HAL open science

Auto-tuning de précision et validation numérique

Quentin Ferro

▶ **To cite this version:**

Quentin Ferro. Auto-tuning de précision et validation numérique. Neural and Evolutionary Computing [cs.NE]. Sorbonne Université, 2024. English. NNT : 2024SORUS412 . tel-04910717

HAL Id: tel-04910717

https://theses.hal.science/tel-04910717v1

Submitted on 24 Jan 2025

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

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

Thèse présentée pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Specialité
Informatique

École Doctorale
Informatique, Télécommunications et Électronique (Paris)

Auto-tuning de précision et validation numérique

Présentée par
Quentin Ferro

Soutenue le **16 Octobre 2024**,
devant un jury composé de :

Daniel Menard, Professeur, IETR, INSA Rennes	*Rapporteur*
Guillaume Revy, Maître de Conférences - HDR, LIRMM, Université de Perpignan	*Rapporteur*
Pierre Fortin, Professeur, CRIStAL, Université de Lille	*Président du Jury*
Stef Graillat, Professeur, LIP6, Sorbonne Université	*Co-Directeur*
Thibault Hilaire, Maître de Conférences - HDR, LIP6, Sorbonne Université	*Encadrant*
Fabienne Jézéquel, Maître de Conférences - HDR, LIP6, Univ. Paris-Panthéon-Assas	*Directrice*

// Résumé

Titre : *Auto-tuning de précision et validation numérique*
Mots clés : Nombres flottants, Précision, Auto-tuning

Résumé :

Cette thèse se concentre sur l'auto-ajustement de la précision des nombres flottants, en particulier via l'outil PROMISE. L'auto-ajustement consiste à réduire la précision des variables flottantes d'un code, de manière automatique, tout en respectant une contrainte de précision sur le résultat final. Abaisser la précision des variables présente de nombreux avantages en termes de performances en temps, en mémoire, et en énergie consommée, d'où son application à des codes HPC. À cette fin, de nombreux outils existent. La particularité de PROMISE est l'utilisation de l'Arithmétique Stochastique Discrète (ASD) via la bibliothèque de validation numérique CADNA, qui lui permet d'estimer au mieux la précision des résultats.

Alors que la réduction de réseaux de neurones utilise en général des méthodes spécifiques, ces derniers pourraient aussi tirer bénéfice de l'auto-ajustement de la précision. En ce sens, PROMISE a été appliqué à quatre différents réseaux de neurones, démontrant la possible diminution des précisions flottantes dans la phase d'inférence, sans compromettre la précision des résultats. Deux approches sont testées. La première consiste à prendre la même précision pour les paramètres d'un même neurone. Elle permet d'abaisser le plus de précisions possibles. La seconde consiste à prendre une précision par couche. Malgré un plus grand nombre de variables en précision élevée, elle permet d'obtenir un résultat plus rapidement. Le résultat de PROMISE dépendant de l'entrée choisie pour l'inférence, elle peut aussi se révéler moins spécifique à une entrée unique. L'auto-ajustement avec PROMISE a aussi été étudié dans la phase d'apprentissage des réseaux de neurones. Malgré une application limitée par le caractère aléatoire de cette phase, cela a permis de montrer que l'abaissement de la précision des variables flottantes avait très peu d'impact sur la phase d'apprentissage.

L'application de PROMISE met aussi en avant des améliorations significatives en termes de performances. En mémoire, les résultats sont équivalents aux données théoriques d'utilisation mémoire de chaque format flottant. En temps, les résultats d'accélération obtenus pour des codes vectorisés et non-vectorisés se rapprochent des résultats théoriques mais sont ternis par certaines opérations (*casts* et appels aux fonctions de bibliothèques). Tous ces résultats viennent confirmer l'intérêt de l'abaissement de la précision, notamment au sein de codes vectorisés.

Outre les performances des codes étudiés, celles de PROMISE ont également été examinées. La parallélisation de l'algorithme principal utilisé par PROMISE a été réalisée. L'implémentation d'un outil d'instrumentation basé sur Clang/LLVM a aussi été réalisée. Cet outil permet d'instrumenter des codes pour CADNA, remplaçant un script Perl, non-robuste et *ad hoc*. Il permet aussi d'instrumenter automatiquement des codes pour PROMISE, instrumentation effectuée jusqu'ici à la main. Une troisième version de cet outil, sous forme d'API Python, vient remplacer l'analyse et la création de code effectuées au sein de PROMISE, rendant ces étapes plus robustes.

Abstract

Title : *Precision Auto-tuning and Numerical Validation*

Keywords : Floating-point, Precision, Auto-tuning

Abstract :

This thesis focuses on the auto-tuning of floating-point precision, particularly via the PROMISE tool. Auto-tuning involves automatically reducing the precision of floating-point variables in a code while respecting an accuracy constraint on the final result. Lowering the precision of variables offers numerous advantages in terms of time, memory, and energy performance, making it applicable to HPC codes. To this end, many tools exist. The particularity of PROMISE lies in its use of Discrete Stochastic Arithmetic (DSA) through the CADNA numerical validation library, allowing it to optimally estimate the accuracy of results.
While neural network reduction generally uses specific methods, they could also benefit from precision auto-tuning. Therefore, PROMISE was applied to four different neural networks, demonstrating the possible reduction of floating-point precisions during the inference phase without compromising result accuracy. Two approaches were tested. The first approach involves using the same precision for the parameters of a single neuron, allowing for the maximum reduction of precisions. The second approach assigns a precision per layer. Despite having more variables in high precision, it allows for faster results. Since PROMISE's outcome depends on the input chosen for inference, it can also be less specific to a single input. Auto-tuning with PROMISE was also studied during the training phase of neural networks. Despite being limited by the randomness of this phase, it showed that reducing the precision of floating-point variables had very little impact on the training phase.
The application of PROMISE also highlights significant performance improvements. In terms of memory, the results are equivalent to the theoretical data on memory usage for each floating-point format. In terms of time, the acceleration results obtained for vectorized and non-vectorized codes are close to the theoretical results but are somewhat hindered by certain operations (casts and library function calls). All these results confirm the interest in reducing precision, particularly within vectorized codes.
In addition to examining the performance of the studied codes, the performance of PROMISE was also evaluated. The main algorithm used by PROMISE was parallelized. The implementation of an instrumentation tool based on Clang/LLVM was also carried out. This tool allows for the instrumentation of codes for CADNA, replacing a Perl script that was neither robust nor *ad hoc*. It also allows for the automatic instrumentation of codes for PROMISE, which had to be done manually. A third version of this tool, in the form of a Python API, replaces the analysis and code generation performed within PROMISE, making these steps more robust.

Remerciements

La rédaction des remerciements vient conclure le travail réalisé lors des trois années précédentes, composé de périodes de travail passionnantes comme de moments difficiles. Ce travail n'aurait donc pas été possible sans le soutien et l'aide de nombreuses personnes.

Tout d'abord, je tiens à remercier Daniel Menard et Guillaume Revy qui ont accepté de rapporter ce manuscrit. Leurs retours et remarques ont été très instructifs et ont participé à enrichir ce document. Mes remerciements s'adressent aussi à Pierre Fortin pour avoir accepté de faire partie du jury et d'endosser le rôle de président. Présenter mon travail devant ce jury a été un plaisir,et je remercie ce dernier pour les questions très pertinentes qui m'ont été posées, donnant lieu à des échanges plus intéressants les uns que les autres.

Merci également à Fabienne Jézéquel, Stef Graillat et Thibault Hilaire qui m'ont tous les trois encadré durant ces années de thèse. Grâce à leurs conseils et leur expertise, j'ai eu la chance de beaucoup apprendre, que ce soit sur le monde de la recherche, comme sur le monde de l'arithmétique flottante.

Une thèse, c'est aussi de nombreuses rencontres au sein d'une équipe, d'un laboratoire, et même d'ailleurs. En ce sens, je tiens à remercier toutes les personnes dont j'ai pu croiser le chemin, en conférence ou lors de séminaires, et avec qui les discussions, scientifiques ou non, ont toujours été intéressantes et ont rendu cette expérience d'autant plus enrichissante.

Mes remerciements à tous les membres du laboratoire LIP6, et notamment à tous les membres de l'équipe PEQUAN qui m'a accueilli en son sein de la meilleure manière possible. Cette thèse s'inscrit aussi dans le projet ANR INTERFLOP, dont je remercie tous les membres qui ont été d'un accueil irréprochable.

Je tiens aussi à remercier en particulier mes collègues et amis doctorants avec qui j'ai partagé ces années. Merci à Arthur Filoche pour les PEQUAN Records et les sessions dig. Merci à Théo Archambault et Dimitri Lesnoff pour les pauses-café, qui ne seront plus jamais les mêmes sans un pédantix, une partie d'échec ou une énigme à résoudre. Merci à Roméo Molina avec qui j'ai partagé de nombreux moments de ces trois années, au laboratoire et en conférence, comme à faire des danses Ariégeoises ou des sessions d'écriture à la campagne.

Enfin, je tiens à remercier ma famille sans qui tout ce parcours n'aurait pas été possible. Merci notamment à mes parents, qui m'ont toujours soutenu malgré des moments difficiles, et à mon frère qui a toujours été présent quand j'en avais besoin.

Table des matières

Table des figures xi

Liste des tableaux xiii

Introduction 1

I. État de l'art 9

1. Arithmétique flottante 11
 1.1. Norme IEEE-754 11
  1.1.1. Nombres normalisés, nombres dénormalisés, nombres spéciaux . . . 12
  1.1.2. Formats 12
  1.1.3. Exemples 13
  1.1.4. Modes d'arrondi 13
  1.1.5. Unité d'arrondi 14
 1.2. Autres formats 15
 1.3. Erreurs d'arrondi 16
 1.4. Conclusion 18

2. Validation numérique 19
 2.1. Différentes approches 19
  2.1.1. Analyse directe et analyse inverse 19
  2.1.2. Arithmétique par intervalles 20
  2.1.3. Analyse par interprétation abstraite 21
  2.1.4. Méthodes probabilistes 21
 2.2. Arithmétique Stochastique Discrète et son implémentation 22
  2.2.1. Méthode CESTAC 22
  2.2.2. Arithmétique Stochastique Discrète (ASD) 23
  2.2.3. CADNA 23
 2.3. Autres outils probabilistes de validation numérique 27
  2.3.1. VERROU 27
  2.3.2. VERIFICARLO 28
 2.4. Conclusion 28

3. Auto-ajustement de la précision 29
 3.1. Delta-Debug 29
 3.2. PROMISE 32

3.3. Autres outils d'auto-tuning de précision 36
3.3.1. Outils statiques 36
3.3.2. Outils dynamiques 37
3.4. Conclusion 38

II. Contributions 41

4. Auto-tuning de précision de réseaux de neurones avec PROMISE 43
4.1. Réseaux de neurones 44
4.2. PROMISE et inférence 46
4.2.1. Méthodologie 46
4.2.2. Résultats expérimentaux 47
4.3. PROMISE et entraînement 62
4.3.1. Entraînement de réseaux de neurones 62
4.3.2. Application de PROMISE 63
4.4. Conclusion 65

5. PROMISE et HPC 67
5.1. Performances en mémoire 67
5.1.1. Méthodologie 68
5.1.2. Résultats 68
5.2. Performances en temps 72
5.2.1. Méthodologie 72
5.2.2. Résultats 73
5.3. Amélioration des performances de PROMISE 76
5.4. Conclusion 78

6. Instrumentation de codes pour la validation et l'auto-ajustement de la précision 79
6.1. LLVM et Clang 80
6.2. Instrumentation LLVM 80
6.2.1. Arbre de la Syntaxe Abstraite (AST) Clang 80
6.2.2. Matchers et Replacements 81
6.3. Instrumentation pour CADNA 84
6.4. Instrumentation pour PROMISE 88
6.4.1. Remplacement des types 88
6.4.2. *Parsing* de codes 91
6.5. Conclusion 92

7. Conclusion générale et perspectives 95

Table des figures

1.1. Représentation des quatre modes d'arrondi pour x et y réels positifs 14

3.1. Delta-Debug pour l'auto-tuning de précision avec deux types différents 32
3.2. Fonctionnement de PROMISE pour les types half/simple/double 36

4.1. Réseau de neurones à deux couches 45
4.2. Produit de convolution 2D 45
4.3. Max pooling avec un filtre 2×2 46
4.4. Logigramme de la traduction d'un réseau de neurones Python en programme C++ pour PROMISE 47
4.5. Représentation matricielle des calculs effectués dans le réseau de neurones sinus 48
4.6. Nombre de variables de chaque type et temps de calcul pour le réseau sinus avec comme valeur d'entrée 0.5 49
4.7. Précision de chaque couche pour le réseau sinus avec comme valeur d'entrée 0.5 50
4.8. Nombre de variables de chaque type et temps de calcul pour le réseau sinus avec comme valeur d'entrée 2.37 51
4.9. Précision de chaque couche pour le réseau sinus avec comme valeur d'entrée 2.37 51
4.10. Nombre de variables de chaque type et temps de calcul pour le réseau sinus avec 1000 valeurs d'entrée 52
4.11. Précision de chaque couche pour le réseau sinus avec 1000 valeurs d'entrée . . 52
4.12. Images de la base de données MNIST (https://en.wikipedia.org/wiki/MNIST_database) 53
4.13. Nombre de variables de chaque type et temps de calcul pour le réseau MNIST avec comme valeur d'entrée test_data[61] 54
4.14. Précision de chaque couche pour le réseau MNIST avec comme valeur d'entrée test_data[61] 54
4.15. Nombre de variables de chaque type et temps de calcul pour le réseau MNIST avec comme valeur d'entrée test_data[91] 55
4.16. Précision de chaque couche pour le réseau MNIST avec comme valeur d'entrée test_data[91] 56
4.17. Images de la base de données CIFAR10 (https://www.cs.toronto.edu/~kriz/cifar.html) 57
4.18. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau CIFAR avec comme valeur d'entrée test_data[386] 57
4.19. Précision de chaque couche pour le réseau CIFAR avec comme valeur d'entrée test_data[386] 58
4.20. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau CIFAR avec comme valeur d'entrée test_data[731] 58

4.21. Précision de chaque couche pour le réseau CIFAR avec comme valeur d'entrée test_data[731] 59
4.22. Pendule inversé (image de [?]) 60
4.23. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau pendule inversé avec comme valeur d'entrée (0.5,0.5) 60
4.24. Précision de chaque couche pour le réseau pendule inversé avec comme valeur d'entrée (0.5,0.5) 61
4.25. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau pendule inversé avec comme valeur d'entrée (-3,-6) 61
4.26. Précision de chaque couche pour le réseau pendule inversé avec comme valeur d'entrée (-3,-6) 62
4.27. Précision de chaque couche pour le réseau MNIST en entraînement 64

5.1. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau sinus avec comme valeur d'entrée 0.5 68
5.2. Précision de chaque couche et mémoire théorique utilisée pour le réseau sinus avec comme valeur d'entrée 0.5 69
5.3. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau MNIST avec comme valeur d'entrée test_data[61] 69
5.4. Précision de chaque couche et mémoire théorique utilisée pour le réseau MNIST avec comme valeur d'entrée test_data[61] 70
5.5. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau CIFAR avec comme valeur d'entrée test_data[386] 70
5.6. Précision de chaque couche et mémoire théorique utilisée pour le réseau CIFAR avec comme valeur d'entrée test_data[386] 71
5.7. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau pendule inversé avec comme valeur d'entrée (0.5,0.5) 71
5.8. Précision de chaque couche et mémoire théorique utilisée pour le réseau pendule inversé avec comme valeur d'entrée (0.5, 0.5) 72
5.9. Nombre de variables PROMISE de chaque type et temps d'exécution total pour le réseau MNIST avec comme valeur d'entrée test_data[61] 74
5.10. Précision de chaque couche et temps d'exécution total pour le réseau MNIST avec comme valeur d'entrée test_data[61] 74
5.11. Nombre de variables PROMISE de chaque type et temps d'exécution des produits matrice-vecteur pour le réseau MNIST avec comme valeur d'entrée test_data[61] 75
5.12. Précision de chaque couche et temps d'exécution des produits matrice-vecteur pour le réseau MNIST avec comme valeur d'entrée test_data[61] 75
5.13. Temps de calcul de PROMISE en utilisant les différentes versions du Delta-Debug sur le réseau de neurones MNIST avec comme valeur d'entrée test_data[61] . 77
5.14. Temps de calcul de PROMISE en utilisant les différentes versions du Delta-Debug sur le réseau de neurones CIFAR avec comme valeur d'entrée test_data[386] . 77

6.1. Représentation graphique simplifiée de l'AST pour l'expression "double result = x * 42.0 + 12.0;" 81

Liste des tableaux

1.1. Nombres spéciaux définis par la norme IEEE-754 12
1.2. Formats définis par la norme IEEE-754 13
1.3. Exemples de représentations de nombres flottants en binary32/simple 13
1.4. Unité d'arrondi pour chaque format standard 15
1.5. Formats spécifiques 16
1.6. Exemple d'accumulation d'erreurs avec somme de n fois la valeur $1/n$ 16

3.1. Exemple de configurations testées par le Delta-Debug - Situation 2 31
3.2. Exemple de configurations testées par le Delta-Debug - Situation 3 31

Introduction

Grâce à l'amélioration constante de la performance des ordinateurs, le calcul numérique permet aujourd'hui d'obtenir des résultats de calculs complexes à grande échelle : il s'agit du **Calcul Haute Performance**, ou **HPC** (*High Performance Computing*). Largement utilisé dans l'industrie, le Calcul Haute Performance permet de simuler et de prédire le résultat de phénomènes ou de modèles physiques et d'aider à la prise de décision. Appliqué dans de nombreux domaines (météorologie, astrophysique, nucléaire, aéronautique, etc.), il peut par exemple servir à simuler la physique d'un barrage, et aider de sa construction jusqu'à son bon fonctionnement au fil des années.

Comme son nom l'indique, un code HPC répond à des enjeux de performances. En plus d'être un gain non-négligeable lors de simulation, une meilleure performance en temps permet aussi de répondre à des contraintes lors de l'application du code dans des cas concrets, par exemple lors du calcul d'une position GPS en aéronautique. Une meilleure performance mémoire permet de la même manière de répondre à des contraintes matérielles, la taille des données à traiter étant de plus en plus importante quel que soit le domaine.

Cependant, le Calcul Haute Performance est soumis à différentes sources d'erreurs dues aux approximations effectuées, que ce soit lors de la traduction du phénomène physique en modèle mathématique (ensemble d'équations par exemple), ou lors des calculs numériques. Ce sont ces derniers, introduisant des erreurs d'arrondi inhérentes à l'arithmétique des ordinateurs, qui nous intéresseront en particulier. Avant l'application d'un code HPC, il faut vérifier la **qualité numérique** de celui-ci. Obtenir un code calculant un résultat de qualité s'avère souvent crucial, notamment lors de l'application dans des domaines critiques où une erreur dans le résultat ou au cours des calculs peut avoir des conséquences importantes dans le cas réel. Nous pouvons citer par exemple l'explosion de la fusée Ariane 5 en 1996, due à une mauvaise prise en compte de certains résultats intermédiaires intervenant dans le calcul du système de guidage inertiel principal.

Cette problématique est le fil principal de cette thèse qui porte sur l'amélioration des performances de codes par l'utilisation de types numériques de tailles plus faibles, qui contiennent moins d'informations. L'utilisation de tels types, moins précis, engendre cependant des erreurs plus importantes sur les résultats. L'amélioration des performances, réalisée de manière automatique en cherchant à utiliser un type de taille réduite pour les différentes variables, se fait alors toujours en respectant une précision souhaitée sur le résultat final. Cela est réalisé en utilisant des outils d'**auto-ajustement** des types des variables. Nous utiliserons en particulier l'outil PROMISE [**?**], qui a la particularité d'être combiné à un outil de **validation numérique**, CADNA [**?**, **?**, **?**].

Contexte

Simulation numérique

En raison des contraintes inhérentes aux expérimentations physiques concrètes (coût, échelle, dangerosité, etc.), les expérimentations sont souvent réalisées par le biais de simulations numériques. Bien qu'il soit aujourd'hui possible de simuler un modèle physique simple sur un ordinateur portable, cela devient impraticable lorsque des centaines de millions de calculs sont requis. Le Calcul Haute Performance est donc utilisé pour la simulation numérique de larges modèles, avec un nombre de calculs pouvant aller jusqu'à plusieurs quadrillions par seconde (un quadrillion est égal à 10^{24}). Nous retrouvons son application principalement en recherche et développement dans des domaines tels que la mécanique, la science des matériaux, l'astrophysique, le nucléaire, l'aéronautique, la biologie, la chimie, etc. De nombreuses utilisations sont aussi faites en sciences humaines comme pour des études de démographie ou de sociologie, ou encore dans le secteur de la finance.

La simulation numérique désigne "le procédé selon lequel on exécute un (des) programme(s) sur un (des) ordinateur(s) en vue de représenter un phénomène physique" [**?**]. Mais avant d'exécuter un tel programme, il faut savoir comment traduire le phénomène physique en une suite d'instructions informatiques. C'est ici que la modélisation mathématique entre en jeu. En prenant en compte les principes physiques fondamentaux (principes de la thermodynamique, principe de conservation de la masse, principe d'inertie, etc.), la modélisation permet de représenter le problème étudié par un ensemble d'équations mettant en lien les différents paramètres nécessaires à la bonne modélisation du phénomène (vitesse, température, etc.). Ces paramètres, qui correspondent aux grandeurs physiques intervenant dans les équations, permettent de suivre l'évolution des différents éléments mis en jeu.
Par la suite, il reste à traduire ces équations en code informatique. Cependant, alors que les paramètres réels sont définis sur un ensemble continu de points (par exemple dans l'espace), il est impossible pour l'ordinateur de calculer une infinité de résultats. Il est alors nécessaire de discrétiser le domaine d'application. Pour cela, deux approches sont principalement utilisées. Les méthodes déterministes consistent à résoudre les équations après discrétisation des variables. Parmi elles, on retrouve par exemple la méthode des volumes finis ou la méthode des éléments finis. Les méthodes probabilistes quant à elles, consistent en des tirages aléatoires successifs afin d'approcher un résultat avec une variance sur la précision de celui-ci. On retrouve parmi elles les méthodes de Monte-Carlo, souvent utilisées.

Quelle que soit la méthode utilisée, elle constitue une première approximation du monde physique. À cette approximation s'ajoute les approximations apparaissant lors des calculs numériques, dues à la représentation des nombres réels dans un ordinateur.

