgule flottante, débugage, optimisation

Résumé : Le Calcul Haute Performance (HPC) est un écosystème dynamique où architectures et codes de calcul scientifiques sont en co-évolution permanente (parallélisme, accélérateurs spécialisés, nouvelles mémoires). Ce dynamisme impose aux développeurs d'adapter leur logiciel régulièrement pour exploiter au mieux tous les nouveaux ressorts technologiques. En cela, les approches de co-design consistant à développer simultanément le logiciel et le matériel sont une voie intéressante. Néanmoins, les efforts de co-design dans le HPC ont surtout été concentrés sur la performance des applications en laissant un peu de côté l'objectif de qualité numérique. Or celle ci est de plus en plus difficile à maintenir d'une génération de supercalculateur à l'autre en raison de la complexité croissante des architectures et des modèles de programmation parallèles. A cela s'ajoute de nouveaux formats de calcul flottants (bfloat16, binary16) qu'il faut réussir à exploiter lors du processus de modernisation.
Ces constats posent deux problématiques : 1) Comment vérifier la qualité numérique des codes lors du processus de modernisation ? Cela nécessite des outils qui permettent, à la fois d'identifier rapidement des sources d'erreurs numériques mais qui doivent également être simple d'utilisation pour des utilisateurs non-experts. 2) Comment tirer parti des nouvelles possibilités qu'offre le matériel ?
Les possibilités d'applications sont nombreuses et amènent donc à un espace considérable de solutions possibles. Les solutions trouvées sont le résultat d'un compromis entre performance de l'application et qualité numérique des calculs mais également reproductibilité des résultats.
Dans cette thèse, nous avons contribué au logiciel Verificarlo qui aide à la détection d'erreurs numériques en injectant divers modèles de bruit dans les calculs flottants. Plus précisément, nous avons développé une approche permettant d'étudier l'évolution des erreurs numérique au cours du temps. Cet outil est basé sur la génération de traces numériques qui permettent de suivre la qualité numérique des variables au cours du temps. Ces traces sont enrichies par des informations de contexte récupérées lors de la compilation puis peuvent être ensuite visualisées de manière élégante.
Nous avons également contribué à VPREC, un modèle de calcul simulant des formats de taille variable. Cet outil a été utilisé pour répondre au problème d'optimisation de formats dans les schémas itératifs. L'optimisation proposée est temporelle puisqu'elle optimise la précision de calcul pour chaque pas de temps. Enfin, une contrainte majeure dans l'élaboration d'outils pour le HPC est la mise à l'échelle. En effet, la taille des codes et la quantité de calcul mis en jeux accroissent drastiquement la complexité des analyses et limitent les approches conventionnelles. Nous avons démontré que les techniques développés dans cette thèse sont applicables sur des codes industriels puisqu'ils ont permis de, premièrement, détecter et corriger une erreur numérique dans le code ABINIT (code ab initio de chimie quantique développé par le CEA et al.). Secondement, ces outils ont permis de réduire la précision de calcul de YALES2 (code de mécanique des fluides développé par le CORIA) et améliorer les performance en réduisant le volumes des communications de 28% et accélérer jusqu'à $1,30$ fois l'exécution.

Title : Tools for debugging and optimizing floating-point computations in HPC

Keywords : HPC, floating point, debugging, optimization

Abstract : High Performance Computing (HPC) is a dynamic ecosystem where scientific computing architectures and codes are in permanent co-evolution (parallelism, specialized accelerators, new memories). This dynamism requires developers to adapt their software regularly to exploit all the new technological innovations. For this purpose, co-design approaches consisting of simultaneously developing software and hardware are an interesting approach. Nevertheless, co-design efforts have mainly focused on application performance at the cost of numerical quality. However, this is becoming increasingly difficult to maintain from one generation of supercomputer to the next. In addition, there are new computation formats (bfloat16, binary16) performance factor that have to be used during the modernization process.
These findings raise two issues: 1) How to check the digital quality of codes during the modernization process? This requires tools that allow both to quickly identify sources of numerical errors and to be user-friendly for non-expert users. 2) How can we take advantage of the new possibilities offered by the equipment? The applications possibilities are manifold and therefore lead to a considerable space of possible solutions. The solutions found are the result of a compromise between the performance of the application and the numerical quality of the computations, but also the reproducibility of the results.
In this thesis, we contributed to the Verificarlo software that helps to detect numerical errors by injecting various noise models into floating computations. More precisely, we have developed an approach to study the evolution of numerical errors over time. This tool is based on the generation of numerical traces that allow the numerical quality of the variables to be tracked over time. These traces are enriched by context information retrieved during compilation and can then be viewed in an elegant way.
We also contributed to VPREC, a calculation model simulating formats of varying sizes. This tool has been used to address the problem of format optimization in iterative schemes. The proposed optimization is temporal since it optimizes the computation precision for each time step.
Finally, a major constraint in the development of tools for HPC is the scaling up. Indeed, the size of the codes and the number of computations involved drastically increase the complexity of the analyses and limit conventional approaches. We have demonstrated that the techniques developed in this thesis are applicable to industrial codes since they have made it possible, first, to detect and correct a numerical error in the ABINIT code (ab initio code for quantum chemistry developed by the CEA et al.). Secondly, these tools have reduced the computation accuracy of YALES2 (fluid mechanics code developed by CORIA) and improved performance by reducing communication volumes by 28% and accelerating execution up to 1.30 times.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France