Arithmétique des ordinateurs

Dans un ordinateur, un nombre réel ne peut être représenté que sur un certain nombre de bits. La plupart des nombres réels, susceptibles d'avoir une infinité de décimales, ne peuvent alors pas être représentés exactement. L'ensemble des réels représentables dans un ordinateur est donc un ensemble fini. Afin de représenter le plus de nombres possibles,

des stratégies sont établies. Les nombres à virgule fixe sont un type de données permettant de représenter des réels avec un nombre fixe de chiffres après la virgule. La position de la virgule est fixée, avec à sa gauche la partie entière binaire et à sa droite la partie fractionnaire binaire, c'est-à-dire que chaque chiffre à droite de la virgule correspond à une puissance négative de 2. Cette représentation est souvent utilisée dans des microcontrôleurs qui ne possèdent pas d'autre unité de calcul. La représentation en virgule fixe possède cependant ses limites. L'écart entre deux nombres à virgule fixe consécutifs est toujours de l'unité la plus faible 2^{-n_f}, avec n_f le nombre de bits pour la partie fractionnaire. De ce fait, en considérant un nombre réel x n'ayant pas de représentation exacte en virgule fixe, il faut approcher celui-ci par un des nombres à virgule fixe aux alentours, que l'on peut noter X. En prenant le plus proche, cela nous donne une erreur absolue bornée par $2^{-n_f}/2$. Si l'on considère maintenant l'erreur relative sur de tels nombres, celle-ci dépend de la valeur considérée. Pour toute valeur proche de 0, on peut remarquer que l'erreur relative est de 100%. En effet, quelle que soit la valeur x proche de 0, sa représentation en virgule fixe sera $X = 0$, et l'erreur relative entre la valeur réelle et sa représentation, donnée par $\frac{|x-X|}{x}$, se simplifie en $\frac{x}{x} = 1$. Autrement dit, toute l'information est perdue pour ces valeurs. En revanche, à mesure que les valeurs s'éloignent de 0, la limite supérieure de cette erreur diminue progressivement.

Afin de limiter cette répartition inégale de l'erreur relative, les nombres à virgule flottante peuvent être utilisés. Ceux-ci peuvent être rapprochés de la fameuse "écriture scientifique", c'est-à-dire un nombre multiplié par une puissance de 10. De la même manière, les nombres à virgule flottante stockent un exposant qui permet de placer la virgule après son évaluation (puissance de 2 en binaire). Par rapport aux nombres à virgule fixe, cela permet de représenter une plage de nombres bien plus grande, mais aussi de représenter des nombres proches de 0 plus petits, limitant la perte d'information. Ainsi, la représentation par les nombres à virgule flottante, ou plus simplement nombres flottants, est l'arithmétique la plus utilisée dans les ordinateurs. Une première apparition de tels nombres remonte à l'ordinateur de Konrad Zuse, le Z3, apparu en 1941 [**?**]. Alors que par la suite, selon les constructeurs, différentes gestions des nombres flottants apparaissent, la norme IEEE-754 [**?**, **?**, **?**] vient en 1985 fixer un standard pour la représentation des nombres flottants en base 2. Malgré cela, le fait de ne pas pouvoir représenter exactement tous les nombres réels introduit des erreurs d'arrondi qu'il faut prendre en compte et analyser.

Analyser les erreurs d'arrondi

Analyser et estimer les erreurs d'arrondi s'avère crucial dans le cadre du HPC. À chaque résultat numérique, une erreur d'arrondi due à l'approximation en nombre flottant est introduite. Au fil des opérations successives, ces erreurs s'accumulent, pouvant donner un résultat erroné. Or, obtenir un résultat incorrect peut avoir des conséquences dans certains domaines critiques, comme dans le cas de l'explosion de la fusée Ariane évoquée précédemment. Pour analyser les erreurs d'arrondi, différentes approches sont utilisées.
Parmi elles, l'analyse statique consiste à analyser le programme sans avoir à l'exécuter. Elle se fait en analysant textuellement le code, en appliquant par exemple une analyse par interprétation abstraite [**?**]. Autre méthode statique, l'analyse inverse [**?**] consiste à déterminer

les paramètres initiaux d'un système à partir du résultat calculé, considéré alors comme correct. L'appellation "analyse inverse" provient du fait que les erreurs sont "réinjectées" dans le problème initial. L'analyse dynamique, qui comporte différentes méthodes, consiste quant à elle à exécuter le code pour en extraire des informations. Les résultats obtenus permettent de déterminer l'erreur sur le résultat. Au sein des méthodes dynamiques, nous retrouvons les approches probabilistes qui consistent à obtenir plusieurs fois le résultat d'un même programme afin d'en estimer l'erreur. Comme évoqué précédemment, les plus courantes sont les méthodes de Monte-Carlo. Enfin, l'arithmétique par intervalles [**?**] consiste à manipuler des intervalles au lieu de valeurs uniques afin d'inclure les incertitudes.

Alors que les méthodes statiques et d'arithmétique par intervalles permettent d'obtenir des bornes sûres sur le résultat, celles-ci ne sont pas adaptées à une application sur des codes HPC. C'est pour cela que nous utiliserons dans notre cas la méthode probabiliste CESTAC [**?**, **?**] au travers de l'Arithmétique Stochastique Discrète (ASD). En calculant plusieurs fois le résultat de chaque opération au sein d'un programme, avec différentes propagations des erreurs d'arrondi, l'ASD permet d'estimer le nombre de chiffres significatifs corrects du résultat. Le logiciel CADNA [**?**, **?**, **?**], développé au LIP6, implémente l'ASD pour une utilisation au sein de codes C/C++ et Fortran. Il permet d'estimer le nombre de chiffres corrects des résultats et de détecter les instabilités numériques.

Améliorations grâce aux précisions faibles

Une variable flottante représente un nombre réel sur un certain nombre de bits, fixé par le type choisi. On parle indifféremment de précision d'une variable, la précision de l'approximation flottante étant directement liée au nombre de bits utilisés. Les différentes précisions et leurs représentations en mémoire sont définies par la norme IEEE-754. Les 3 précisions les plus utilisées sont *half*, simple et double. Celles-ci seront définies précisément ultérieurement, mais retenons pour l'instant que la précision half utilise 2 fois moins de bits que la précision simple, et 4 fois moins que la précision double. Une précision plus faible, c'est-à-dire qui utilise moins de bits, présente ainsi un avantage important en mémoire par rapport aux précisions plus élevées. De plus, l'utilisation d'une précision plus faible permet bien souvent de meilleures performances. L'écriture en mémoire sur 2 fois moins de bits prend 2 fois moins de temps, et, par exemple, un code vectorisé en précision simple (resp. half) permet d'effectuer 2 fois (resp. 4 fois) plus d'opérations en simultané que le même code en précision double.
Utiliser une précision plus faible dans des codes HPC se révèle alors être une solution intéressante pour gagner en performance en temps, en mémoire, ainsi qu'en consommation énergétique. Cependant, l'utilisation d'une précision plus faible entraîne une perte évidente de... précision (comme son nom l'indique). L'utilisation d'une précision plus faible doit alors toujours être réalisée de manière réfléchie, sans compromettre le résultat.

Pour réaliser cela, des outils d'auto-ajustement de la précision sont utilisés. Ceux-ci prennent en entrée un code utilisant des nombres flottants et essaie d'en réduire le plus possible leurs précisions respectives, tout en respectant une précision demandée sur le

résultat. La précision du résultat est obtenue par comparaison avec un résultat de référence qui est en général le résultat du code exécuté avec la précision la plus élevée.
Les outils d'auto-ajustement de la précision peuvent se diviser en deux catégories : ceux avec une approche statique (FPTuner [?], Salsa [?], Rosa/Daisy [?, ?], TAFFO [?], POP [?]) et ceux avec une approche dynamique (CRAFT HPC [?], Precimonious [?], HiFPTuner [?], ADAPT [?], FloatSmith [?], PROMISE [?]). Nous utilisons dans notre cas l'outil dynamique PROMISE, développé au LIP6. Celui-ci a la particularité d'utiliser l'outil CADNA pour obtenir un résultat de référence validé numériquement. En particulier, c'est une des différences avec l'outil Precimonious, proche de PROMISE. La seconde différence principale est que, en plus de la précision du résultat, Precimonious prend aussi comme critère la performance en temps des programmes avec précisions réduites testés, tandis que PROMISE se concentre sur la minimisation des précisions utilisées.

Objectifs de la thèse

L'auto-ajustement de la précision avec PROMISE a été réalisé sur des codes issus de l'industrie, comme MICADO [?], un code de simulation de cœurs nucléaires développé par EDF. Cependant, alors que les grands modèles de langages, ou LLM (de l'anglais *Large Language Model*), tels que ChatGPT ou Gemini, sont devenus incontournables ces dernières années, l'auto-ajustement de la précision de réseaux de neurones, sur lesquels sont basés les LLM, a très peu été étudié. En effet, pour la compression de réseaux de neurones, c'est-à-dire la réduction de la taille du modèle, les méthodes spécifiques aux réseaux de neurones sont souvent privilégiées, comme par exemple l'élagage. Dans le cadre de cette thèse, nous nous proposons donc d'étudier l'auto-ajustement de la précision de différents réseaux de neurones.

L'application de PROMISE reste cependant contrainte par plusieurs facteurs.
Premièrement, PROMISE utilise l'outil CADNA afin de valider numériquement son résultat de référence. Si l'on veut pouvoir utiliser PROMISE sur un code, alors celui-ci doit aussi pouvoir être utilisé avec CADNA. À cette fin, CADNA a précédemment été étendu pour les bibliothèques externes largement utilisées au sein de codes HPC, comme la bibliothèque d'échange de données MPI [?]. Deuxièmement, le code donné à PROMISE doit être instrumenté pour transmettre les informations nécessaires au bon déroulement de l'auto-ajustement de la précision. Il est alors difficile de considérer l'auto-ajustement de la précision de codes trop complexe pour l'instrumentation. De plus, afin de tester les différentes possibilités dans la réduction des précisions, PROMISE doit recréer différentes versions du même code qu'il aura auparavant dû analyser et enregistrer. Cela peut s'avérer gourmand en temps sur des codes HPC de grande ampleur. Améliorer l'instrumentation de codes, qu'elle soit pour CADNA ou PROMISE, peut alors représenter un gain en performance non-négligeable pour l'outil.

Contributions

PROMISE a été appliqué à 4 réseaux de neurones différents : deux réseaux d'interpolation (fonction sinus et fonction de Lyapunov) et deux réseaux de classification (bases de données MNIST et CIFAR). Les résultats sont présentés pour chaque réseau de neurones de manière exhaustive, i.e. toutes les précisions possibles ont été demandées (1 à 15 chiffre(s), correspondant au maximum pour la précision double). Dans certains cas, le nombre maximal de chiffres exacts est inférieur à 15. Étant donné les calculs effectués dans un réseau de neurones, deux situations différentes ont été considérées pour chaque réseau : soit en fixant un type par neurone, soit en fixant un type par couche. Bien que les deux approches donnent des résultats différents, nous montrons que la réduction de réseaux de neurones en phase d'inférence est bien réalisable, avec certaines limites. Nous montrons en revanche que la réduction dès la phase d'entraînement est quant à elle beaucoup plus contrainte.

Les résultats sur les réseaux de neurones ont aussi permis d'étudier les bénéfices de la réduction de la précision et de la précision mixte via l'application de PROMISE. Les gains de performance, en temps comme en mémoire, sont présentés et analysés. Les mêmes codes en versions vectorisées et non vectorisées sont étudiés, mettant en avant l'intérêt de l'utilisation de la vectorisation. Dans le cadre de cette étude liée aux performances, les performances de l'outil PROMISE ont aussi été considérées. Une amélioration proposée a été la parallélisation de l'algorithme utilisé par PROMISE, le Delta-Debug [**?**, **?**]. Cette parallélisation permet d'obtenir, sur les situations testées, un gain en temps d'un facteur au plus 3.2.

Enfin, dans l'objectif d'améliorer les performances de PROMISE, mais aussi de rendre l'outil beaucoup plus robuste et applicable à des codes plus complexes, un outil d'instrumentation basé sur LLVM a été créé. Celui-ci est utilisé pour l'instrumentation de codes pour CADNA comme pour PROMISE. Dans le cas de PROMISE, il est utilisé pour instrumenter automatiquement le fichier d'entrée, ainsi que pour la récupération d'informations depuis le fichier et la création des nouveaux fichiers à tester.

Publications et interventions

Articles de conférences internationales

—

—

Interventions dans des conférences ou workshops

—

—

Poster

—

Plan

Le travail réalisé est présenté en deux grandes parties.

La première partie est consacrée à l'état de l'art, introduisant l'arithmétique flottante dans le chapitre 1, notamment les définitions des différents types flottants par la norme IEEE-754 ainsi que les erreurs d'arrondi associées à l'arithmétique flottante. Le chapitre 2 présente la validation numérique, les méthodes ainsi que les outils associés, avec notamment une présentation de l'Arithmétique Stochastique Discrète et de CADNA (section 2.2). Le chapitre 3 clôt cette première partie en présentant l'auto-ajustement de la précision, méthode et outils associés, avec en particulier une partie consacrée à PROMISE (section 3.2).

La seconde partie est dédiée aux contributions. Elle décrit d'abord l'application de l'auto-ajustement de la précision sur les réseaux de neurones dans le chapitre 4, présentant les notions principales de ceux-ci, puis les possibilités d'application de l'auto-tuning de précision sur les phases d'inférence et d'entraînement. Le chapitre 5 vient compléter l'étude sur les réseaux de neurones en mettant en avant les bénéfices de l'application de l'auto-ajustement de la précision, sur des codes non-vectorisés et vectorisés. Enfin, le chapitre 6 porte sur la réalisation d'un outil d'instrumentation décliné en une version pour CADNA et une version pour PROMISE.

Première partie

État de l'art

Arithmétique flottante

1

Manipuler des nombres dans un ordinateur n'est pas une tâche évidente puisque ceux-ci doivent être représentés sur un nombre fini de bits. Pour représenter un nombre entier, par exemple 10, nous pouvons prendre sa représentation binaire 1010_2 (x_2 indique un nombre binaire). Cependant, la mémoire d'un ordinateur étant limitée, il existe là déjà une limite, le nombre le plus grand représentable. La tâche se complique d'autant plus lorsque nous voulons représenter des nombres réels. Une représentation symbolique des réels, ne manipulant pas leur valeur mais une représentation abstraite, est possible, mais se révèle très peu performante. Au contraire, l'arithmétique à virgule flottante, ou plus simplement arithmétique flottante, représente un bon compromis entre performance et précision dans la représentation des nombres réels.
L'ensemble des nombres flottants, noté $\mathbb{F}$, est un ensemble fini de nombres pouvant s'écrire de la façon suivante :

$$x = s \times m \times \beta^e \tag{1.1}$$

où $s \in \{-1, 1\}$ est le signe, $m = d_0.d_1...d_{p-1}$ la mantisse avec les chiffres $0 \leq d_i < \beta, 0 \leq i \leq p-1$, p la précision (taille de la mantisse), β la base et $e \in [e_{min}, e_{max}]$ l'exposant.

La représentation d'un nombre flottant en machine dépendait à l'origine du processeur, et donc de son constructeur, qui fixait une base β, un intervalle $[e_{min}, e_{max}]$ et une taille de mantisse p. Ces différences rendaient difficile la portabilité des programmes d'un processeur à un autre. La nécessité d'une représentation commune à tous les processeurs est donc apparue, et s'est concrétisée avec l'apparition en 1985 de la norme IEEE–754 [**?**].

1.1 Norme IEEE-754

Révisée en 2008 [**?**] puis 2019 [**?**], la norme IEEE-754 fait office de référence en matière d'arithmétique flottante. Elle a notamment permis de :

— fixer différents formats de représentation ;

— définir les différents modes d'arrondi autorisés ;

— définir les valeurs spéciales $(0, -\infty, +\infty, NaN)$;

— définir les opérations élémentaires $(+, -, \times, \div, \sqrt{})$.

En général, et dans toute la suite de ce document, la base β est 2 (binaire oblige). L'exposant, pouvant être positif ou négatif, est biaisé afin d'être stocké sous forme d'un nombre entier non signé. En binaire, ce biais b est égal à $2^{n-1} - 1$ où n est le nombre de bits de l'exposant. Le biais est donc constant une fois la taille de l'exposant fixée. L'exposant devient un exposant biaisé $e_b = e + b \in [1, 2 \cdot e_{max}]$, valeur finalement stockée en mémoire. Le signe,

quant à lui, est toujours stocké sur un seul bit b_s permettant d'évaluer $s = (-1)^{b_s} \in \{-1, 1\}$.

1.1.1 Nombres normalisés, nombres dénormalisés, nombres spéciaux

Pour un même format, la norme IEEE-754 défini les nombres normalisés et les nombres dénormalisés, permettant de représenter deux plages de nombres différentes. Pour les nombres normalisés, le premier bit de la mantisse est toujours égal à 1 et n'est donc pas stocké explicitement, on l'appelle le "bit implicite". Cela permet de gagner un bit de précision et de représenter un intervalle de nombres plus grand. Ainsi la valeur des nombres normalisés est donnée par $x = -1^{b_s} \times (1.m) \times 2^{e_b - b}$, avec m la mantisse stockée en mémoire, e_b l'exposant stocké en mémoire et b le biais. Les nombres dénormalisés sont les nombres ayant en mémoire un exposant biaisé fixé à 0 ainsi qu'une mantisse non nulle. Pour ces derniers, aucun bit implicite n'est ajouté. Leur valeur est donc donnée par $x = -1^{b_s} \times m \times 2^{-b}$. Ils permettent notamment de représenter des valeurs plus proches de 0 que les nombres normalisés. En effet, le plus petit nombre normalisé positif est égal à $1.00..0_2 \times 2^{1-b}$ où la mantisse est donc égale à 0 et l'exposant biaisé à 1 (l'association mantisse nulle et exposant biaisé nul est réservée pour représenter 0), tandis que le plus petit nombre dénormalisé est égal à $0.00..1_2 \times 2^{-b+1}$. Les nombres dénormalisés assurent de plus la continuité avec un plus grand nombre dénormalisé égal à $0.11..1_2 \times 2^{-b+1}$. Outre ces deux représentations, la norme IEEE-754 fixe la représentation de nombres spéciaux, notamment 0, donnés dans le tableau suivant.

x	bit de signe	exposant biaisé	mantisse
$+0$	0	0	0
-0	1	0	0
$+\infty$	0	$2 \times e_{max} + 1$	0
$-\infty$	1	$2 \times e_{max} + 1$	0
NaN (Not a Number)	0	$2 \times e_{max} + 1$	$m \neq 0$

Tableau 1.1. Nombres spéciaux définis par la norme IEEE-754

Remarque 1.1.1. Il y a deux zéros définis par la norme IEEE-754, un positif et un négatif.

1.1.2 Formats

Pour définir un format (ou type), il suffit donc de fixer la taille de la mantisse ainsi que celle de l'exposant. Les différents formats définis par la norme IEEE-754 sont représentés dans le tableau ci-dessous. Dans la colonne mantisse, un "+1" est indiqué correspondant au bit implicite, rajoutant 1 aussi à la précision p du format.

La norme IEEE-754 définit aussi un format étendu associé à chaque format standard, permettant d'utiliser des formats avec une taille de mantisse à l'intermédiaire entre deux formats standards. De plus, 3 formats en base 10 sont aussi définis depuis la révision de 2008. Tous ces formats ne sont cependant pas toujours disponibles et sont peu utilisés. Ils ne

format/nom	taille	mantisse (bits)	exposant (bits)
binary16/half	16 bits	10 +1	5
binary32/simple	32 bits	23 +1	8
binary64/double	64 bits	52 +1	11
binary128/quadruple	128 bits	112 +1	15

Tableau 1.2. Formats définis par la norme IEEE-754

sont pas considérés ici.
Le terme de *précision* sera aussi utilisé pour faire référence à un format, la précision d'un nombre flottant étant directement reliée, par définition, au format utilisé. On parlera par exemple de *précision simple* ou *précision double*, ou encore de la précision d'une variable, en parlant de son format de définition.

1.1.3 Exemples

Nous donnons dans le tableau ci-dessous quelques exemples de représentations de nombres. Pour mieux comprendre leurs représentations, prenons comme exemple 10.375. Converti en binaire, nous avons $10 = 1010_2$ et $0.375 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0.011_2$. D'où $10.375 = 1.010011_2 \times 2^3$, notre représentation flottante normalisée, l'unité 1 étant notre bit implicite. La mantisse est donc 010011_2, complétée de 0 en mémoire, l'exposant biaisé est $3 + 127 = 130$ pour une représentation en précision simple, soit 10000010_2 en binaire.

x	bit de signe	exposant biaisé	mantisse
1	0	01111111	00000000000000000000000
2.5	0	10000001	01000000000000000000000
10.375	0	10000010	01001100000000000000000
-10.375	1	10000010	01001100000000000000000

Tableau 1.3. Exemples de représentations de nombres flottants en binary32/simple

1.1.4 Modes d'arrondi

Les nombres flottants (ensemble fini) ne permettant pas de représenter tous les nombres réels (ensemble continu), un choix doit être fait lors du stockage d'un nombre n'ayant pas de représentation exacte en flottant, ce qui est souvent le cas pour le résultat d'une opération entre deux nombres flottants. Les différents modes d'arrondi permettent de choisir une approximation en flottant du nombre en question.
La norme IEEE-754 en définit 4 différents :

— L'arrondi vers $+\infty$ qui arrondit à la plus grande valeur la plus proche ;

— L'arrondi vers $-\infty$ qui arrondit à la plus petite valeur la plus proche ;

— L'arrondi vers 0 qui arrondit à la valeur la plus proche de 0 ;

— L'arrondi au plus près qui arrondit à la valeur la plus proche. Si le nombre se trouve à mi-chemin entre deux valeurs, il est arrondi à la valeur la plus proche ayant un bit de poids faible pair. Il s'agit du mode d'arrondi par défaut.

Notons ceux-ci $\circ_{+\infty}$, $\circ_{-\infty}$, $\circ_0$ et $\circ_\sim$ respectivement. On appelle modes d'arrondi dirigés les modes d'arrondi $\circ_{+\infty}$, $\circ_{-\infty}$ et $\circ_0$. Tous sont représentés dans la Figure 1.1 pour deux nombres réels positifs.
Pour chacune des opérations élémentaires $\diamond \in \{+, -, \times, \div\}$, on a :

$$\forall v_1, v_2 \in \mathbb{F}, v_1 \diamond_\circ v_2 = \circ(v_1 \diamond v_2) \tag{1.2}$$

avec $\diamond_\circ$ l'opération flottante et $\circ \in \{\circ_{+\infty}, \circ_{-\infty}, \circ_0, \circ_\sim\}$ le mode d'arrondi. L'équation 1.2 nous donne le comportement des opérations élémentaires sur les nombres flottants. Le résultat d'une opération flottante $\diamond_\circ$ avec un mode d'arrondi $\circ$ choisi est le même que le résultat d'une opération exacte ensuite arrondi avec $\circ$. La norme IEEE-754 définit de plus l'opération racine carrée $\sqrt{}$.

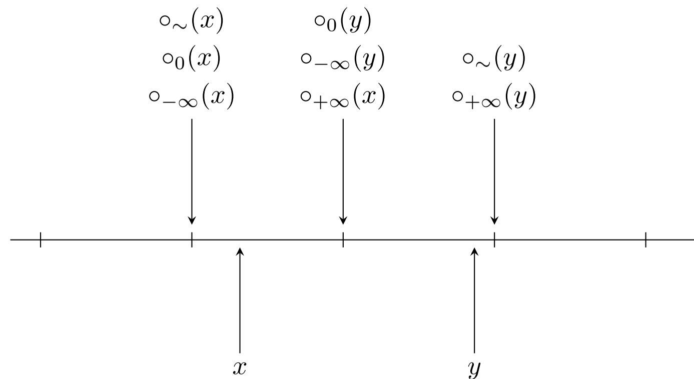

Figure 1.1. Représentation des quatre modes d'arrondi pour x et y réels positifs

1.1.5 Unité d'arrondi

La notion d'unité d'arrondi, aussi appelé *machine epsilon* (epsilon de la machine), est une notion définie comme suit.

Définition 1.1.2 (Unité d'arrondi). L'unité d'arrondi $\mathbf{u}$ des nombres flottants en base β, de précision p est définie comme :

$$\mathbf{u} = \tfrac{1}{2}\beta^{1-p}.$$

L'unité d'arrondi représente le maximum de la différence relative entre l'approximation flottante et le résultat réel d'un calcul, due au mode d'arrondi. En effet, pour tous X, Y flottants, pour toute opération $\diamond \in \{+, -, \times, \div\}$ et pour tout mode d'arrondi $\circ \in \{\circ_{+\infty}, \circ_{-\infty}, \circ_0, \circ_\sim\}$, on a :

$$\circ(X \diamond Y) = (X \diamond Y)(1 + \epsilon),$$

où ϵ est l'erreur relative sur le résultat approché. Cette erreur vérifie $|\epsilon| < \mathbf{u}$ pour l'arrondi au plus près $\circ_\sim$, et $|\epsilon| < 2\mathbf{u}$ pour les modes d'arrondi dirigés $\circ_{+\infty}, \circ_{-\infty}$ et $\circ_0$. Cette notion est donc très utile pour calculer les bornes d'erreur.
Le tableau ci-dessous donne la valeur de l'unité d'arrondi pour chacun des formats définis dans le Tableau 1.2.

format	Unité d'arrondi
half	2^{-11}
simple	2^{-24}
double	2^{-53}
quadruple	2^{-113}

Tableau 1.4. Unité d'arrondi pour chaque format standard

1.2 Autres formats

Outre ceux définis par la norme IEEE-754, il existe d'autre formats souvent plus récents permettant de représenter les nombres flottants. C'est le cas par exemple des nombres *unums* (pour *universal numbers*), introduits par Gustafson dans son livre *The End of Error* [**?**], dont la dernière version, les nombres *posits* (*unums* type III), a été introduite en 2017 [**?**] et complétée en 2022 [**?**]. À la différence des nombres flottants, les nombres *posit* sont composés de quatre parties : le bit de signe, le régime, l'exposant et la fraction (qui correspond à la mantisse des flottants IEEE-754). Le régime permet de déterminer une échelle large pour les nombres en utilisant une séquence de bits simple, et ainsi de représenter efficacement des nombres très grands ou très petits. La valeur du régime k se calcule en fonction des bits de tête du régime, c'est-à-dire la séquence de bits égaux au début de la représentation binaire. Soit m la longueur de cette séquence. Si les bits de tête sont égaux à 0, alors $k = -m$; s'ils sont égaux à 1, alors $k = m - 1$. Par exemple, pour un régime égal à 0001 en binaire, on a $k = -3$, pour $001x$, $k = -2$ et pour 1110, $k = 2$. La valeur du régime k apparait ensuite dans le facteur $useed^k$ dans l'évaluation du nombre posit, où $useed = 2^{2^{es}}$ avec es la taille de l'exposant. Dans les nombres posits, la taille du régime, de l'exposant et de la fraction sont donc variables ce qui permet de choisir le rapport souhaité entre plage de valeurs et précision.

Le format bfloat16 (`https://cloud.google.com/tpu/docs/bfloat16`) [**?**], quant à lui, se rapproche des flottants IEEE-754. C'est un format défini sur 16 bits dont les caractéristiques sont données dans le Tableau 1.5. Ce format permet d'avoir la même plage de données que le format binary32 (même taille d'exposant) tout en occupant moins d'espace (plus petite mantisse). Il est couramment utilisé en Intelligence Artificielle (IA), notamment au sein des TPU (*Tensor Processing Units*), puces d'accélération de réseaux de neurones spécialement conçues et optimisées pour l'entrainement et l'inférence de ces derniers. Notons que le standard C++23 (ISO/IEC 14882 :2023) a été introduit fin 2023 et permet d'utiliser le type `std::bfloat16_t` disponible dans la bibliothèque `<stdfloat>`. C++23 est compatible avec g++13.

Enfin, dans le même registre, il existe des formats définis sur 8 bits aussi à destination des réseaux de neurones [**?**] : E4M3 (4 bits pour l'exposant et 3 pour la mantisse) et E5M2 (5 bits pour l'exposant et 2 pour la mantisse) (voir Tableau 1.5). Le format E5M2 suit les conventions IEEE-754 concernant les nombres spéciaux définis en Section 1.1.1 tandis que le format E4M3 en fait fi pour étendre sa plage de valeur. Ces deux formats restent moins utilisés que les bfloat16.

format	taille	mantisse (bits)	exposant (bits)
bfloat16	16 bits	7	8
E4M3	8 bits	4	3
E5M2	8 bits	5	2

Tableau 1.5. Formats spécifiques

1.3 Erreurs d'arrondi

À chaque approximation d'un nombre réel par un nombre flottant, on perd a priori de la précision numérique à cause de l'arrondi effectué. On parle alors d'*erreur d'arrondi*. Celle-ci peut être très faible lors de l'approximation d'un nombre réel en flottant (ou même nulle, si ce nombre possède une représentation flottante exacte), mais l'accumulation de ces erreurs au fil des opérations d'un programme peut engendrer des erreurs importantes, avec un résultat numérique loin du résultat escompté.
Par exemple, en sommant n fois la valeur $\frac{1}{n}$, on peut voir que le résultat n'est pas exactement la valeur 1 avec le programme suivant.

```
#include <iostream>

int main()
{
    float val = 0.;
    for(int i=0; i<10; i++) {
        val += 0.1;
    }
    std::cout << 1-val << std::endl;
}
```

Listing 1.1 Exemple d'accumulation d'erreurs

Celui-ci calcule la somme de n fois la valeur $\frac{1}{n}$ pour $n = 10$, stockée dans la variable `val`, et nous donne comme résultat $1 - val =$ -1.19209e-07. De plus, en augmentant la valeur de n, cette erreur sur le résultat augmente, comme le montre le Tableau 1.6.

Valeur n	Erreur obtenue
100	6.55651e-07
1000	9.29832e-06
10000	-5.3525e-05
100000	9.9015e-04

Tableau 1.6. Exemple d'accumulation d'erreurs avec somme de n fois la valeur $1/n$

Outre l'accumulation d'arrondi, nous retrouvons deux types d'erreurs importantes liées à l'arrondi.
La première, l'absorption, apparait lors de l'addition (ou soustraction) de deux valeurs ayant des ordres de grandeurs suffisamment éloignés, ce qui rend la plus petite valeur négligeable, se faisant absorber par l'autre valeur. Nous pouvons prendre comme exemple le programme suivant faisant la somme de 2^{30} et 2, puis affichant le résultat si celui-ci est égal à 2^{30}.

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
    float a = 1 << 30;
    float b = 2.0;
    if(a+b==a){
        printf("%.1f+%.1f = %.1f", a, b, a);
    }
}
```

Listing 1.2 Exemple d'absorption lors du calcul de $2^{30} + 2$

Ce programme affiche le résultat suivant : `1073741824.0 + 2.0 = 1073741824.0`, ce qui est évidemment faux.

La seconde, l'annulation catastrophique, survient lors de la soustraction de deux nombres flottants proches entachés d'erreur, ce qui donne lieu à la perte de chiffres significatifs (par annulation), altérant ainsi le résultat. Le programme présenté dans le Listing 1.3 est un exemple donné par S. M. Rump [?] qui calcule le résultat du polynôme $P(x, y) = 9x^4 - y^4 + 2y^2$. Avec $x = 10864$ et $y = 18817$, la fonction retourne 2, alors que le résultat exact est 1. Cela est dû à deux annulations successives lors du calcul `a-b+c` effectué dans l'instruction `return` de la fonction `rump`. En effet, l'erreur d'arrondi introduite dans le calcul de `b = y*y*y*y` change le dernier chiffre du résultat de 1 à 0 ($1.2537228453050112\underline{0}e + 17$ au lieu de $1.2537228453050112\underline{1}e + 17$). Cet arrondi a peu d'impact sur le résultat de `a-b`, où l'on obtient -708158976.0 au lieu de -708158977.0, soit une erreur relative de l'ordre de 10^{-9}. Cependant, cela a beaucoup plus d'impact sur le résultat final, à cause de l'addition de `c = 2.0*y*y` qui a pour valeur 708158978.0, donnant une erreur relative égale à 1, soit 100%.

```c
#include <stdio.h>

double rump(double x, double y) {
    double a, b, c;
    a = 9.0*x*x*x*x;
    b = y*y*y*y;
    c = 2.0*y*y;
    return a-b+c;
}

int main(void)
    double x, y;
    x = 10864.0;
    y = 18817.0;
    printf("%f\n", rump(x, y));
}
```

Listing 1.3 Exemple d'annulation catastrophique lors du calcul de $P(10864, 18817)$ avec $P(x, y) = 9x^4 - y^4 + 2y^2$

1.4 Conclusion

Cette partie nous a permis d'introduire les nombres flottants et leur utilisation en machine, tels que définis par la norme IEEE-754. Bien qu'efficace, nous avons vu aussi les limites de cette représentation et les erreurs qui peuvent apparaître lors de l'utilisation des nombres flottants. Garder le contrôle sur ces erreurs est donc aussi devenu un objectif, notamment via les outils de validation numérique.

Validation numérique

2

Bien qu'en apparence anodine, les erreurs présentées précédemment peuvent se révéler fatales lors d'applications à grande échelle, notamment si celles-ci s'accumulent. L'accumulation d'erreurs d'arrondi ou la mauvaise considération des flottants a donné lieu à des bugs connus. C'est le cas par exemple du bug d'un missile Patriot de l'armée américaine qui échoua à intercepter un missile irakien en 1991 lors de la guerre du Golfe. En effet, l'horloge interne du système comptait le temps en 1/10 de seconde, ce qui ne possède pas de représentation binaire finie. Stocké en virgule fixe, 1/10 était donc arrondi à 24 bits par le système. Cet arrondi créa l'accumulation d'un léger décalage pour chaque 1/10 de seconde écoulé et entraîna une erreur d'environ 0.34s lorsque celui-ci dut intervenir, entrainant l'échec de son interception.
De ce fait, la validation numérique, consistant à trouver les sources d'erreurs et estimer leur impact sur la précision du résultat, est devenu un enjeu pour éviter la répétition de tels problèmes. Pour cela, de nombreux outils existent. Ceux-ci sont basés sur des méthodes d'analyse d'erreur, qui peuvent être soit statiques, soit dynamiques.
Parmi les méthodes dynamiques, on peut mentionner les méthodes probabilistes permettant d'estimer la précision du résultat, c'est-à-dire son nombre de chiffres corrects, par l'application d'une méthode d'estimation probabiliste comme les méthodes de Monte-Carlo. Nous utiliserons dans notre cas l'Arithmétique Stochastique Discrète (ASD), basée sur la méthode probabiliste CESTAC, et implémentée dans l'outil CADNA.

2.1 Différentes approches

2.1.1 Analyse directe et analyse inverse

L'analyse directe d'erreur consiste à majorer la distance entre un résultat calculé $\hat{y}$ et le résultat exact y, en considérant soit l'erreur absolue, $|\hat{y} - y|$, soit l'erreur relative, $|\hat{y} - y|/|y|$ avec $|y| \neq 0$. L'analyse inverse est une méthode statique très répandue, introduite par Wilkinson [**?**]. À l'inverse de l'analyse directe, l'analyse inverse ne cherche pas à quantifier l'écart entre le résultat calculé $\hat{y}$ et le résultat y, mais considère le résultat $\hat{y}$ comme le résultat exact obtenu à partir de données perturbées. En perturbant les entrées du programme, l'analyse inverse cherche à trouver le jeu de données tel que $\hat{y}$ soit le résultat exact. L'erreur inverse est ainsi la perturbation minimale sur les données qui fournit le résultat $\hat{y}$. En arithmétique flottante, si ces perturbations sont de l'ordre de $\mathbf{u}$, on parle alors de programme inverse stable.

Les deux notions d'erreur directe et inverse sont reliées par le conditionnement par la relation empirique suivante [?] :

$$\text{erreur directe} \leq \text{conditionnement} \times \text{erreur inverse}. \tag{2.1}$$

Le conditionnement représente la sensibilité du résultat vis-à-vis de perturbations des données en entrée. Plus celui-ci est grand, plus une perturbation des données affecte le résultat. Pour un conditionnement grand, on parle de problème mal conditionné.

Dans le cas où le problème serait inverse stable, l'équation 2.1 devient :

$$\text{erreur directe} \leq \text{conditionnement} \times \mathbf{u}. \tag{2.2}$$

Avec un conditionnement de l'ordre de $\mathbf{u}^{-1}$, la précision peut alors être de l'ordre de 1. Pour avoir un problème bien conditionné, il faut alors un conditionnement petit devant $\mathbf{u}^{-1}$. L'analyse inverse est par exemple utilisée dans la bibliothèque LAPACK [?]. L'un des principaux avantages de LAPACK est qu'il fournit des limites d'erreur pour toutes les quantités calculées.

2.1.2 Arithmétique par intervalles

Une deuxième méthode couramment utilisée est l'arithmétique par intervalles. Celle-ci consiste à effectuer les opérations non pas avec des nombres, mais avec des intervalles. Un nombre x est représenté par un intervalle $X = [\underline{x}, \overline{x}]$ incluant x. Les principaux opérateurs arithmétiques définis pour deux intervalles $X = [\underline{x}, \overline{x}]$ et $Y = [\underline{y}, \overline{y}]$ sont :

- $X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$;
- $X - Y = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$;
- $-X = [-\overline{x}, -\underline{x}]$;
- $X \times Y = [\min(\underline{x} \times \underline{y}, \overline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \overline{y}), \max(\underline{x} \times \underline{y}, \overline{x} \times \underline{y}, \underline{x} \times \overline{y}, \overline{x} \times \overline{y})]$;
- $Y^{-1} = [\min(1/\underline{y}, 1/\overline{y}), \max(1/\underline{y}, 1/\overline{y})]$ si $0 \notin [\underline{y}, \overline{y}]$;
- $X/Y = [\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]^{-1}$ si $0 \notin [\underline{y}, \overline{y}]$.

Les opérations sur les intervalles sont définies de telle sorte que, pour X et Y deux intervalles, $\diamond$ une opération, l'intervalle résultant $X \diamond Y$ vérifie

$$\forall x \in X, \forall y \in Y, x \diamond y \in X \diamond Y. \tag{2.3}$$

Pour que cette proposition soit vraie aussi en arithmétique flottante, il faut utiliser un mode d'arrondi dirigé afin de prendre en compte l'erreur d'arrondi : un mode d'arrondi vers $-\infty$ pour la borne inférieure du résultat, et un mode d'arrondi vers $+\infty$ pour la borne supérieure. L'utilisation de l'arithmétique par intervalles, telle que définie précédemment, entraine une perte de performance importante puisqu'il faut par exemple au moins 2 opérations flottantes pour l'addition de deux intervalles (borne supérieure et borne inférieure), et 8 opérations flottantes (quatre opérations arithmétiques effectuées avec deux modes d'arrondi différents),

plus l'application des fonctions $\min$ et $\max$, pour une multiplication. De plus, bien que l'arithmétique par intervalles garantisse des bornes exactes pour le résultat calculé, elle a aussi tendance à surestimer ces bornes. Cette surestimation provient de plusieurs facteurs :

— élargissement naturel des intervalles due à l'arithmétique flottante ;
— décorrélation de variables : par exemple pour un intervalle $X = [0, 5]$, $X - X = [0 - 5, 5 - 0] = [-5, 5]$, au lieu de $[0, 0]$;
— le phénomène de *wrapping* apparaissant lorsque l'image d'un intervalle multidimensionnel est un intervalle unidimensionnel.

Plusieurs bibliothèques et logiciels implémentent l'arithmétique par intervalles, comme C-XSC [?] pour du code C++, et INTLAB (INTerval LABoratory) [?] (https://www.tuhh.de/ti3/intlab/) implémenté dans MATLAB.

2.1.3 Analyse par interprétation abstraite

L'analyse par interprétation abstraite se base sur l'analyse sémantique d'un programme. Par exemple, la sémantique des traces considère les traces possibles d'exécutions $\sigma_0\sigma_1\ldots$ (possiblement infinies) du programme, avec σ_i chaque état du programme. Un état σ_i comprend toutes les informations caractéristiques à un instant t (variables, état de la mémoire, horloge interne, etc.). La définition d'une spécification décrivant les exécutions souhaitées du programme pour la sémantique considérée permet de le vérifier. L'interprétation abstraite est souvent une approximation de la sémantique réelle, permettant de vérifier certains aspects du programme, mais en en omettant d'autres.

Des outils comme Fluctuat [?] (http://www.lix.polytechnique.fr/~putot/fluctuat.html) et FLDLib (https://github.com/fvedrine/fldlib) sont basés sur l'interprétation abstraite. Ils permettent d'obtenir la différence entre le résultat et un résultat de référence calculé dans une sémantique idéalisée des nombres réels. Cela leur permet de relever les sources d'instabilités numériques et d'évaluer les erreurs d'arrondi ainsi que la sensibilité aux paramètres du programme.

2.1.4 Méthodes probabilistes

Les méthodes probabilistes consistent à analyser l'erreur grâce au calcul de plusieurs résultats. Une perturbation aléatoire est ajoutée à ces résultats, formant ainsi un échantillon pour l'analyse. Ces différents résultats peuvent être considérés comme formant une variable aléatoire, ce qui permet l'application de techniques statistiques pour évaluer et comprendre l'erreur. Pour cela, l'arrondi aléatoire est par exemple souvent utilisé. Celui-ci consiste à arrondir le résultat de chaque opération vers $+\infty$ ou $-\infty$ avec la même probabilité $1/2$.

Les méthodes probabilistes sont utilisées dans le cadre du HPC, car, contrairement aux méthodes présentées précédemment, celles-ci peuvent être appliquées à des codes de grande taille. Nous utilisons dans notre cas l'Arithmétique Stochastique Discrète.

2.2 Arithmétique Stochastique Discrète et son implémentation

L'Arithmétique Stochastique Discrète (ASD) est une méthode d'analyse des erreurs d'arrondi basée sur la méthode CESTAC [**?**, **?**].

2.2.1 Méthode CESTAC

La méthode CESTAC (Controle et Estimation STochastique des Arrondis de Calculs) permet d'estimer la propagation des erreurs d'arrondi qui se produisent lors de calculs avec des nombres à virgule flottante. Basée sur une approche probabiliste, elle utilise un mode d'arrondi aléatoire : à chaque opération, le résultat est arrondi vers $+\infty$ ou vers $-\infty$ avec la même probabilité. En utilisant ce mode d'arrondi, le même programme est exécuté N fois, ce qui nous donne un échantillon de taille N $\{R_1, \ldots, R_N\}$ du résultat calculé. Ces éléments $R_1, \ldots, R_N$ peuvent être considérés comme une distribution de variable aléatoire Gaussienne [**?**]. Le résultat final devient ainsi $\bar{R}$, la moyenne des R_i.
De plus, la précision de $\bar{R}$, c'est-à-dire son nombre de chiffres significatifs exacts, noté $C_{\bar{R}}$, peut être estimée par application de la loi de Student :

$$C_{\bar{R}} = \log_{10}\left(\frac{\sqrt{N}|\bar{R}|}{\tau_\beta \sigma}\right) \tag{2.4}$$

où

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}(R_i - \bar{R})^2,$$

et τ_β est le quantile de la loi de Student pour $N-1$ degrés de liberté et un niveau de probabilité $1-\beta$.

En pratique, $\beta = 5\%$ pour un taux de confiance à 95%, et la taille de l'échantillon est de $N = 3$. En effet, il a été montré que $N = 3$ est, dans une certaine mesure, la valeur optimale : elle est à la fois plus fiable qu'avec $N = 2$ et l'augmentation de la taille de l'échantillon n'améliore pas la qualité de l'estimation.

La méthode CESTAC appliquée au résultat final d'un programme peut s'avérer invalide lorsqu'un résultat intermédiaire est lui-même non-significatif, c'est-à-dire que l'erreur d'arrondi du résultat est du même ordre de grandeur que le résultat lui-même. Cette notion correspond à la notion de *zéro informatique* définie ci-dessous (Définition 2.2.1). Pour contrer cela, la méthode CESTAC doit être appliquée de manière synchrone afin de relever à chaque opération la précision du résultat (*auto-validation* de la méthode). L'utilisateur est ainsi alerté lors de l'apparition d'un résultat non-significatif.

2.2.2 Arithmétique Stochastique Discrète (ASD)

L'Arithmétique Stochastique Discrète correspond à l'application de la méthode CESTAC de manière synchrone, combinée à la notion de *zéro informatique* et aux relations stochastiques discrètes. Un nombre réel devient donc un N-uplet, et les opérations entre N-uplets se font élément par élément en utilisant le mode d'arrondi aléatoire. Cela permet d'estimer le nombre de chiffres significatifs exacts d'un nombre grâce à l'équation 2.4.
Afin de vérifier au cours de l'application si un résultat est significatif, l'ASD utilise la notion de *zéro informatique* définie ci-dessous.

Définition 2.2.1 (Zéro informatique). Un résultat $\bar{R}$ est un *zéro informatique*, noté $@.0$, si et seulement si :

$$\forall i, R_i = 0 \quad \text{ou} \quad C_{\bar{R}} \leq 0.$$

Un *zéro informatique* est donc soit le zéro mathématique, soit un nombre non-significatif. Afin de conserver la cohérence avec les opérateurs arithmétiques et relationnels, nous définissons les relations stochastiques discrètes suivantes.

Définition 2.2.2. Soient X et Y deux résultats calculés avec la méthode CESTAC.

(1) $X = Y \ ssi \ X - Y = @.0$;

(2) $X > Y \ ssi \ \bar{X} > \bar{Y}$ et $X - Y \neq @.0$;

(3) $X \geq Y \ ssi \ \bar{X} \geq \bar{Y}$ ou $X - Y = @.0$.

Ces opérateurs relationnels stochastiques sont primordiaux afin d'assurer le bon déroulement d'un programme informatique. En effet, pour X et Y deux résultats informatiques, et x et y les résultats exacts correspondants, on a

$$x > y \nRightarrow X > Y \text{ et } X > Y \nRightarrow x > y.$$

Cela peut causer des problèmes au sein de programmes scientifiques, par exemple avec des tests d'arrêts non vérifiés. L'utilisation de relations prenant en compte les imprécisions numériques permet de limiter l'apparition de tels problèmes.

2.2.3 CADNA

CADNA (Control of Accuracy and Debugging for Numerical Applications) [**?**, **?**, **?**] est un outil de validation numérique implémentant l'Arithmétique Stochastique Discrète. En pratique, la bibliothèque CADNA fournit des types stochastiques (ex : double_st pour double stochastique) en C, C++ et Fortran. Ces types permettent de stocker 3 valeurs flottantes pour une même variable, plus un entier correspondant à la précision, i.e. le nombre de chiffres significatifs exacts. CADNA redéfinit aussi les opérations arithmétiques ($+$, $-$, $\times$, $\div$), les relations d'ordre ($\leq$, $<$, $\geq$, $>$, $==$) ainsi que les fonctions mathématiques ($\cos$, $\sin$, $\exp$, etc.) pour les nombres stochastiques. L'affichage des nombres stochastiques, soit la moyenne des trois valeurs stockées, avec le nombre de chiffres significatifs corrects estimé avec un taux de confiance de 95% (Section 2.2.1) est réalisé avec la fonction `strp()` renvoyant une chaîne de caractères.

Pour utiliser CADNA, il suffit à l'utilisateur de déclarer ses variables avec des types stochastiques, après inclusion de la bibliothèque CADNA (`cadna.h` en C/C++). L'utilisateur doit aussi appeler les fonctions `cadna_init(numb_instability, cadna_instability, cancel_level, init_random)` en début de code et `cadna_end()` en fin de code. Le premier paramètre de la fonction `cadna_init()`, `numb_instability`, est obligatoire et permet de préciser le nombre d'instabilités que l'on souhaite détecter. Initialisé à -1, il active la détection de toutes les instabilités numériques. Initialisé à 0 en revanche, il désactive la détection de toutes les instabilités numériques. Cela permet surtout d'évaluer le surcoût des opérations surchargées. Ces deux valeurs sont les plus utilisées. Les autres paramètres sont optionnels. Le second, `cadna_instability` permet de préciser le type d'instabilité que l'on souhaite détecter. Par défaut, toutes les formes d'instabilités sont détectées. Le troisième, `cancel_level` permet de préciser le niveau de détection d'une annulation lors d'une opération. Enfin, le dernier, `init_random` permet d'initialiser une variable interne à CADNA utilisée pour l'arrondi aléatoire. La fonction `cadna_end()` permet de "fermer" la bibliothèque et d'afficher sur la sortie standard le résultat de la détection des instabilités numériques.

Un exemple de code est donné avec et sans CADNA dans les Listings 2.1 et 2.2 (exemple tiré de la documentation CADNA https://cadna.lip6.fr). Ces codes calculent les racines d'un polynôme du second degré. Leurs sorties respectives sont données dans les Listings 2.3 et 2.4. Nous pouvons voir que le calcul en flottant sans CADNA ne permet pas d'obtenir le bon résultat pour le déterminant, $d = 0$, car le mauvais branchement est exécuté. À l'inverse, CADNA, en prenant en compte la précision des opérandes dans les relations d'ordre ou d'égalité, permet de détecter que le résultat est un zéro informatique ($d = @.0$), et d'obtenir le bon résultat.

```c
#include <stdio.h>
#include <math.h>

int main()
{
  float a = 0.3;
  float b = -2.1;
  float c = 3.675;
  float d, x1,x2;

  //      CASE : A = 0
  if (a==0.)
    if (b==0.) {
      if (c==0.) printf("Every complex value is solution.\n");
      else printf("There is no solution.\n");
    }
    else {
      x1 = - c/b;
      printf("The equation is degenerated.\n");
      printf("There is one real solution %+.6e\n",x1);
    }
  else {
    //      CASE : A /= 0
    b = b/a;
    c = c/a;
```

```
    d = b*b - 4.0*c;
    printf("d = %+.6e\n",d);
    //   DISCRIMINANT = 0
    if (d==0.) {
      x1 = -b*0.5;
      printf("Discriminant is zero.\n");
      printf("The double solution is %+.6e\n",x1);
    }
    else {
      //       DISCRIMINANT > 0
      if (d>0.) {
          x1 = ( - b - sqrtf(d))*0.5;
          x2 = ( - b + sqrtf(d))*0.5;
      printf("There are two real solutions.\n");
      printf("x1 = %+.6e x2 = %+.6e\n",x1,x2);
      }
      else {
    //       DISCRIMINANT < 0
    x1 = - b*0.5;
    x2 = sqrtf(-d)*0.5;
    printf("There are two complex solutions.\n");
    printf("z1 = %+.6e  +  i * %+.6e\n",x1,x2);
    printf("z2 = %+.6e  +  i * %+.6e\n",x1, -x2);
      }
    }
  }
  printf("The exact discriminant value is 0.\n");
  return 0;
}
```

Listing 2.1 Code source de calcul des racines d'un polynôme du second degré

```
#include <stdio.h>
#include <math.h>
#include <cadna.h>

using namespace std;
int main()
{

  cadna_init(-1);

  float_st a = 0.3;
  float_st b = -2.1;
  float_st c = 3.675;
  float_st d, x1,x2;

  //      CASE: A = 0
  if (a==0)
    if (b==0.) {
      if (c==0.) printf("Every complex value is solution.\n");
      else printf("There is no solution.\n");
    }
    else {
      x1 = - c/b;
      printf("The equation is degenerated.\n");
```

```
      printf("There is one real solution %s\n",strp(x1));
    }
  else {
    //     CASE: A /= 0
    b = b/a;
    c = c/a;
    d = b*b - 4.0*c;
    printf("d = %s\n",strp(d));
    //   DISCRIMINANT = 0
    if (d==0.) {
      x1 = -b*0.5;
      printf("Discriminant is zero.\n");
      printf("The double solution is %s\n",strp(x1));
    }
    else {
      //      DISCRIMINANT > 0
      if (d>0.) {
          x1 = ( - b - sqrtf(d))*0.5;
          x2 = ( - b + sqrtf(d))*0.5;
      printf("There are two real solutions.\n");
      printf("x1 = %s x2 = %s\n",strp(x1),strp(x2));
      }
      else {
    //      DISCRIMINANT < 0
    x1 = - b*0.5;
    x2 = sqrtf(-d)*0.5;
    printf("There are two complex solutions.\n");
    printf("z1 = %s  +  i * %s\n",strp(x1),strp(x2));
    printf("z2 = %s  +  i * %s\n",strp(x1), strp(-x2));
      }
    }
  }

  cadna_end();
}
```

Listing 2.2 Code CADNA de calcul des racines d'un polynôme du second degré

```
----------------------------------
|  Second order equation         |
|  without CADNA                 |
----------------------------------
d = -3.814697e-06
There are two complex solutions.
z1 = +3.500000e+00  +  i * +9.765625e-04
z2 = +3.500000e+00  +  i * -9.765625e-04
```

Listing 2.3 Sortie du calcul des racines d'un polynôme sans CADNA

```
-----------------------------------------------------------------
CADNA_C_HALF 3.1.11 software
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
```

```
Overflow detection for half-precision: ON
Underflow detection for half-precision: ON
----------------------------------------------------------------
----------------------------------
|  Second order equation          |
|  with CADNA                     |
----------------------------------
d =  @.0
Discriminant is zero.
The double solution is  0.3499999E+001
----------------------------------------------------------------
CADNA_C_HALF 3.1.11 software
There is  1 numerical instability
1 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)
----------------------------------------------------------------
```

Listing 2.4 Sortie du calcul des racines d'un polynôme avec CADNA

L'application de CADNA à un programme entraîne un temps d'exécution au minimum 3 fois supérieur au programme original. Au départ bien supérieur, ce surcoût a été réduit au travers des différentes versions de CADNA permettant l'application sur des codes HPC. CADNA a aussi été rendu applicable sur des codes vectorisés [?]. De la même manière, la bibliothèque est désormais applicable sur des codes MPI [?] et OpenMP [?], ou encore sur GPU [?].
De plus, la bibliothèque CADNA a été étendue et rendue applicable sur les variables en précision quadruple [?] ainsi qu'en précision half [?]. La précision half peut être soit native sur les processeurs qui la supportent, soit émulée à l'aide d'une bibliothèque développée par C. Rau (http://half.sourceforge.net), utilisée dans la suite de ce document. Notons enfin que l'Arithmétique Stochastique Discrète est aussi implémentée dans l'outil SAM (*Stochastic Arithmetic in Multiprecision*) [?] pour les formats en précision arbitraire.

2.3 Autres outils probabilistes de validation numérique

Cette section présente à titre d'exemples les outils VERROU et VERIFICARLO. À la différence de CADNA, qui utilise une approche synchrone, ces deux outils utilisent une approche asynchrone où une exécution donne un résultat perturbé. Il faut alors pour ces outils plusieurs exécutions afin d'obtenir l'ensemble des résultats perturbés, alors qu'une seule est suffisante à CADNA pour détecter les instabilités numériques.

2.3.1 VERROU

L'outil VERROU [?] (http://github.com/edf-hpc/verrou) est fondé sur Valgrind (https://valgrind.org). Il modifie automatiquement chaque instruction flottante pour renvoyer un résultat arrondi vers $+\infty$ ou $-\infty$ de manière aléatoire, au lieu d'arrondir au plus près par défaut. Cette "Arithmétique en Arrondi Aléatoire" (AAA), qui peut être vue comme la méthode CESTAC asynchrone, permet à VERROU d'obtenir des résultats globaux affectés par l'accumulation des perturbations. Ces résultats permettent d'évaluer la qualité numérique

du code. De plus, en ne perturbant qu'une partie du programme, VERROU peut par la suite localiser de potentielles erreurs. Une seconde méthode pour localiser de potentielles erreurs consiste à comparer deux passes du programme perturbées par l'AAA afin de trouver les branchements (par exemple un "`if`") donnant des comportements différents. Un des principaux avantages de VERROU est sa compatibilité avec les outils de développement (mêmes avantages que Valgrind : pas besoin de recompiler, pas besoin d'un accès au code source, etc.) et avec les langages (C, C++, Fortran, Python, assembleur, etc.).

2.3.2 VERIFICARLO

L'outil VERIFICARLO [?] (`http://www.github.com/verificarlo/verificarlo`) est intégré à LLVM et permet d'utiliser l'arithmétique de Monte-Carlo (MCA) [?] de manière automatique pour des codes C, C++ et Fortran. Cette arithmétique peut introduire des perturbations sur les opérandes et sur les résultats d'opérations arithmétiques. Ces différents résultats peuvent être considérés comme formant une variable aléatoire ce qui permet d'estimer leurs précisions par méthode de Monte-Carlo. Pour cela, l'outil instrumente directement la Représentation Intermédiaire (IR) LLVM du code source, qui est indépendante du langage source, et qui se situe après la compilation, permettant ainsi de conserver les optimisations (ce qui inclut les optimisations d'opérations flottantes comme `-ffast-math` ou `-freciprocal-math`).
Le principal défaut que l'on peut noter est que l'arithmétique de Monte-Carlo nécessite un nombre d'échantillons assez large par rapport à l'Arithmétique Stochastique Discrète, la rendant plus coûteuse en temps.

2.4 Conclusion

Les outils de validation numérique représentent aujourd'hui un élément important dans la chaîne de validation d'un code scientifique. Nous avons vu que différentes approches existent, présentant des avantages différents. Nous avons en particulier introduit l'Arithmétique Stochastique Discrète et l'outil CADNA qui l'implémente. Nous utiliserons dans la suite l'outil CADNA dans le cadre de l'auto-ajustement de la précision afin d'obtenir une version des programmes en précision mixte tout en respectant un critère de précision sur les résultats.

Auto-ajustement de la précision

3

L'auto-ajustement (ou *auto-tuning*) de la précision d'un programme consiste à trouver le plus petit format/type utilisable de manière automatique tout en respectant une contrainte de précision sur le résultat final, à laquelle peut s'ajouter une contrainte sur la performance du programme. L'objectif est d'obtenir de meilleures performances, que ce soit en mémoire ou en temps d'exécution. En effet, utiliser une précision plus faible prend moins de place et certaines opérations arithmétiques et de lecture/écriture en mémoire sont plus rapides en précision plus faible d'un facteur proportionnel à la taille du type. Bien souvent, l'auto-ajustement de la précision donne une version du programme comportant différents types pour chaque variable, on parle de précision mixte.

Pour réaliser cela, différentes approches existent. Certaines consistent à modifier le type des variables [**?**, **?**, **?**], d'autres à modifier le type des opérations [**?**], voire à passer les opérations entre types flottants à des opérations entre types à virgules fixes [**?**]. Ces approches peuvent se distinguer comme précédemment par leur implémentation soit statique soit dynamique au sein de différents outils.
Le document *Tools for Reduced Precision Computation : A Survey* [**?**] offre une vue d'ensemble de l'état de l'art sur l'auto-ajustement de la précision, identifiant notamment les outils les plus importants. Parmi eux, nous pouvons citer FPTuner [**?**], Salsa [**?**], Rosa/Daisy [**?**, **?**], TAFFO [**?**] ou encore POP [**?**] comme outils ayant une approche statique. Ceux-ci sont difficilement applicables sur des codes industriels à grande échelle. Au contraire, les outils dynamiques, parmi lesquels nous pouvons citer CRAFT HPC [**?**], Precimonious [**?**], HiFPTuner [**?**], ADAPT [**?**], FloatSmith [**?**] et PROMISE [**?**], ont comme objectif d'être appliqués à des codes HPC. Nous présenterons certains de ces outils à titre d'exemples dans la section 3.3. L'outil PROMISE, qui sera l'outil utilisé dans la suite de ce document, est présenté en section 3.2. Notons qu'il existe aussi des outils pour l'auto-tuning de précision sur GPU tels que AMPT-GA [**?**], GPUMixer [**?**] ou GRAM [**?**].

Nombre de ces outils dynamiques utilisent une approche "essai-et-erreur", notamment par le biais de l'algorithme du Delta-Debug, utilisé aussi dans notre outil PROMISE. Nous présentons ci-dessous l'algorithme du Delta-Debug ainsi que son utilisation au sein de PROMISE pour créer les différentes configurations associant un type à chaque variable.

3.1 Delta-Debug

L'algorithme du Delta-Debug [**?**, **?**] est une méthode de débogage permettant d'isoler la source de l'erreur au sein d'un programme en considérant un ensemble de modifications apportées à celui-ci.

Considérons un programme P auquel nous apportons des changements pour obtenir une nouvelle version P'. On considère que P est notre programme de base, qui fonctionne, tandis que P' ne fonctionne pas. Notons $C = \{\Delta_1, \Delta_2, ..., \Delta_n\}$ l'ensemble des changements. Chaque sous-ensemble $c \subseteq C$ est appelé une configuration. La configuration $c = \emptyset$ correspond à aucun changement appliqué, et donc au programme P. Il y a 2^n configurations possibles. Pour déterminer si une configuration engendre une erreur, nous considérons une fonction $test$ donnant deux résultats possibles :

— *pass*, noté ✓, qui indique que la configuration n'engendre pas d'erreur ;

— *fail*, noté ✗, qui indique que la configuration engendre une erreur.

Remarque 3.1.1. Dans le cas général, les fonctions $test$ pour le Delta-Debug peuvent aussi avoir comme résultat *unresolved*, cependant ce résultat n'est pas utilisé dans notre cas, et nous détaillons l'algorithme ne prenant pas en compte cette possibilité.

Notons que l'on a : $test(\emptyset) =$ ✓ et $test(C) =$ ✗.

Définition 3.1.2 (Ensemble induisant l'échec)**.** On appelle ensemble induisant l'échec un ensemble de changements $c \subseteq C$ tel que :

$$\forall c' \subseteq C (c \subseteq c' \Rightarrow test(c') = \text{✗}).$$

En pratique, l'algorithme va créer une à une les différentes configurations possibles, générer le programme associé et le tester. Le but est de trouver un ensemble minimal induisant l'échec, c'est-à dire un ensemble tel que retirer un changement de cet ensemble n'induit plus l'échec.

Définition 3.1.3 (Ensemble minimal induisant l'échec)**.** Un ensemble induisant l'échec $c \subseteq C$ est minimal si

$$\forall c' \subset c, test(c') = \text{✓}.$$

Si l'ensemble C des changements est de taille 1 ($|C| = 1$) alors nous avons directement trouvé un ensemble minimal (unique dans ce cas). Sinon, C est coupé en deux sous-ensembles c_1, c_2 de taille à peu près égale à $|C|/2$ qui sont testés séparément. Cela donne trois sorties possibles :

— c_1 ne passe pas, il contient donc un sous-ensemble induisant l'échec (Situation 1) ;

— c_2 ne passe pas, il contient donc un sous-ensemble induisant l'échec (Situation 2) ;

— c_1 et c_2 passent, le sous-ensemble induisant une erreur est donc une intersection de sous-ensembles de c_1 et c_2 (Situation 3).

Pour les deux premières situations, il suffit de continuer à chercher un ensemble minimal au sein du sous-ensemble c_1 ou c_2 qui ne passe pas. Le Tableau 3.1 présente un exemple dans le cas où c_2 ne passe pas, et dans lequel les nombres i correspondent aux changements Δ_i, et les points dénotent un Δ_i non appliqué dans la configuration.

Étape	c_i	configuration						test
1	c_1	1	2	3	.	.	.	✓
2	c_2	.	.	.	4	5	6	✗
3	c_1	.	.	.	4	5	.	✓
4	c_2	.	.	.	.	.	6	✗
Résultat		.	.	.	.	.	6	

Tableau 3.1. Exemple de configurations testées par le Delta-Debug - Situation 2

Pour la dernière situation (c_1 et c_2 passent), il faut aussi tester des sous-ensembles qui sont une intersection de sous-ensembles de c_1 et c_2, comme présenté dans le Tableau 3.2.

Étape	c_i	configuration						test
1	c_1	1	2	3	.	.	.	✓
2	c_2	.	.	.	4	5	6	✓
3	c_1	1	2	.	4	5	6	✓
4	c_2	.	.	3	4	5	6	✗
5	c_1	1	2	3	4	5	.	✓
6	c_2	1	2	3	.	.	6	✗
Résultat		.	.	3	.	.	6	

Tableau 3.2. Exemple de configurations testées par le Delta-Debug - Situation 3

Nous pouvons alors présenter l'algorithme de recherche du Delta-Debug tel que donné par [**?**, **?**]. La fonction $dd(c)$ renvoie tous les changements dans c induisant des échecs. Nous notons r l'ensemble des changements qui restent appliqués.

Définition 3.1.4 (Algorithme du Delta-Debug)**.** L'algorithme du Delta-Debug est $dd(c)$ tel que :
$dd(c) = dd_2(c, \emptyset)$,

et, avec $c_1, c_2 \subseteq c$ tels que $c_1 \cup c_2 = c$, $c_1 \cap c_2 = \emptyset$, $|c_1| \approx |c_2| \approx |c|/2$,

$$dd_2(c, r) = \begin{cases} c & \text{si } |c| = 1 \\ dd_2(c_1, r) & \text{si } test(c_1 \cup r) = ✗ \\ dd_2(c_2, r) & \text{si } test(c_2 \cup r) = ✗ \\ dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{sinon.} \end{cases}$$

Nous pouvons remarquer que l'algorithme du Delta-Debug n'est pas exhaustif, i.e. en prenant $n = |C|$, il ne teste pas les 2^n configurations possibles, mais permet d'obtenir avec une complexité moyenne en $\mathcal{O}(n \log n)$, un ensemble minimal induisant un échec.

L'algorithme du Delta-Debug est appliqué dans le cadre de l'auto-tuning de précision afin de tester de la même manière différentes configurations en considérant cette fois les différentes précisions possibles pour chaque variable. Une configuration est donc une fonction qui à chaque variable associe un des types disponibles. Dans cette situation, l'algorithme cherche un ensemble maximal n'induisant pas l'échec, i.e., par opposition à la définition 3.1.3, un ensemble de variables pouvant passer dans la précision la plus faible possible dans lequel

rajouter une variable induirait l'échec. La fonction *test* devient une fonction comportant une contrainte sur la précision du résultat (celle-ci peut être affinée ou comporter d'autres contraintes selon les besoins et situations).

L'utilisation du Delta-Debug dans PROMISE, que nous présentons par la suite, est décrite dans la Figure 3.1. Nous commençons avec un ensemble de variables dans la précision la plus élevée qui passe notre test sur la précision du résultat, puis nous essayons de réduire la précision d'un maximum de variables en prenant des sous-ensembles les plus grands possibles. Lorsqu'un sous-ensemble passe notre test sur la précision, nous essayons de compléter celui-ci avec un maximum d'autres variables, au lieu d'aller tester les autres sous-ensembles possibles de même taille. Nous obtenons ainsi un sous-ensemble maximal de variables pouvant prendre la précision la plus faible tout en satisfaisant le test sur la précision du résultat. Encore une fois, l'algorithme n'étant pas exhaustif, il ne s'agit possiblement pas de la meilleure configuration possible, mais nous privilégions une complexité plus faible, donc un temps d'exécution plus faible.

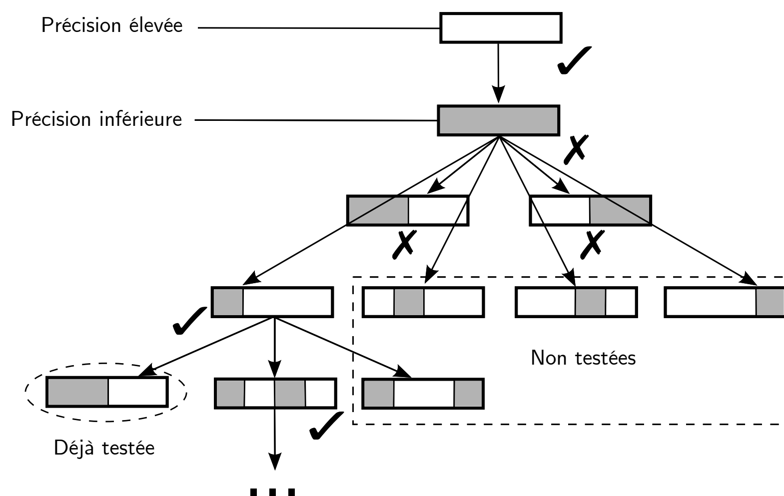

Figure 3.1. Delta-Debug pour l'auto-tuning de précision avec deux types différents

3.2 PROMISE

PROMISE (PRecision OptiMISE) [?] (http://promise.lip6.fr) est un outil d'auto-tuning de précision. À partir d'un code C/C++ initial et d'une précision requise sur le résultat, il retourne un code en précision mixte, réduisant la précision d'un ensemble maximal de variables (dans le sens opposé à la définition 3.1.3), tout en conservant un résultat qui satisfait la contrainte sur la précision.

Pour ce faire, PROMISE prend en entrée un code instrumenté où certaines variables sont déclarées avec des types particuliers que PROMISE reconnaît : ils sont de la forme "`__PROMISE__`", forme générique, ou "`__PR_xx__`", où "xx" permet de spécifier un type personnalisé qui peut être réutilisé pour plusieurs variables. En effet, celles-ci peuvent ainsi être forcées à avoir la même précision. Cela peut être utile pour éviter les erreurs de compilation

ou des casts de variables. On appellera parfois les variables déclarées avec un "type PROMISE" les "variables PROMISE".

Le Listing 3.1 présente un exemple de code qui calcule la longueur d'un arc de courbe, avec sa version instrumentée dans le Listing 3.2. On peut voir l'utilisation du type PROMISE générique "`__PROMISE__`", ainsi que des types PROMISE spécifiques comme "`__PR_fun__`", qui forcera donc la variable `t1` à avoir la même précision que le retour de la fonction `fun`. La fonction "`PROMISE_CHECK_VAR(var)`" est une macro qui transmet la variable `var` passée en paramètre à PROMISE afin d'en tester sa précision. Sa version pour les tableaux, "`PROMISE_CHECK_ARRAY(var, size)`", prend en second argument la taille du tableau.

```
double fun(double x){
  int k, n = 5;
  double t1;
  double d1 = 1.0;

  t1 = x;
  for ( k = 1; k <= n; k++ )
    {
      d1 = 2.0 * d1;
      t1 = t1+ sin(d1 * x)/d1;
    }
  return t1;
}

int main( int argc, char **argv) {

  int i,n = 1000000;
  double h;
  double t1, t2, dppi;
  double s1;
  std::ofstream res;
  std::cout.precision(15);

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for ( i = 1; i <= n; i++)
    {
      t2 = fun(i * h);
      s1 = s1 + sqrt(h*h + (t2 - t1) * (t2 - t1));
      t1 = t2;
    }

  std::cout << s1 << std::endl;

  return 0;
}
```

Listing 3.1 Exemple de code pour PROMISE (http://promise.lip6.fr/examples.html)

```
__PR_fun__ fun(__PR_1__ x){
  int k, n = 5;
  __PR_fun__ t1;
  __PR_1__ d1 = 1.0;

  t1 = x;
  for ( k = 1; k <= n; k++ )
    {
      d1 = 2.0 * d1;
      t1 = t1+ sin(d1 * x)/d1;
    }
  return t1;
}

int main( int argc, char **argv) {

  int i,n = 1000000;
  __PR_1__ h;
  __PROMISE__ t1, t2, dppi;
  __PROMISE__ s1;
  std::ofstream res;
  std::cout.precision(15);

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for ( i = 1; i <= n; i++)
    {
      t2 = fun(i * h);
      s1 = s1 + sqrt(h*h + (t2 - t1) * (t2 - t1));
      t1 = t2;
    }

  std::cout << s1 << std::endl;
  PROMISE_CHECK_VAR(s1);
  return 0;
}
```

Listing 3.2 Code instrumenté pour PROMISE (promise.lip6.fr/examples.html)

PROMISE utilise ensuite l'algorithme du Delta-Debug comme présenté précédemment dans la Figure 3.1 pour trouver de manière efficace un des plus grands ensembles de variables pouvant passer en précision plus faible. Pour sa fonction test, PROMISE calcule un résultat de référence en utilisant la bibliothèque CADNA présentée précédemment. Cela permet d'avoir un résultat dont on peut estimer la précision grâce à l'Arithmétique Stochastique Discrète. En cela PROMISE se distingue des autres outils qui considèrent comme résultat de référence un résultat calculé dans la plus grande précision utilisée (souvent double). Les différentes configurations sont testées en créant le code correspondant, en l'exécutant et en comparant le nombre de chiffres en commun entre le résultat obtenu pour la configuration

et le résultat de référence.

Le Listing 3.3 montre le résultat de l'application de PROMISE sur l'exemple présenté précédemment en demandant 5 chiffres de précision sur le résultat. Nous pouvons voir que cela donne un code en précision mixte, les différentes variables PROMISE ayant pris une précision soit double, soit simple. En particulier, les déclarations avec le même type PROMISE spécifique (`fun` et `t1`, et `d1` et `h`) ont pris la même précision, alors que ce n'est pas le cas des variables avec un type PROMISE générique.

```
double fun(double x){
 int k, n = 5;
 double t1;
 double d1; d1= 1.0;

 t1 = x;
 for ( k = 1; k <= n; k++ )
   {
     d1 = 2.0 * d1;
     t1 = t1+ sin(d1 * x)/d1;
   }
 return t1;
}

int main( int argc, char **argv) {

 int i,n = 1000000;
 double h;
 double t1;double t2;float dppi;
 double s1;
 std::ofstream res;
 std::cout.precision(15);

 t1 = -1.0;
 dppi = acos(t1);
 s1 = 0.0;
 t1 = 0.0;
 h = dppi / n;

 for ( i = 1; i <= n; i++)
   {
     t2 = fun(i * h);
     s1 = s1 + sqrt(h*h + (t2 - t1) * (t2 - t1));
     t1 = t2;
   }

 std::cout << s1 << std::endl;

 return 0;
}
```

Listing 3.3 Résultat de PROMISE avec 4 chiffres de précision demandée

PROMISE est utilisé principalement en considérant les trois précisions flottantes half, simple et double, définies précédemment dans le Tableau 1.2. Le fonctionnement global de

PROMISE est résumé dans la Figure 3.2. Avec ces trois types, PROMISE utilise deux fois le Delta-Debug : la première fois pour passer de la précision double à simple, la seconde pour passer de la précision simple à half.

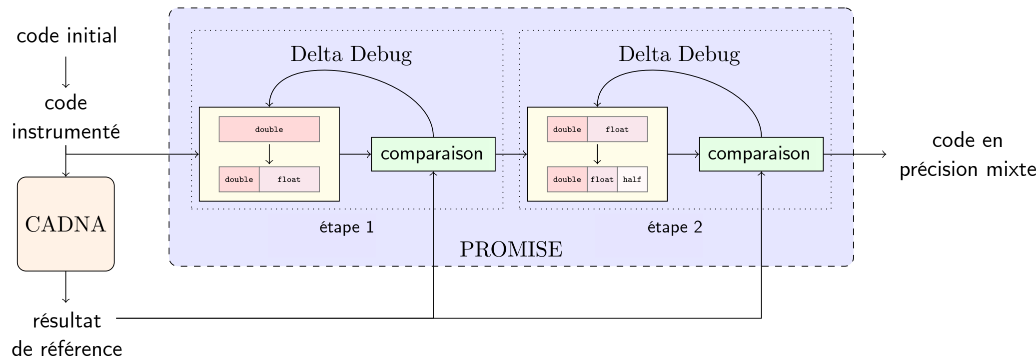

Figure 3.2. Fonctionnement de PROMISE pour les types half/simple/double

3.3 Autres outils d'auto-tuning de précision

Cette section présente à titre d'exemples d'autres outils d'auto-tuning de précision, avec des approches statiques et dynamiques.

3.3.1 Outils statiques

Salsa [?] est un outil permettant d'améliorer la précision des programmes en arithmétique flottante. Il utilise la représentation abstraite pour calculer les intervalles de valeurs de chaque variable ainsi que les bornes d'erreur. Étant donné ces informations, ainsi qu'un ensemble de règles de modifications possibles, Salsa modifie sémantiquement les expressions arithmétiques d'un programme, les remplaçant par des expressions mathématiquement équivalentes permettant de diminuer les erreurs d'arrondi.
Salsa a été adapté [?] pour réaliser de l'auto-tuning de précision en deux passes après une première analyse statique permettant d'évaluer les intervalles de valeurs de chaque variable. La première passe, "en avant", propage les erreurs d'arrondi sur chaque résultat intermédiaire. La seconde passe, "en arrière" cette fois, se base sur ces résultats pour établir la précision minimale nécessaire de chaque variable afin de respecter la précision demandée sur le résultat par l'utilisateur.

TAFFO [?] est un *framework* d'auto-ajustement de la précision implémenté comme un ensemble de *plug-ins* dans LLVM. Il permet principalement de transformer les opérations en arithmétique flottante en des opérations équivalentes basées sur une autre représentation des nombres en machine (virgule fixe notamment). Cela permet d'utiliser certains programmes de manière plus efficace sur des microcontrôleurs ne pouvant pas nécessairement effectuer d'opérations flottantes, mais aussi d'améliorer la performance de certains programmes sur

des machines plus standards voire haut de gamme. Pour utiliser TAFFO, l'utilisateur doit annoter le programme C/C++ qu'il souhaite ajuster pour transmettre des informations à TAFFO (comme donner des variables à passer en précision fixe). TAFFO consiste ensuite en 5 passes :

— **L'initialisation** qui traite les annotations données par l'utilisateur dans le programme et détermine quelles opérations seront impactées.

— **L'analyse des besoins** qui complète la représentation intermédiaire LLVM en effectuant notamment une passe d'analyse statique basée sur l'arithmétique par intervalles.

— **L'allocation des types** qui détermine quel type de données doit être utilisé pour chaque résultat intermédiaire. Cette passe essaie dans un second temps de minimiser la perte de précision due aux opérations de cast.

— **La transformation du code** qui modifie le code pour forcer l'utilisation des types déterminés dans la passe précédente.

— **La propagation des erreurs** qui établit une estimation de l'erreur sur le résultat par rapport au code d'origine.

3.3.2 Outils dynamiques

PRECIMONIOUS [**?**] est un outil d'auto-ajustement de la précision avec une approche dynamique. Il utilise l'algorithme du Delta-Debug pour tester les différentes configurations de la manière suivante : chaque variable est associée à un ensemble de types possibles. Puis, à chaque itération, l'algorithme détermine un ensemble minimal de variables devant garder la plus grande précision possible. Pour ces variables, l'ensemble de types associés est réduit à cette précision uniquement. Ces variables sont donc ignorées par la suite. Pour les autres variables, le type de plus grande précision est retiré de leurs ensembles de types associés. Ceci est répété sur les sous-ensembles de variables jusqu'à ce qu'à chaque variable ne soit associé plus qu'un seul type. L'ensemble minimal de variables devant rester en précision la plus élevée est établi en respectant à la fois une contrainte sur la précision du résultat et une contrainte de performance.
Les différentes variantes à tester, pour chaque nouvelle configuration de types, sont créées en compilant le programme originel vers la représentation intermédiaire LLVM, puis en appliquant les modifications de types directement dans cette représentation intermédiaire. Le résultat est un fichier binaire ayant pris en compte les modifications de types.
Pour la majorité des programmes analysés, PRECIMONIOUS réduit la précision de ces derniers et obtient des améliorations de performance pouvant atteindre 41%.

FloatSmith [**?**] combine trois différents outils. Le premier, CRAFT - Configurable Runtime Analysis for Floating-point Tuning [**?**] (`https://github.com/crafthpc/craft`), est un outil d'analyse de programmes en virgule flottante qui permet de trouver les variables dont la précision peut être réduite. Il permet donc de créer et tester les différentes configurations. Différentes stratégies ont été développées pour cela, dont le Delta-Debug. Le second, TypeForge (`https://github.com/LLNL/typeforge`), est un outil d'instrumentation qui permet de remplacer les types utilisés dans un code par un type souhaité. Il est aussi utilisé pour

trouver des dépendances de types entre les variables, et donc réduire l'espace de recherche. Enfin, ADAPT [?] (http://github.com/LLNL/adapt-fp) peut éventuellement être utilisé pour réduire encore l'espace de recherche en utilisant la différentiation automatique. FloatSmith permet ainsi à l'utilisateur d'obtenir une configuration respectant le critère de précision demandé et maximisant le gain en temps.

Enfin, les auteurs de [?] introduisent un outil d'auto-tuning de précision basé sur l'outil d'analyse fp2mp [?], le découpage de boucles [?] et le Delta-Debug.
L'outil **fp2mp** est basé sur LLVM et instrumente les programmes en virgule flottante avec des opérations en précision arbitraire (ou "précision multiple") en utilisant la bibliothèque MPFR (https://www.mpfr.org), et permettant ainsi d'évaluer l'impact de telles modifications de précision sur le résultat. Il permet aussi de mettre à jour la précision des opérations dans la représentation intermédiaire LLVM. Ainsi peut-il aussi être utilisé en tant qu'outil d'auto-tuning de précision.
Également introduite dans [?], la méthode de division de boucles (*loop splitting*) consiste à diviser une boucle en plusieurs sous-boucles. Ces dernières itèrent ainsi sur des espaces d'itération plus petits. L'idée est d'exécuter ces sous-boucles avec des précisions différentes. L'outil de modification de boucles permet aussi d'appliquer le déroulage de boucles, qui consiste à dupliquer les instructions au sein d'une boucle afin de réduire l'espace d'itération, diminuer les tests de fin de boucle et ainsi optimiser le rapport espace-temps.
L'outil d'auto-ajustement de la précision introduit dans [?] applique dans un premier temps les modifications de boucles données par l'utilisateur au programme, ce qui donne une représentation intermédiaire modifiée. L'outil fp2mp permet ensuite d'introduire les opérations en précision arbitraire équivalentes aux opérations flottantes dans la représentation intermédiaire, et de récupérer la liste des opérations pouvant changer de précision. Enfin, le Delta-Debug permet de trouver un sous-ensemble maximal de transformations permettant d'obtenir un résultat acceptable.
Un travail conjoint a été réalisé avec Youssef Fakhreddine, co-auteur de [?], lors d'un échange supporté par le GDR-IM (Groupement de Recherche Informatique Mathématique) - CNRS. Ce travail a permis de comparer l'application de fp2mp, modifiant les types des opérations, à celle de PROMISE, modifiant les types des variables. Étant donné cette différence cruciale, il est difficile de comparer les résultats obtenus. Néanmoins, les solutions obtenues avec chaque outil sont cohérentes. La division et le déroulage de boucles ont aussi été effectués à la main afin d'appliquer PROMISE sur le code correspondant, permettant de comparer les résultats avec ceux obtenus en utilisant la division de boucles dans LLVM et fp2mp.

3.4 Conclusion

Dans ce chapitre, nous avons présenté l'auto-tuning de précision, qui permet de diminuer la précision des variables dans un code sans compromettre la précision requise sur le résultat final. L'utilisation de précisions plus faibles présente un bénéfice en termes de performance. Pour réaliser cela, nous avons introduit l'outil PROMISE, qui calcule un résultat de référence dont la précision est estimée grâce à CADNA, puis utilise l'algorithme du Delta-Debug pour

tester les différentes configurations possibles. Prenant en entrée un programme et une contrainte sur le nombre de chiffres significatifs exacts souhaité pour le résultat, PROMISE permet d'obtenir une version en précision mixte du programme qui respecte la contrainte sur le résultat.

Deuxième partie

Contributions

4 Auto-tuning de précision de réseaux de neurones avec PROMISE

Comme vu au chapitre précédent, nous pouvons utiliser les outils d'auto-tuning de précision pour réduire la précision des variables d'un programme et ainsi réduire l'espace utilisé par celui-ci. Dans ce chapitre, nous utilisons PROMISE afin de réduire la précision des variables présentes dans un réseau de neurones.

Les réseaux de neurones deviennent de plus en plus gros et requièrent de plus en plus d'énergie, alors même que l'énergie disponible peut être limitée, notamment dans le cadre de systèmes embarqués. Réduire l'espace utilisé par les réseaux de neurones représente donc un objectif important : on parle de compression de réseaux de neurones. Pour ce faire, les méthodes les plus connues sont l'élagage (ou *pruning*), qui consiste à supprimer certains poids du réseau de neurones (voir par exemple [**?**]), la décomposition tensorielle, qui consiste à décomposer les tenseurs afin de les stocker de manière plus compacte (voir [**?**]), et enfin la quantification des réseaux de neurones qui consiste à réduire la précision des paramètres du réseau, généralement en remplaçant les opérations en virgule flottante sur 32 bits par des opérations en virgule fixe sur b bits (souvent $b \in \{1, 2, 4, 8, 16\}$) (voir [**?**]).
Dans notre cas, nous utilisons l'auto-tuning de précision sur les paramètres du réseau de neurones. L'auto-tuning de précision de la phase d'inférence des réseaux de neurones utilisant des nombres à virgule fixe a été étudié dans [**?**]. Grâce à un système linéaire de contraintes, la précision minimale pour chaque neurone est trouvée en prenant en compte un seuil sur la précision du résultat final. Dans le cadre de réseaux de neurones utilisant des nombres flottants, l'article [**?**] s'intéresse à l'auto-tuning de précision de la phase d'inférence de réseaux d'interpolation, c'est-à-dire approchant des fonctions mathématiques. Les auteurs proposent un algorithme prenant en compte une tolérance δ sur l'écart relatif entre le résultat final et le résultat de référence obtenu en précision double. Pour respecter cette tolérance, et trouver la précision minimale possible pour chaque neurone, l'algorithme résout un problème de programmation linéaire. Cela constitue une première différence avec notre étude où l'application de PROMISE est basée sur une méthode "essai et erreur". La seconde différence, primordiale, est que le résultat de référence calculé dans PROMISE est obtenu et validé par application de l'Arithmétique Stochastique Discrète.
En effet, l'application de PROMISE aux réseaux de neurones nous permet non seulement de les compresser en réduisant directement la précision de leurs paramètres, mais aussi de valider numériquement le résultat grâce à l'application de CADNA au sein de l'outil PROMISE. Bien que la sécurité et la robustesse des codes de réseaux de neurones aient été largement étudiées [**?**, **?**, **?**, **?**, **?**] notamment via des outils vérificateurs tels que MIPVerify [**?**], peu d'études ont finalement porté sur l'impact possible de l'accumulation des erreurs d'arrondi sur les réseaux de neurones. Malgré cela, il a été montré que la robustesse des réseaux de

neurones et des vérificateurs peut être remise en question notamment à cause de l'utilisation des nombres flottants [?]. Certains outils basés sur l'arithmétique par intervalle proposent donc un contrôle plus précis de la robustesse des réseaux de neurones [?, ?]. Le framework introduit dans [?] permet quant à lui d'analyser directement les erreurs d'arrondi au sein de la phase d'inférence des réseaux de neurones et d'obtenir une borne sur l'erreur absolue et l'erreur relative grâce à l'arithmétique des intervalles.

Nous commençons par décrire certaines notions liées aux réseaux de neurones en Section 4.1, puis nous verrons l'application de PROMISE sur les phases d'inférence en Section 4.2 et d'apprentissage en Section 4.3.

4.1 Réseaux de neurones

Un réseau de neurones est un système de calcul défini par différents neurones répartis sur différentes couches. Les couches les plus classiques sont les *couches denses* qui prennent en entrée un vecteur et consistent principalement en un produit matrice-vecteur. Plus précisément, pour un vecteur d'entrée $x^{(k)} \in \mathbb{R}^n$, $k \in \mathbb{N}$, la couche dense calcule la sortie $x^{(k+1)} \in \mathbb{R}^m$ selon l'équation suivante :

$$x^{(k+1)} = g^{(k)}(W^{(k)}x^{(k)} + b^{(k)}) \tag{4.1}$$

où $W^{(k)} \in \mathbb{R}^{m\times n}$ est la matrice de poids, $b^{(k)} \in \mathbb{R}^m$ est le vecteur de biais et $g^{(k)}$ est la fonction d'activation. Un neurone d'une couche est alors caractérisé par son vecteur de poids, qui constitue une ligne de la matrice de poids, ainsi qu'un scalaire, le biais. Ces deux derniers sont appelés les *paramètres* d'un neurone, ou plus globalement de la couche ou du réseau de neurones correspondant.
La fonction d'activation $g^{(k)}$ est une fonction non-linéaire et monotone. Parmi les plus utilisées, nous retrouvons :

— Sigmoïde : $\sigma(x) = 1/(1 + e^{-x})$ pour tout $x \in \mathbb{R}$;

— Tangente hyperbolique : $\tanh(x)$ pour tout $x \in \mathbb{R}$;

— Unité linéaire rectifiée : $\text{ReLU}(x) = \max(x, 0)$ pour tout $x \in \mathbb{R}$, (ReLU de l'anglais *Rectified Linear Unit*) ;

— Softmax : normalise un vecteur $x \in \mathbb{R}^n$ en une distribution de probabilité sur les classes de sorties. Pour chaque élément x_i de x, $\text{softmax}(x_i) = e^{x_i} / \sum e^{x_j}$.

La Figure 4.1 montre une représentation graphique d'un réseau de neurones à deux couches.

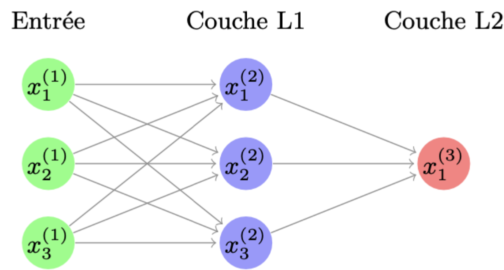

Figure 4.1. Réseau de neurones à deux couches

Les couches denses sont parfois généralisées aux tableaux multidimensionnels en utilisant le produit tensoriel.

Parmi les autres types de couches principalement utilisées, nous retrouvons les *couches convolutives*, appliquant un produit de convolution. Celui-ci s'applique en général sur un tableau de pixels de 3 dimensions représentant une image (niveaux de couleur RVB). Un filtre, généralement une matrice 3x3, parcourt le tableau et s'applique pas à pas aux sous-parties de l'image, calculant le produit scalaire entre la partie de l'image concernée et le filtre. Ce produit est ensuite récupéré dans une matrice de sortie. Puis le filtre se décale, jusqu'à parcourir toute l'image. Le pas peut être choisi par l'utilisateur, mais il est en général égal à 1. La matrice résultante est souvent appelée carte d'activation ou *feature map*. Un exemple 2D est donné en Figure 4.2.

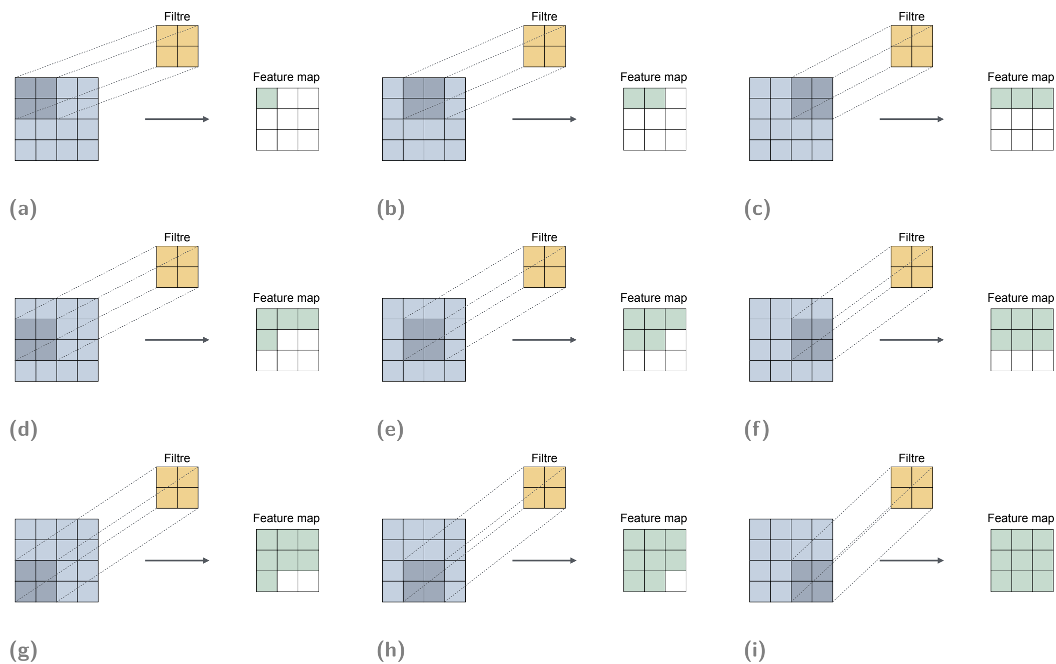

Figure 4.2. Produit de convolution 2D

Enfin, il existe aussi des couches ne possédant pas de poids ni de biais et permettant simplement d'échantillonner ou de modifier la forme des données. Parmi celles-ci, nous avons la couche de *max pooling* qui réduit la taille de l'entrée en appliquant un filtre prenant la valeur maximale par zone. Elle va en général de pair avec une couche de convolution, à la suite de celle-ci. Une illustration en est donnée en Figure 4.3. La couche de *flatten* permet quant à elle de passer d'un tenseur en entrée à un vecteur, en dépliant simplement le tenseur d'entrée.

Figure 4.3. Max pooling avec un filtre 2×2

4.2 PROMISE et inférence

4.2.1 Méthodologie

Pour la mise en œuvre des réseaux de neurones, nous utilisons le langage Python avec Keras [1] ou PyTorch [2]. Keras et Pytorch sont deux bibliothèques Python open source qui définissent des structures de données et des fonctions permettant de créer et d'entraîner des modèles de réseaux de neurones. Elles nous permettent également d'enregistrer les modèles construits au format HDF5 (*Hierarchical Data Format*) [3], un format de fichier conçu pour stocker et organiser des données volumineuses. Le format HDF5 n'utilise que deux types d'objets : les *datasets*, qui sont des tableaux multidimensionnels de type homogène, et les groupes, qui contiennent des datasets, ou d'autres groupes. Les fichiers HDF5 peuvent être lus par des programmes Python à l'aide du paquet h5py. Les données associées peuvent être manipulées à l'aide de Pandas [4], une bibliothèque Python qui propose des structures de données et des opérations pour gérer de grandes quantités de données.

1. https://keras.io
2. https://pytorch.org
3. https://www.hdfgroup.org
4. https://pandas.pydata.org

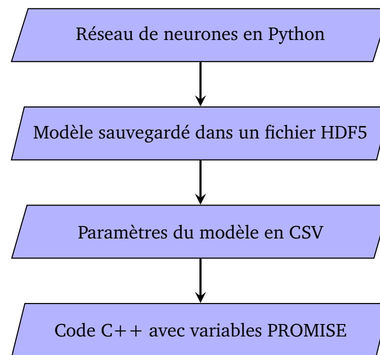

Figure 4.4. Logigramme de la traduction d'un réseau de neurones Python en programme C++ pour PROMISE

Keras est utilisé pour développer, entraîner et sauvegarder nos réseaux de neurones, sauf dans le cas du pendule inversé qui utilise PyTorch. Les étapes du processus de traduction depuis le modèle Python jusqu'au code C++ sont résumées dans la Figure 4.4. Pour chaque réseau de neurones, nous convertissons d'abord le fichier HDF5 du modèle en plusieurs fichiers CSV à l'aide d'un script Python. Le script charge le fichier HDF5, stocke les paramètres dans des DataFrames Pandas, puis enregistre les paramètres dans des fichiers CSV en utilisant la fonction Pandas `DataFrame.to_csv`. Pour chaque couche qui en a besoin, nous créons un fichier CSV avec les poids de la couche et un fichier CSV avec le biais de la couche. En effet, certaines couches n'ont pas besoin de poids ni de biais, par exemple les couches de *flatten* qui ne font que changer la forme des données (de 2 dimensions à 1 dimension par exemple). Ensuite, nous utilisons les données des fichiers CSV pour créer un programme C++, une fois de plus en utilisant un script Python qui lit les fichiers CSV, crée les variables nécessaires ainsi que les calculs correspondant au modèle. Les scripts de traduction sont basés sur le travail effectué dans la bibliothèque keras2c[5]. Une fois la version C++ créée, nous appliquons PROMISE sur celle-ci.

4.2.2 Résultats expérimentaux

Les résultats obtenus pour quatre réseaux de neurones différents sont présentés dans cette section. Pour les réseaux de neurones utilisant une base de données, test_data[i] désigne la $(i+1)^{\text{ème}}$ entrée du jeu de test fourni par la base de données. PROMISE est appliqué à chaque réseau de neurones en considérant un type par neurone (half, simple ou double), puis un type par couche, c'est-à-dire que tous les paramètres d'une couche ont la même précision. Dans notre analyse, la différence entre les deux approches réside uniquement dans le nombre de déclarations de variables avec un type PROMISE différent dans le code. Cependant, dans les couches denses, le fait d'avoir un type par neurone implique des produits scalaires indépendants, alors qu'un type par couche permettrait de calculer des produits matrice-vecteur et donc d'être plus performant.

Comme nous considérons la phase d'inférence, nous devons fixer l'entrée du réseau de neurones. Les résultats sont obtenus avec des données d'entrée en double précision car

5. https://f0uriest.github.io/keras2c/

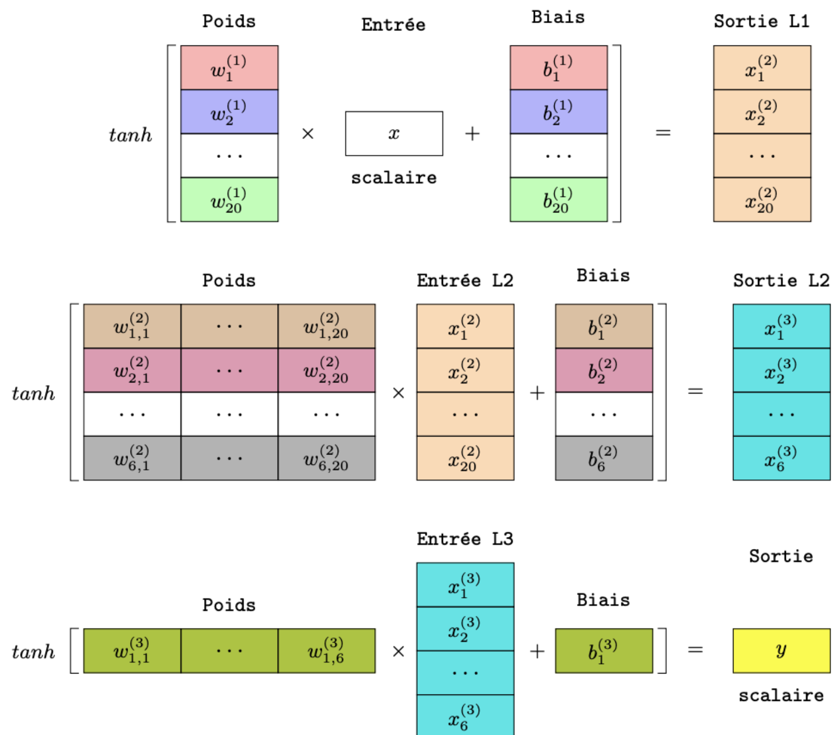

Figure 4.5. Représentation matricielle des calculs effectués dans le réseau de neurones sinus

le type des données d'entrée n'a pas d'impact sur les configurations obtenues. Il est donc ignoré pour l'instant. Celui-ci a cependant un impact sur le temps d'exécution d'un réseau de neurones avec une configuration choisie, car, intervenant dans les calculs de la première couche, il force la conversion (*cast*) des paramètres de celle-ci.
Conformément à la Figure 3.2, pour tout réseau de neurones, la valeur de référence est la valeur calculée à la première étape de PROMISE à l'aide de CADNA. Sauf indication contraire, tous les résultats présentés dans cette section ont été obtenus sur un processeur Intel Core i5-8400 de 2,80 GHz avec 6 cœurs et 16 Go de RAM.

4.2.2.a. Approximation de sinus

Pour approcher la fonction sinus, nous utilisons un réseau de neurones classique avec 3 couches denses. Il s'agit d'un problème hypothétique, puisqu'il n'est pas nécessaire d'utiliser un réseau de neurones pour calculer le sinus. Cependant, cet exemple simple permet de valider notre approche. La fonction d'activation $tanh$ est utilisée dans les 3 différentes couches denses. Les couches ont respectivement 20, 6 et 1 neurone(s) et l'entrée est une valeur scalaire x. La Figure 4.5 présente le calcul effectué par le réseau de neurones. Les variables colorées sont des variables PROMISE, dont la précision peut être modifiée. En considérant ici un type par neurone, les variables de la même couleur se voient attribuer la même précision. Le type de sortie de chaque couche est également pris comme variable PROMISE avec un type à ajuster. La même logique est appliquée pour les autres réseaux de

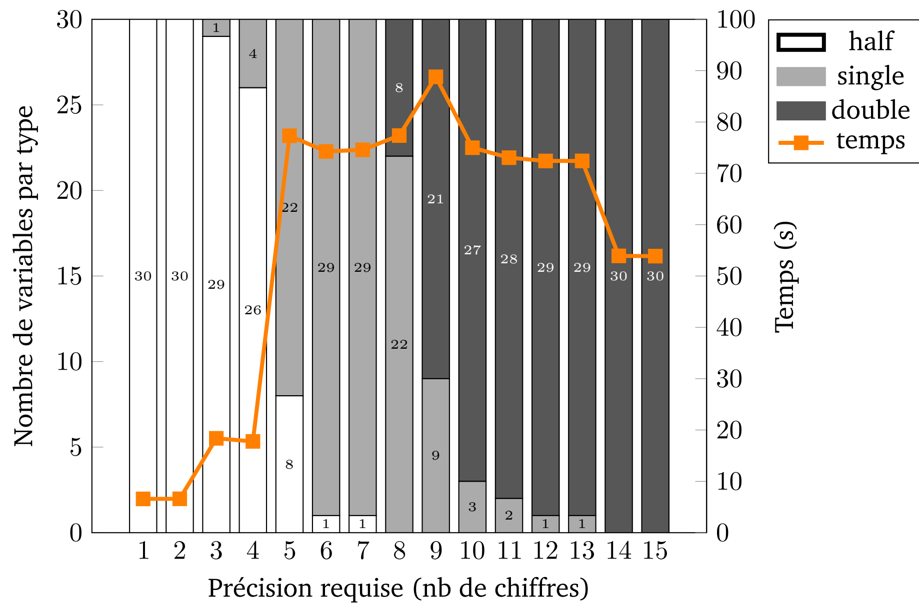

Figure 4.6. Nombre de variables de chaque type et temps de calcul pour le réseau sinus avec comme valeur d'entrée 0.5

neurones lorsque l'on considère un type par neurone. Dans cet exemple, nous attribuons un type à 27 neurones et 3 sorties, soit 30 types PROMISE.

La Figure 4.6 montre la distribution des différents types avec une valeur d'entrée de 0.5 en considérant un type par neurone. L'axe des abscisses représente la précision requise sur les résultats, c'est-à-dire le nombre de chiffres significatifs en commun avec le résultat de référence calculé à l'aide de CADNA. On peut noter l'évolution de la distribution en fonction du nombre de chiffres significatifs exacts requis sur le résultat. Comme attendu, nous n'avons d'abord que des variables en précision half, puis certaines d'entre elles passent en simple, puis en double, jusqu'à ce que toutes soient en précision double. Par conséquent, le fait d'exiger la plus grande précision n'est pas compatible avec la réduction de la précision dans ce réseau de neurones. Cependant, un bon compromis peut être trouvé, puisque nous n'avons que des variables en précision simple et half pour une précision requise allant jusqu'à 7 chiffres, et que nous avons encore 1/3 de variables en précision simple pour une précision requise jusqu'à 9 chiffres.

La Figure 4.6 présente également le temps de calcul de PROMISE en secondes pour chaque précision requise. Il correspond au temps de calcul du résultat de référence, plus le temps nécessaire pour appliquer deux fois l'algorithme Delta-Debug (de la précision double à simple puis de la précision simple à half), en compilant et en exécutant la distribution testée à chaque fois. On peut remarquer que le temps de calcul reste raisonnable, compte tenu des 3^{30} possibilités (moins de 2 minutes).

La Figure 4.7 montre le résultat en considérant une précision par couche avec la même entrée 0.5. Cette approche par couche permet de diminuer le temps d'exécution de PROMISE, mais n'aide pas vraiment à diminuer la précision des paramètres du réseau de neurones. Au contraire, chaque fois qu'un paramètre d'une couche nécessite une précision supérieure,

Figure 4.7. Précision de chaque couche pour le réseau sinus avec comme valeur d'entrée 0.5

tous les paramètres de la même couche passent en précision supérieure. Néanmoins, nous pouvons remarquer que la première couche (qui représente 2/3 des neurones) reste en précision half pour une précision requise jusqu'à 4 chiffres.

Les Figures 4.8 et 4.9 montrent que la valeur d'entrée peut avoir un léger impact sur la distribution des types. En effet, celles-ci représentent les résultats obtenus avec une valeur d'entrée différente égale à 2.37, au lieu de 0.5 précédemment. D'après la Figure 4.6, avec la valeur d'entrée 0.5, PROMISE fournit une distribution de types avec 8 neurones en précision half et 22 neurones en précision simple pour une précision requise de 5 chiffres. D'après 4.8, avec une valeur d'entrée 2.37, nous obtenons seulement 4 neurones en précision half pour une précision requise de 5 chiffres. En fait, pour 3, 4 ou 5 chiffres de précision requis sur le résultat, moins de neurones sont en précision half qu'avec l'entrée 0.5. Mais si 8 chiffres sont requis, un neurone reste en précision half avec une entrée de 2.37, alors qu'aucun neurone n'est en précision half avec l'entrée 0.5. Malgré tout, la distribution des types en fonction de la précision requise reste globalement la même.

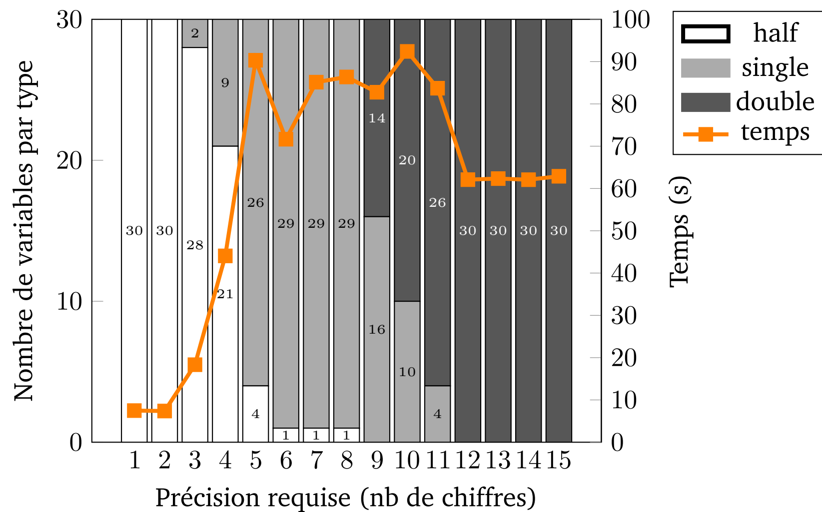

Figure 4.8. Nombre de variables de chaque type et temps de calcul pour le réseau sinus avec comme valeur d'entrée 2.37

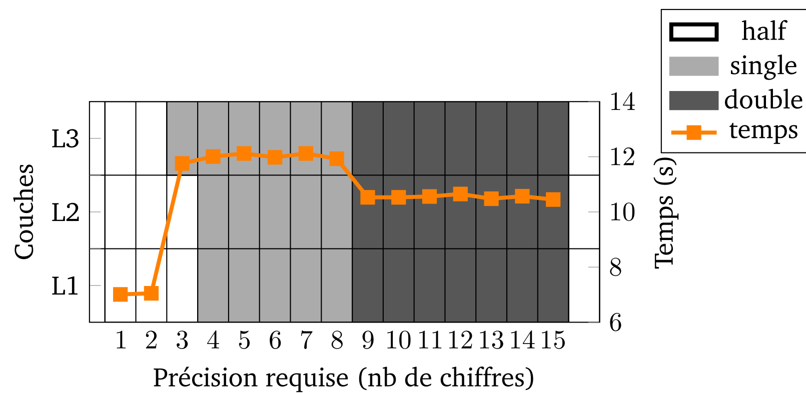

Figure 4.9. Précision de chaque couche pour le réseau sinus avec comme valeur d'entrée 2.37

PROMISE nous permet ainsi de diminuer la précision des paramètres de notre réseau de neurones, mais les résultats peuvent varier selon l'entrée fixée lors de la phase d'inférence. Les Figures 4.10 et 4.11 représentent les résultats les plus pessimistes obtenus en considérant 1000 valeurs d'entrée différentes entre 0 et 2π. Dans ce cas, nous pouvons voir que la précision maximale atteinte est de 12 chiffres seulement contre 15 précédemment, nous avons 3 variables PROMISE qui passent en précision simple dès la première couche et toutes les variables prennent la précision double à partir de 8 chiffres de précision requis, là où pour la valeur d'entrée 2.37, il fallait atteindre 12 chiffres requis, et même 14 chiffres pour la valeur 0.5. Néanmoins, même en considérant le résultat le plus pessimiste sur ces 1000 valeurs, nous avons une majorité de variables en précision réduite jusque 6 chiffres de précision requis sur le résultat. En analysant les configurations obtenues pour les 1000 résultats simultanément, pour 1 chiffre de précision demandé sur le résultat, 999 configurations ont une majorité de variables en précision half, et jusque 5 chiffres de précision demandé plus de la moitié des configurations sont aussi dans ce cas. Enfin, jusqu'à

8 chiffres de précision demandés, la très grande majorité (936/1000) des configurations obtenues ont une majorité de variables en précision simple, et 1 configuration a toujours une majorité de variables en précision half. Pour 9 chiffres, le nombre de configurations avec une majorité de variables en précision simple passe juste en dessous de la moitié avec 449 configurations. Le temps mis par PROMISE pour obtenir les résultats est ici plus long que précédemment étant donné les 1000 valeurs données en entrée. Néanmoins, en croisant les Figures 4.6, 4.8 et 4.10, nous pouvons voir que les temps les plus longs pour calculer une configurations sont obtenus lorsque celle-ci demande un mélange de précision. Par exemple, en Figure 4.10, le temps mis pour 4 chiffres de précision est moins long que pour 3 car pour 4 toutes les variables sont en précision simple, alors que pour 3, nous avons un mélange de précisions simple et half. Cela s'explique par le Delta-Debug qui lorsqu'il trouve un sous-ensemble de variables passant en précision half, essaie de compléter celui-ci, alors que si aucune variable ne passe en précision half, il termine.

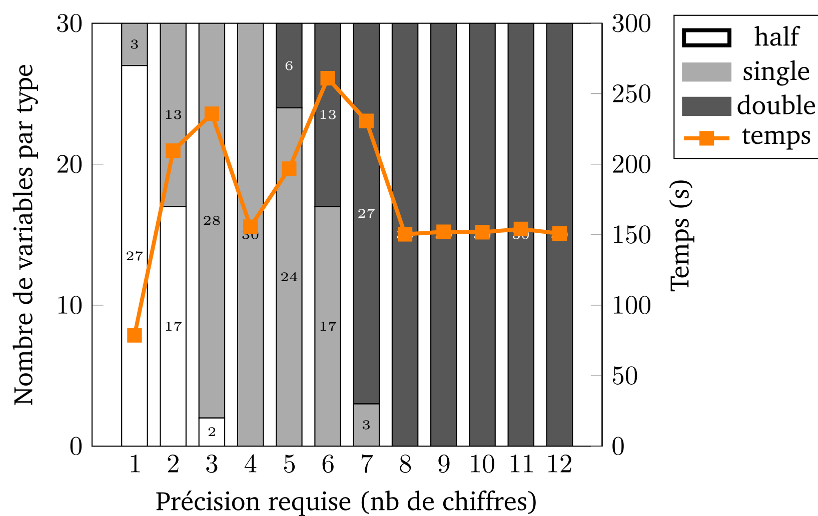

Figure 4.10. Nombre de variables de chaque type et temps de calcul pour le réseau sinus avec 1000 valeurs d'entrée

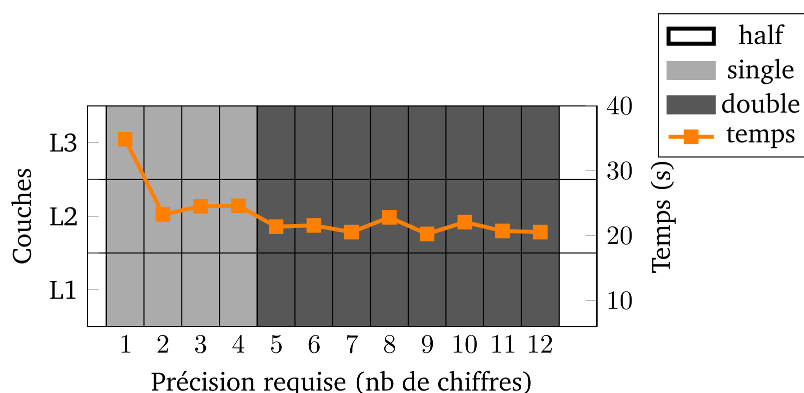

Figure 4.11. Précision de chaque couche pour le réseau sinus avec 1000 valeurs d'entrée

Figure 4.12. Images de la base de données MNIST (https://en.wikipedia.org/wiki/MNIST_database)

4.2.2.b. Classification MNIST

Le réseau de neurones MNIST est un réseau de classification d'images de chiffres écrits à la main provenant de la base de données MNIST [6] (voir Figure 4.12).

Ce réseau utilise également des couches denses classiques. Contrairement au réseau approchant le sinus, l'entrée est un vecteur de taille 784 correspondant à une image en niveau de gris, soit une matrice mise à plat. Cette image passe par deux couches denses : la première avec 64 neurones et la fonction d'activation ReLU, la seconde avec 10 neurones et la fonction d'activation softmax qui fournit la distribution de probabilité pour les 10 différentes classes possibles. Si l'on considère comme précédemment un type par neurone, plus un type pour la sortie de chaque couche, 76 types différents doivent être définis en précision half, simple ou double.
La Figure 4.13 montre la distribution des types en considérant un type par neurone avec l'image d'entrée test_data[61] (62ème donnée de test sur les 10000 fournies par MNIST). L'axe x représente la précision requise sur la sortie consistant en un vecteur de taille 10. La précision maximale atteinte sur la sortie est de 13 chiffres, il n'a pas été possible d'atteindre une précision plus élevée. Une précision aussi élevée n'est toutefois pas nécessaire pour un réseau de classification. Cependant, nous avons tout de même effectué les tests de manière exhaustive : toutes les précisions possibles en précision double ont été successivement testées.
La principale différence avec le réseau de neurones portant sur le sinus réside dans le fait qu'un nombre important de neurones restent en précision half, quelle que soit la précision requise sur le résultat. En fonction de l'entrée, environ 50% des neurones peuvent rester en précision half, et parfois même près de 60%, comme le montre la Figure 4.15. Ainsi, l'application de PROMISE à ce réseau de neurones, même lorsqu'on exige la plus grande précision, permet d'abaisser la précision de ses paramètres. Les neurones restant dans la précision la plus faible varient légèrement en fonction de l'entrée, mais appartiennent toujours à la première couche. Par conséquent, la première couche semble avoir moins d'impact sur la précision du résultat final que la seconde. Les temps de calcul également indiqués dans la Figure 4.13 sont beaucoup plus élevés que pour le réseau d'approximation du sinus. Pour la majorité des précisions demandées, plus de 15 minutes sont nécessaires pour obtenir une version en précision mixte du réseau de neurones. Cela s'explique car

6. http://yann.lecun.com/exdb/mnist

le réseau de neurones portant sur MNIST comporte certes une couche de moins que le réseau d'approximation du sinus, mais plus de neurones. Puisque nous considérons un type par neurone, le nombre de configurations de type possibles (3^{76}) est beaucoup plus élevé, d'où la différence de temps de calcul. À cela s'ajoute aussi le temps d'exécution du réseau de neurones en lui-même, le réseau portant sur MNIST comportant plus de calculs que le réseau approchant le sinus. Cependant, l'exécution de PROMISE permet d'obtenir un code en précision mixte en un temps raisonnable, alors qu'un tel ajustement à la main aurait été beaucoup plus long.

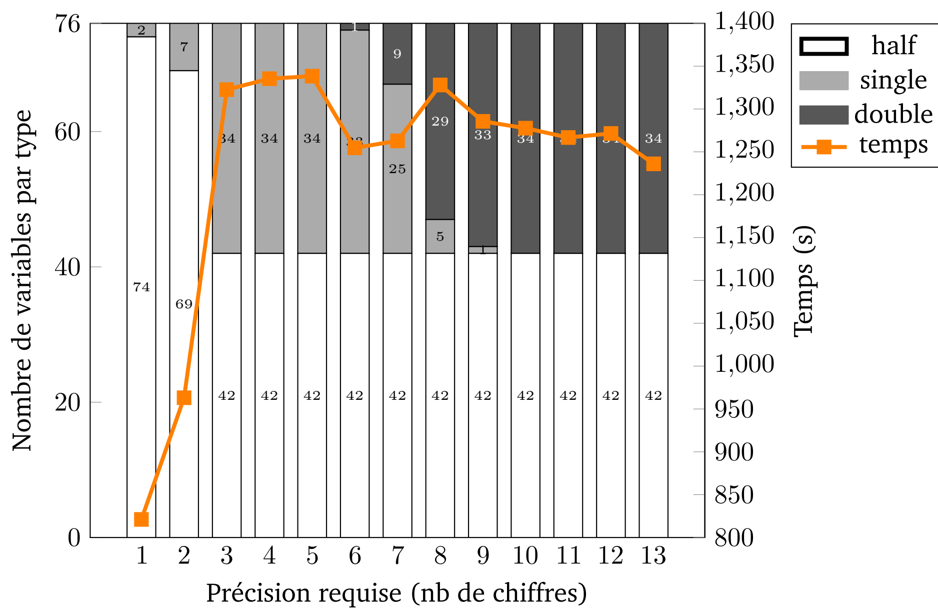

Figure 4.13. Nombre de variables de chaque type et temps de calcul pour le réseau MNIST avec comme valeur d'entrée test_data[61]

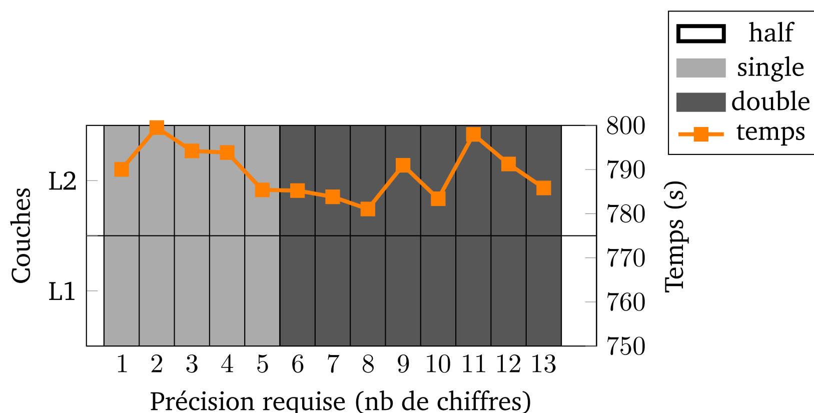

Figure 4.14. Précision de chaque couche pour le réseau MNIST avec comme valeur d'entrée test_data[61]

Que ce soit pour le réseau de neurones approchant le sinus ou le réseau de neurones MNIST, le temps d'exécution de PROMISE tend à augmenter avec la précision demandée.

Cela peut s'expliquer par l'algorithme du Delta-Debug utilisé dans PROMISE. Comme décrit précédemment, PROMISE vérifie d'abord si le critère de précision peut être satisfait en précision double. Ensuite, grâce à l'algorithme Delta-Debug, PROMISE tente d'abaisser la précision des variables de double à simple, ce qui peut être très rapide si la simple précision est suffisante pour satisfaire la précision requise. Enfin, PROMISE tente de transformer les déclarations en précision simple en déclarations en précision half, ce qui à nouveau peut être rapide si la précision half convient. Le nombre de programmes compilés et exécutés par PROMISE tend à augmenter avec la précision requise sur le résultat. Par exemple, dans le cas du réseau de neurones MNIST, si 1 ou 2 chiffres sont requis, 18 configurations de types sont testées par PROMISE, alors que si 7 chiffres sont requis, 260 configurations sont testées.

Les résultats obtenus en considérant une précision par couche sont présentés en Figure 4.14. L'analyse est similaire à celle donnée précédemment pour le réseau de neurones approchant la fonction sinus. Avec l'approche par couche, le temps d'exécution de PROMISE est plus faible qu'avec l'approche par neurone, mais cette approche oblige à déclarer certaines variables avec une plus grande précision. On peut remarquer qu'avec l'approche par couche, les deux couches ont la même précision. Tous les paramètres du réseau sont donc du même type.

Les Figures 4.15 et 4.16 présentent les résultats obtenus avec une autre entrée. Comme pour le réseau de neurones approchant le sinus, le changement d'entrée induit de légers changements dans les configurations de types fournies par PROMISE. Cependant, la même tendance peut être observée.

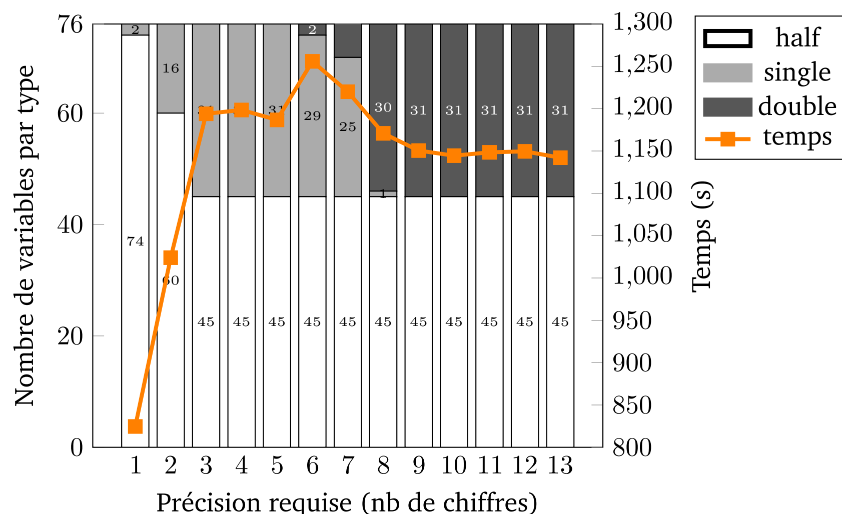

Figure 4.15. Nombre de variables de chaque type et temps de calcul pour le réseau MNIST avec comme valeur d'entrée test_data[91]

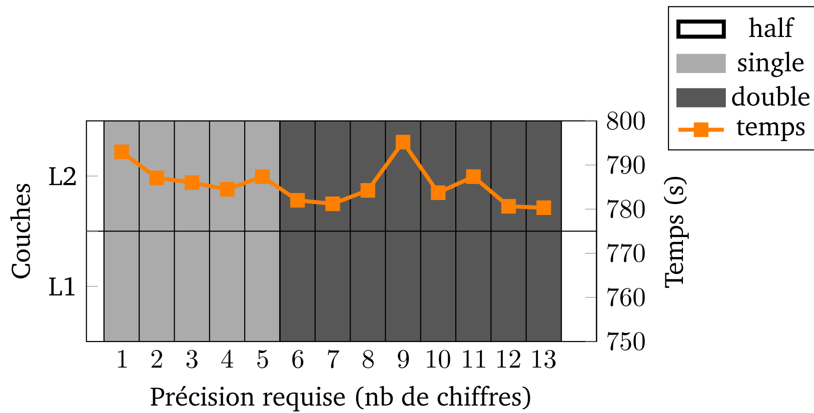

Figure 4.16. Précision de chaque couche pour le réseau MNIST avec comme valeur d'entrée test_data[91]

4.2.2.c. Classification CIFAR

Le réseau de neurones étudié ici est également un réseau de classification d'images, appliquée cette fois à la base de données CIFAR10. CIFAR [7] est une base de données comprenant 100 classes d'images en couleur, réduite à 10 classes dans CIFAR10 (voir Figure 4.17). Les images étant de taille $32 \times 32 \times 3$ (32 pixels de largeur, 32 pixels de hauteur, 3 canaux de couleur), l'entrée du réseau est un tenseur en 3 dimensions de forme (32, 32, 3). Le réseau se compose de 5 couches : une couche convolutive de 32 neurones avec fonction d'activation ReLU, suivie d'un max pooling de taille 2x2, une couche convolutive de 64 neurones avec fonction d'activation ReLU, une couche flatten, et enfin une couche dense de 10 neurones avec fonction d'activation softmax. En considérant un type par neurone, 111 types peuvent être définis au total. Les résultats présentés ici ont été obtenus sur un CPU 2.80 GHz Intel Core i9-10900 avec 20 coeurs et 64GB RAM.
La précision maximale possible des résultats est de 13 chiffres. Comme évoqué précédemment, pour un réseau de classification, une précision aussi élevée n'est pas nécessaire, mais des tests exhaustifs ont été effectués. Les résultats présentés ici correspondent à deux images d'entrée sur les 10 000 fournies par CIFAR10. Les Figures 4.18 et 4.20 présentent les configurations de types données par PROMISE avec respectivement test_data[386] et test_data[731] en considérant un type par neurone. Les Figures 4.19 et 4.21 montrent les résultats obtenus avec les mêmes images d'entrée en considérant un type par couche.

7. https://www.cs.toronto.edu/~kriz/cifar.html

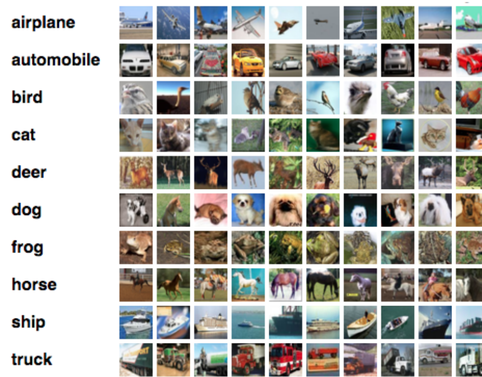

Figure 4.17. Images de la base de données CIFAR10 (https://www.cs.toronto.edu/~kriz/cifar.html)

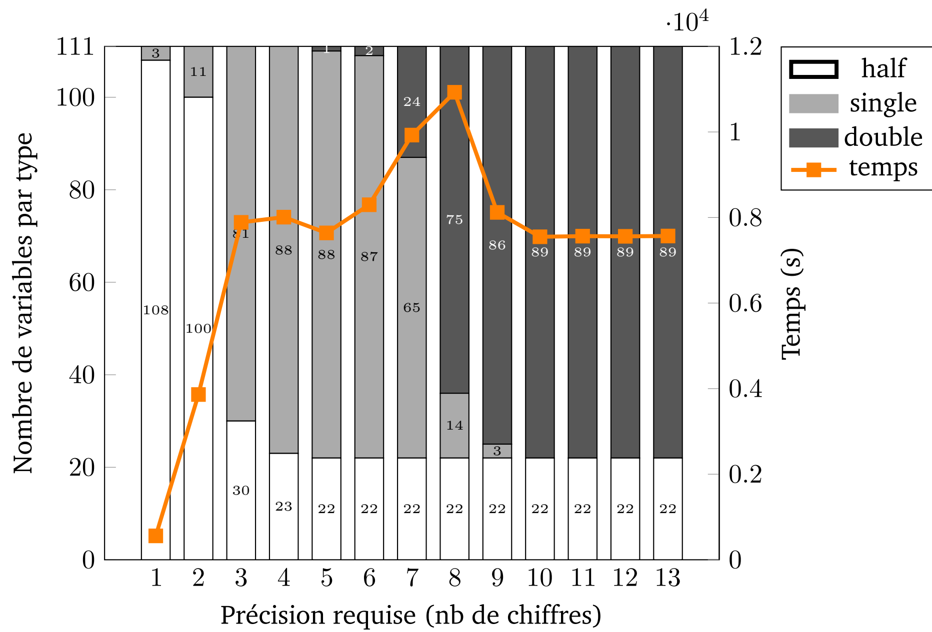

Figure 4.18. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau CIFAR avec comme valeur d'entrée test_data[386]

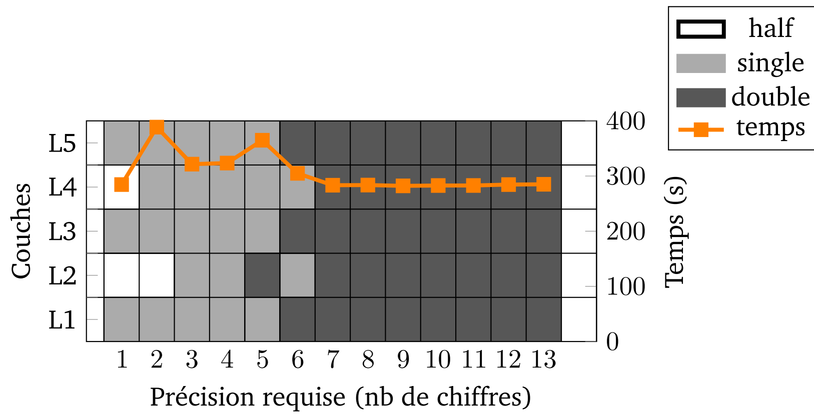

Figure 4.19. Précision de chaque couche pour le réseau CIFAR avec comme valeur d'entrée test_data[386]

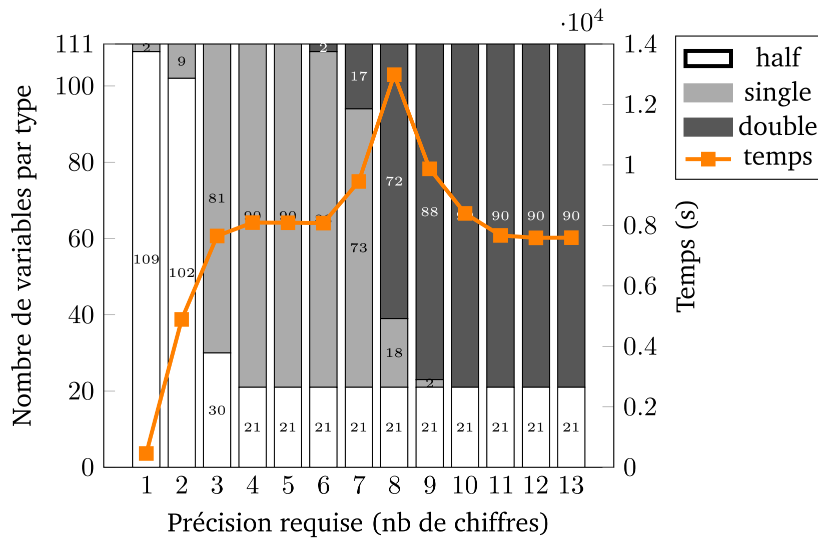

Figure 4.20. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau CIFAR avec comme valeur d'entrée test_data[731]

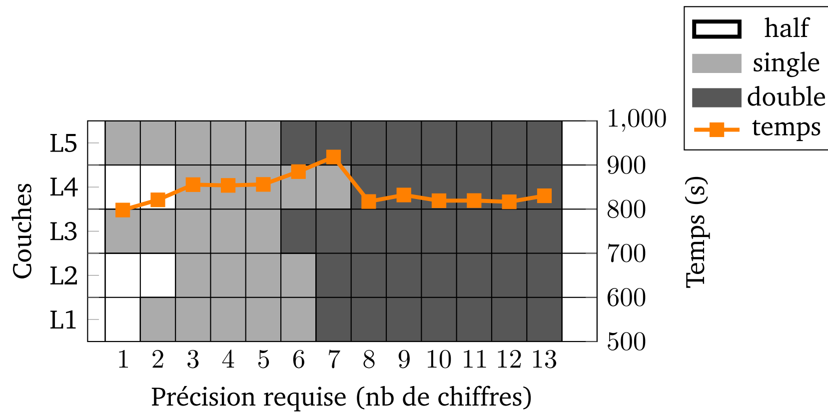

Figure 4.21. Précision de chaque couche pour le réseau CIFAR avec comme valeur d'entrée test_data[731]

Le temps de calcul de PROMISE tend à augmenter avec la précision requise. Comme cela a déjà été mentionné en 4.2.2.b, cela peut s'expliquer par l'algorithme de Delta-Debug. Comme nous l'avons déjà observé, le fait de considérer un type par couche permet d'obtenir des temps de calcul plus faibles pour PROMISE, mais donne souvent des programmes de précision uniforme contrairement à l'approche par neurones. Néanmoins, avec un nombre de couches plus élevé, ici 5, cela se vérifie moins et l'application par couche est donc plus intéressante. Encore une fois, l'image d'entrée a un léger impact sur les configurations de types fournies par PROMISE mais la même tendance peut être observée, avec un certain nombre de variables restant en précision half. D'autres expériences ont aussi été menées sur CIFAR10, mais avec des réseaux de neurones comportant plus de couches (jusqu'à 8 couches). Là encore, PROMISE fournit des configurations de types appropriées en tenant compte de la précision demandée. Cependant, l'exécution de PROMISE (qui comprend la compilation et l'exécution des différentes configurations) rend les tests exhaustifs plus difficiles dans cette situation. Les améliorations possibles de PROMISE décrites dans la Section 5.3 permettraient un ajustement de la précision dans des réseaux de neurones plus importants qui prennent eux-mêmes beaucoup de temps.

4.2.2.d. Pendule Inversé

Nous présentons ici les résultats obtenus avec un réseau de neurones présenté dans [?] dans le contexte de l'apprentissage par renforcement pour le contrôle autonome. Dans [?], des méthodes sont proposées pour l'approximation certifiée de systèmes de contrôle utilisant des réseaux de neurones. Le réseau de neurones utilisé ici, lié à un pendule inversé (Figure 4.22), fournit une approximation d'une fonction de Lyapunov. Il se compose de 2 couches denses et utilise la fonction d'activation $tanh$. L'entrée est un vecteur d'état $x \in \mathbb{R}^2$ et la sortie, un scalaire dans $\mathbb{R}$, est une valeur approchée de la fonction de Lyapunov pour l'entrée donnée. La première couche comporte 6 neurones et la deuxième couche un seul. Étant donné la proximité entre les deux modèles, les résultats attendus sont proches de ceux obtenus pour l'approximation du sinus. La Figure 4.23 présente à la fois la distribution des différentes précisions et le temps d'exécution de PROMISE en fonction de la précision

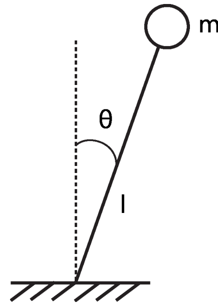

Figure 4.22. Pendule inversé (image de [?])

demandée pour l'entrée (0.5, 0.5) en considérant un type par neurone. Comme prévu, la tendance observée pour les configurations de types est la même que pour l'approximation du sinus. Lorsque la précision requise augmente, la précision des paramètres du réseau augmente également. Si seulement un chiffre de précision est requis, tous les paramètres peuvent être déclarés en précision half, et si au moins 11 chiffres sont requis, tous les paramètres doivent être en double précision. Le temps de calcul reste raisonnable quelle que soit la précision requise. Comme observé précédemment, le temps de calcul tend à augmenter avec la précision requise en raison du nombre de configurations testées par PROMISE.

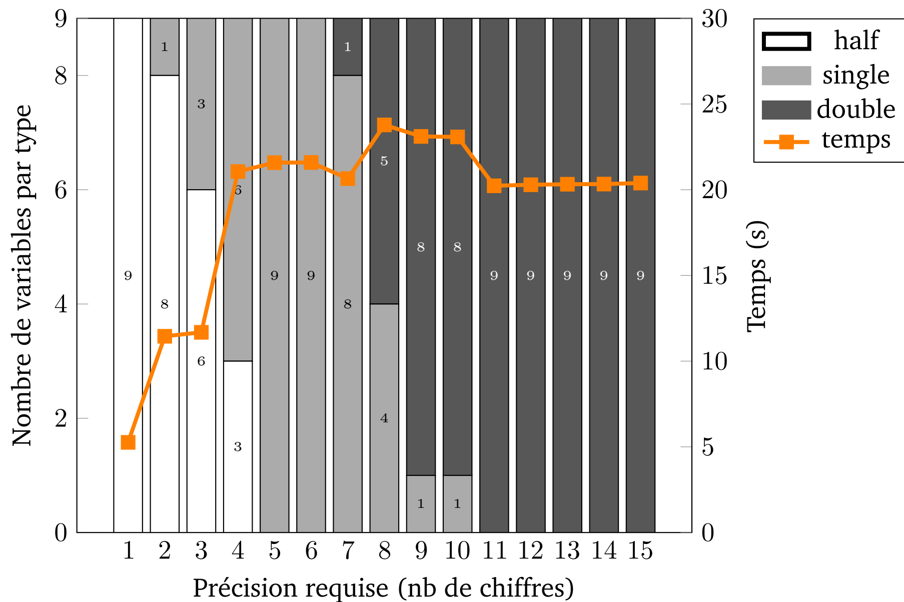

Figure 4.23. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau pendule inversé avec comme valeur d'entrée (0.5,0.5)

La Figure 4.24 présente les résultats obtenus en considérant un type par couche. Encore une fois, avec cette approche, le temps d'exécution de PROMISE est plus faible, mais la plupart des configurations sont en fait en précision uniforme. Les résultats en Figures 4.25 et 4.26 présentent les résultats obtenus avec une entrée composée de deux valeurs négatives

(-3,-6). Comme nous l'avons observé précédemment, le changement d'entrée n'induit aucune différence significative.

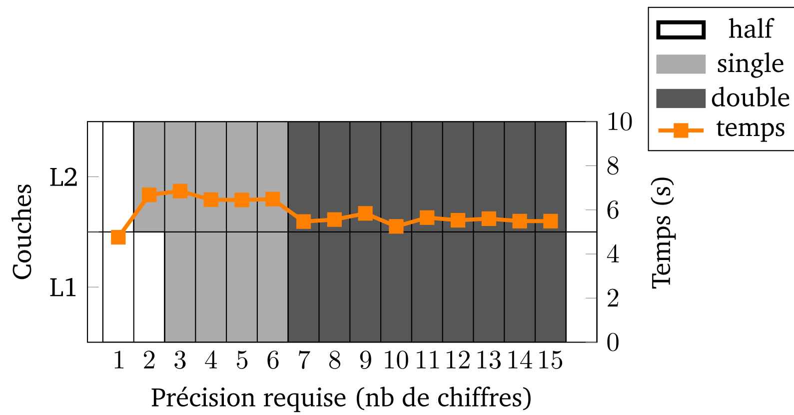

Figure 4.24. Précision de chaque couche pour le réseau pendule inversé avec comme valeur d'entrée (0.5,0.5)

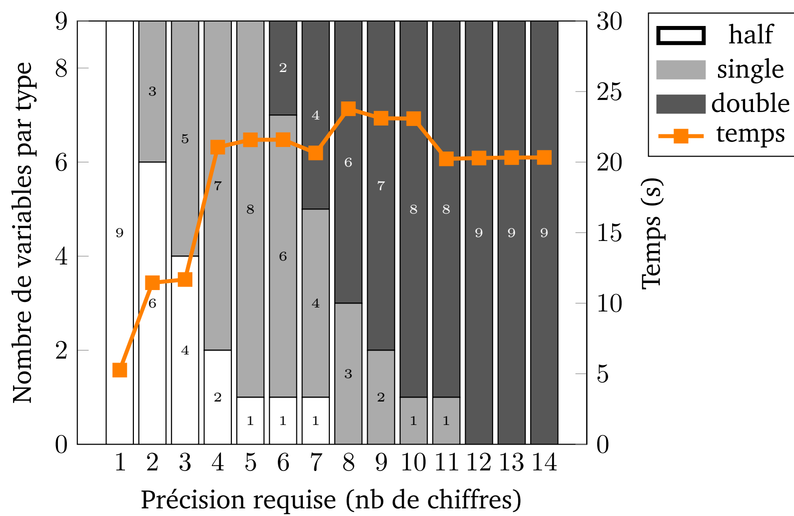

Figure 4.25. Nombre de variables PROMISE de chaque type et temps de calcul pour le réseau pendule inversé avec comme valeur d'entrée (-3,-6)

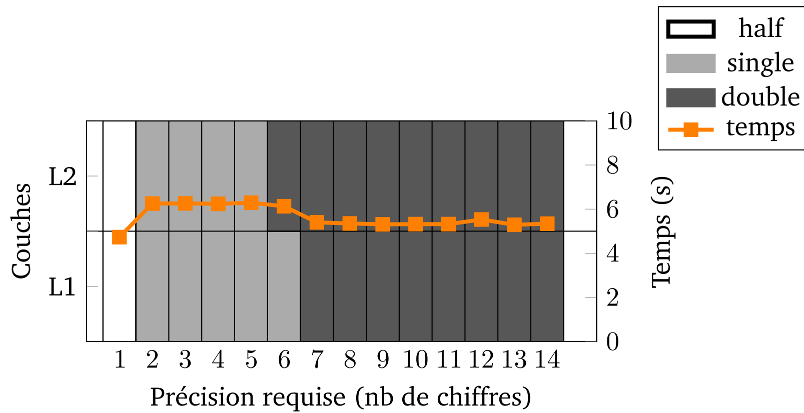

Figure 4.26. Précision de chaque couche pour le réseau pendule inversé avec comme valeur d'entrée (-3,-6)

4.3 PROMISE et entraînement

L'auto-tuning de précision des réseaux de neurones durant leur entraînement a aussi été réalisé. Cependant, plusieurs contraintes apparaissent dans cette situation, rendant les résultats difficiles à exploiter. Avant de présenter le travail réalisé dans la section 4.3.2, nous expliquons le fonctionnement de la phase d'apprentissage d'un réseau de neurones dans la section 4.3.1.

4.3.1 Entraînement de réseaux de neurones

L'entraînement d'un réseau de neurones consiste à trouver les poids et les biais appropriés pour le problème, c'est-à-dire de sorte que pour une entrée x, la sortie $\hat{y}$ soit une bonne approximation de la sortie attendue y. Pour cela, ces paramètres sont d'abord initialisés aléatoirement, suivant souvent une loi de distribution, comme par exemple la distribution uniforme de Glorot [**?**]. Celle-ci consiste à prendre des valeurs dans l'intervalle $[-\sqrt{6/(k+l)}, \sqrt{6/(k+l)}]$ pour chaque paramètre, avec k le nombre d'entrées de la couche respective et l le nombre de neurones de la couche respective (correspondant au nombre de sorties). L'objectif est ensuite de déterminer si chaque paramètre doit être augmenté ou abaissé. Pour cela, l'entraînement consiste à alterner des passes "en avant", où les données traversent le réseau de neurones afin de calculer la sortie, comme lors de l'inférence expliquée dans la Section 4.1, et des passes "en arrière", appelées *backward propagation*, qui servent à mettre à jour les paramètres du réseau. Afin de mettre à jour les paramètres, une fonction d'erreur, aussi appelée fonction de coût, est utilisée après chaque passe en avant. Celle-ci permet d'estimer l'exactitude du résultat en quantifiant l'écart entre les prédictions du modèle et les valeurs réelles des données d'entraînement. La fonction de coût la plus utilisé est l'erreur quadratique moyenne, ou MSE (*mean squared error*), définie comme suit :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

où n est le nombre de sorties du réseau, $\hat{y} \in \mathbb{R}^n$ est le vecteur des valeurs prédites par le réseau, $y \in \mathbb{R}^n$ est le vecteur des valeurs réelles d'entraînement. Une fois cette erreur calculée, on cherche à mettre à jour les poids et les biais de sorte à minimiser l'erreur. Pour cela, plusieurs méthodes de minimisation existent, comme par exemple la méthode des moindres carrés. La plus utilisée est la descente de gradient. Celle-ci consiste à calculer le gradient de l'erreur pour les paramètres, c'est-à-dire la dérivée partielle de l'erreur en fonction de chacun des paramètres, en utilisant la règle de dérivation en chaîne. Le gradient calculé donne la direction vers laquelle le paramètre doit être modifié (direction opposée au gradient). Les paramètres sont ensuite mis à jour avec un certain taux d'apprentissage α (*learning rate*) selon la formule :

$$W' = W - \alpha \nabla E$$

où W' est la matrice de poids après mise à jour, W est la matrice de poids avant mise à jour et ∇E le gradient calculé (la formule est identique pour la mise à jour des biais).

La phase d'apprentissage se termine ensuite lorsque le nombre d'itérations sur les données, ou le seuil d'erreur requis, est atteint. De plus, le jeu de données peut être découpé en lots (*batches*) sur lesquels l'algorithme va passer un certain nombre de fois (nombre d'*epochs*). Ces paramètres ne sont pas présentés ici. Plus de détails peuvent être trouvés dans la littérature [**?**, **?**].

4.3.2 Application de PROMISE

Comme évoqué précédemment, plusieurs contraintes apparaissent lors de l'application de PROMISE sur la phase d'apprentissage. Premièrement, afin de mener à bien cette étude, un programme C++ qui crée et entraîne un réseau de neurones a dû être écrit, contrairement à l'utilisation d'une bibliothèque en Python. Pour cela, le code disponible sur `https://github.com/Cr4ckC4t/neural-network-from-scratch.git` a été repris et adapté afin de pouvoir modifier la précision des paramètres de chaque couche. Ce code crée un réseau de neurones à deux couches s'entraînant à la classification d'un jeu de données à 3 classes sans équivalence réelle (i.e. le jeu de données ne correspond à aucune application réelle). Au départ, le réseau s'entraînait sur des découpages aléatoires du jeu de données en plusieurs paires jeu d'entraînement-jeu de test de manière successive. Cela permettait d'établir une moyenne des résultats de classification. Cela a été modifié afin d'entraîner le modèle sur un jeu d'entraînement découpé en plusieurs lots fixes, et non aléatoire, avec un jeu de test unique, et ainsi d'obtenir un seul résultat de classification et de supprimer le caractère aléatoire qui rendait les comparaisons avec le résultat de référence impossibles.

La seconde contrainte importante est l'initialisation aléatoire des paramètres au début de la phase d'entraînement. Cela ne peut pas être modifié, car mettre un poids identique pour tous les neurones revient à avoir un seul et unique neurone, la dérivée de la fonction coût donnant à chaque fois la même valeur dans ce cas. Le modèle serait alors équivalent à un

modèle linéaire. Avoir des poids choisis aléatoirement rend le résultat légèrement variable. Malgré cela, nous avons pu observer que passer notre réseau de neurones de double à simple, voire half, avait peu d'impact sur le résultat. Cela a été fait en considérant uniquement un type par couche, considérer un type par neurone rendant l'implémentation particulièrement complexe.

En effet, en prenant en compte les contraintes évoquées ci-dessus, nous avons dans un premier temps modifié le code pour qu'il soit le plus proche possible du réseau MNIST créé en Python. Pour cela, les poids sont initialisés selon la distribution uniforme de Glorot. Nos deux couches correspondent à celles du réseau de classification MNIST décrit précédemment : une de 64 neurones avec fonction d'activation ReLU et une de 10 neurones avec fonction d'activation softmax. Le critère de précision dans PROMISE a été changé, demandant à la place d'un nombre de chiffres significatifs exacts sur le résultat, de passer un pourcentage de classification correcte. En demandant jusqu'à 93% de classification correcte sur les données MNIST, nos deux couches peuvent être en half comme le montre la Figure 4.27. À partir de 94%, les résultats varient avec parfois un réseau qui ne peut satisfaire un taux de classification de 94% même en précision double, parfois un réseau qui satisfait ce critère en précision half, ou en précision mixte. Cela est dû aux différences d'initialisation du réseau à chaque test de nouvelle configuration, donnant des résultats légèrement différents.

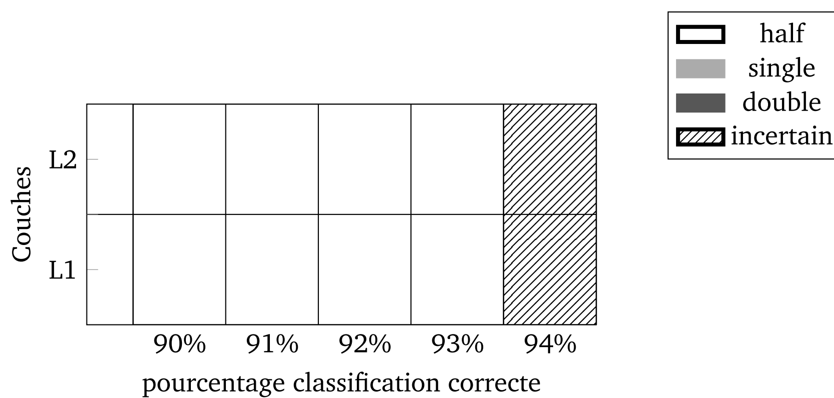

Figure 4.27. Précision de chaque couche pour le réseau MNIST en entraînement

Dans un second temps, nous avons modifié le réseau afin que celui-ci se rapproche du modèle d'approximation de la fonction sinus que nous avons précédemment étudié, mais avec 2 couches seulement : une première avec 20 neurones et la fonction d'activation $tanh$ comme précédemment, et une deuxième avec seulement 1 neurone et la fonction d'activation $tanh$. Dans cette situation, nous créons deux réseaux simultanément, un tout en double, qui devient notre réseau de référence, et un avec une précision soit simple soit double pour chaque couche (mais pas les deux en double). Les deux réseaux s'entraînent sur le même jeu de données de 800 valeurs, et nous comparons les résultats à la fin sur le même jeu de test comprenant 200 valeurs en prenant la plus grande erreur relative sur tous les résultats. Encore une fois, les résultats sont variables étant donné les neurones initialisés aléatoirement suivant une distribution de Glorot. Néanmoins, sur un nombre conséquent de lancers, et pour toutes les combinaisons de précision choisies avec uniquement les précisions simple

et double, nous n'avons pas obtenu d'erreur relative plus grande que de l'ordre de 10^{-2}. En considérant la racine de l'erreur quadratique moyenne ($RMSE = \sqrt{MSE}$, de l'anglais *root-mean-square error*), les différences sont tout aussi faibles avec selon les exécutions une différence entre les RMSE qui est au plus de l'ordre de 10^{-4}.

4.4 Conclusion

L'application de l'auto-ajustement de la précision sur les réseaux de neurones via l'outil PROMISE nous a permis de diminuer significativement la précision de ces derniers lors de leur phase d'inférence. Pour les réseaux d'interpolation, approchant le résultat d'un calcul, la précision de chaque neurone peut être abaissée selon la précision sur le résultat demandée. L'application aux réseaux de classification, au contraire, permet de mettre en évidence un groupe de variables pouvant garder une précision faible quelle que soit la précision demandée sur le résultat, bien que ce groupe puisse légèrement varier selon les situations. Ces paramètres pourraient donc probablement être supprimés (*pruning*).
L'application à la phase d'entraînement se révèle quant à elle contrainte par son caractère aléatoire. Néanmoins, la diminution de la précision de tous les paramètres en phase d'entraînement n'a pas de conséquence significative. Le réseau de classification s'entraînant en format half obtient tout autant de bons résultats que pour des précisions plus élevées, et le réseau d'interpolation testé en précision simple aussi par rapport à la précision double.

5 PROMISE et HPC

La précision réduite ou mixte offre des avantages en termes de temps d'exécution, d'utilisation de la mémoire et de consommation d'énergie. De nombreux algorithmes en précision mixte ont été proposés, notamment en algèbre linéaire, comme le montre l'étude [?]. La conception de tels algorithmes en précision mixte nécessite une bonne connaissance du calcul impliqué. Les outils d'auto-ajustement de la précision visent quant à eux à fournir une version en précision mixte d'un programme qui satisfait aux exigences de précision, quels que soient les algorithmes mis en œuvre.
L'article [?] introduit une suite de programmes de référence (*benchmark*) pour l'analyse des calculs en précision mixte. En outre, les auteurs présentent les performances de diverses stratégies d'auto-ajustement de la précision, tels que les algorithmes combinatoires (utilisés dans FloatSmith [?]), compositionnels (utilisés dans FloatSmith [?]), le Delta-Debug (utilisé dans Precimonious [?] et PROMISE [?]), la classification hiérarchique (utilisé dans CRAFT-HPC [?]), et un algorithme de recherche génétique (GA) (utilisé dans AMPT-GA [?]). L'algorithme du Delta-Debug nécessite plusieurs exécutions du programme pour fournir une configuration de types. Pour déterminer si une configuration est valide, différents tests peuvent être mis en œuvre. Dans PROMISE, qui s'appuie sur l'estimation de l'erreur d'arrondi, le critère porte sur la précision du résultat, alors que le test de Precimonious est basé à la fois sur un résultat de référence calculé avec la précision la plus élevée et sur le temps d'exécution du programme obtenu.
Alors que l'application de PROMISE aux réseaux de neurones a mis en évidence les intérêts en termes de diminution de la précision, nous étudions dans ce chapitre les bénéfices en termes de mémoire utilisée et de performance des codes obtenus. Malgré l'application réalisée uniquement sur ces codes, cela nous permet de mettre en avant les avantages que peut aussi apporter l'application de PROMISE sur des codes HPC plus importants, ainsi que les contraintes existantes.
Les performances des codes obtenus grâce à PROMISE sont présentées selon la précision requise sur le résultat. De plus, nous comparons des codes non vectorisés et vectorisés via l'utilisation d'instructions SIMD pour les performances en temps. Pour finir, nous présentons l'amélioration en temps de l'outil PROMISE lui-même via la parallélisation du Delta-Debug.

5.1 Performances en mémoire

Pour analyser les gains de mémoire possibles grâce à la précision mixte, nous considérons les quatre réseaux de neurones introduits précédemment ainsi que les différentes configurations de types obtenues avec PROMISE pour chacun d'entre eux.

5.1.1 Méthodologie

Différents outils permettent d'analyser l'utilisation de la mémoire, qu'il s'agisse de la quantité de mémoire échangée ou de la mémoire totale utilisée par le programme. Dans notre cas, une étude préliminaire à l'aide de l'outil Massif de Valgrind [1] nous permet de conclure que les valeurs théoriques de mémoire utilisée peuvent être considérées, c'est-à-dire les valeurs obtenues en comptant le nombre de variables de précision half, simple et double dans le programme, connaissant leurs tailles en octets. En effet, les valeurs de mémoire "useful-heap" données par Massif sont cohérentes avec les valeurs théoriques, un léger surplus de mémoire étant alloué lors de l'utilisation de pointeurs. Ce dernier ne représente pas plus de 1% de la mémoire totale utilisée et peut donc être ignoré. Par conséquent, les résultats suivants sont les valeurs théoriques compte tenu du nombre de variables en précision half, simple et double dans notre programme.

5.1.2 Résultats

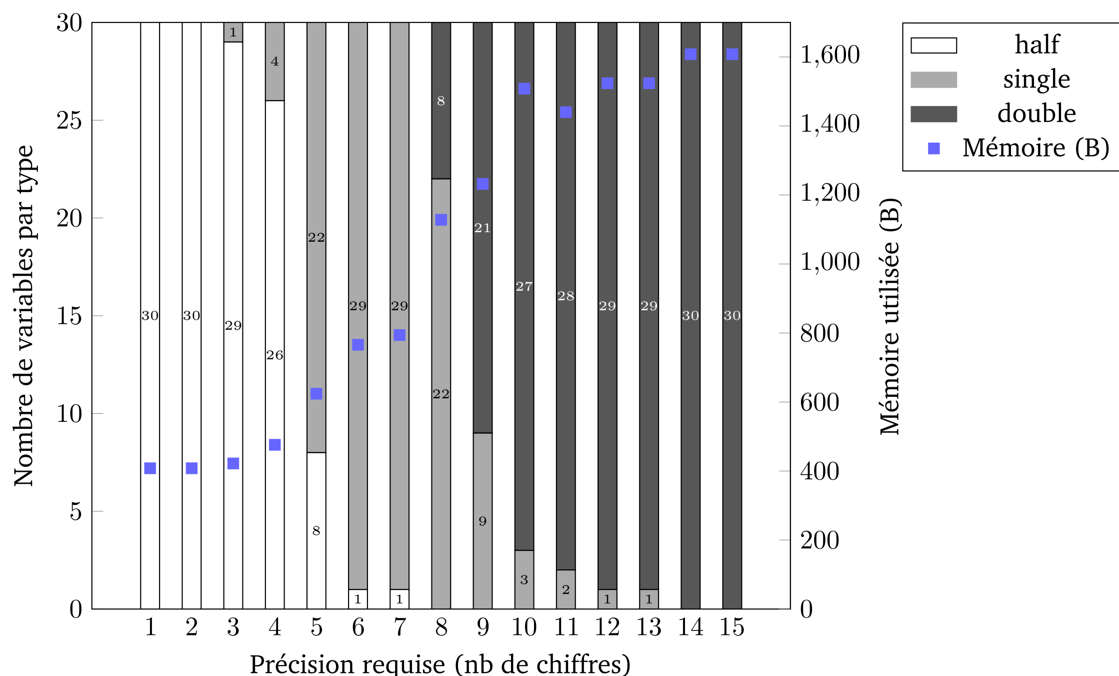

Figure 5.1. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau sinus avec comme valeur d'entrée 0.5

Les Figures 5.1 à 5.8 présentent, dans les programmes fournis par PROMISE pour chaque réseau de neurones, à la fois le nombre de variables de chaque type et la mémoire consommée, en fonction de la précision requise. Les Figures 5.1, 5.3, 5.5, et 5.7 ont été obtenues en considérant un type par neurone, et les Figures 5.2, 5.4, 5.6, et 5.8 en considérant un type par couche.

1. valgrind.org/docs/manual/ms-manual.html

Les données d'entrée sont précisées dans chaque légende correspondante. Comme évoqué au Chapitre 4, les résultats obtenus peuvent différer selon les données en entrée, néanmoins chaque réseau de neurones est présenté ici avec une seule donnée en entrée car nous considérons la mémoire théorique utilisée. Considérer d'autres données d'entrée n'apporte donc pas plus d'information que précédemment.
Comme prévu, le programme nécessite moins de mémoire lorsqu'il utilise une précision plus faible. L'approche par couche tend à utiliser plus de mémoire en raison des configurations obtenues avec cette approche. En effet, lorsqu'une variable d'une couche doit être plus précise, toute la couche doit être plus précise.

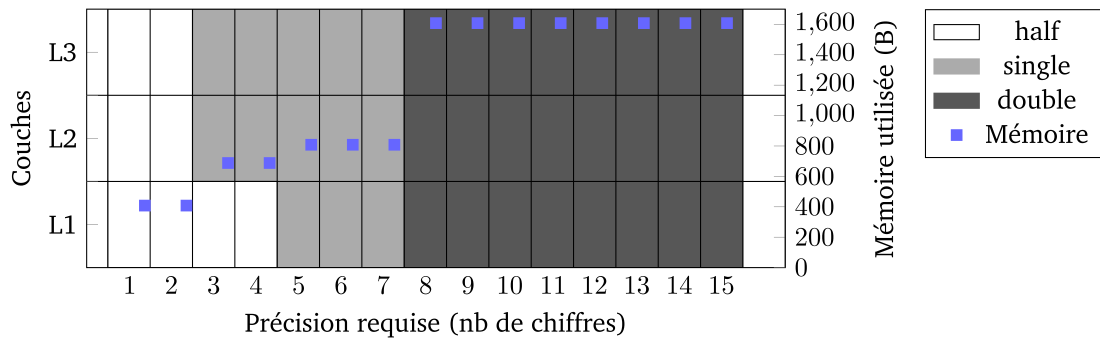

Figure 5.2. Précision de chaque couche et mémoire théorique utilisée pour le réseau sinus avec comme valeur d'entrée 0.5

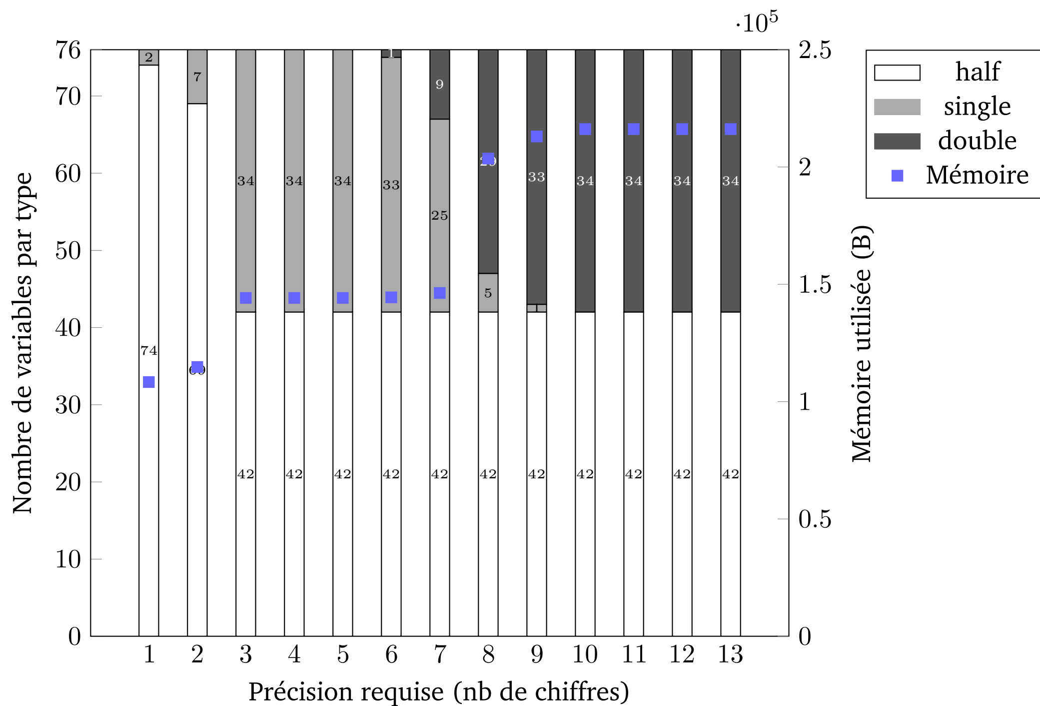

Figure 5.3. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau MNIST avec comme valeur d'entrée test_data[61]

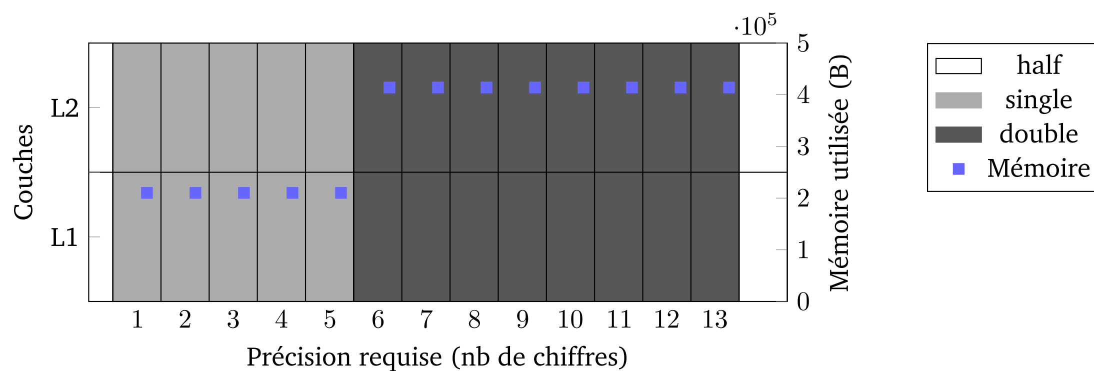

Figure 5.4. Précision de chaque couche et mémoire théorique utilisée pour le réseau MNIST avec comme valeur d'entrée test_data[61]

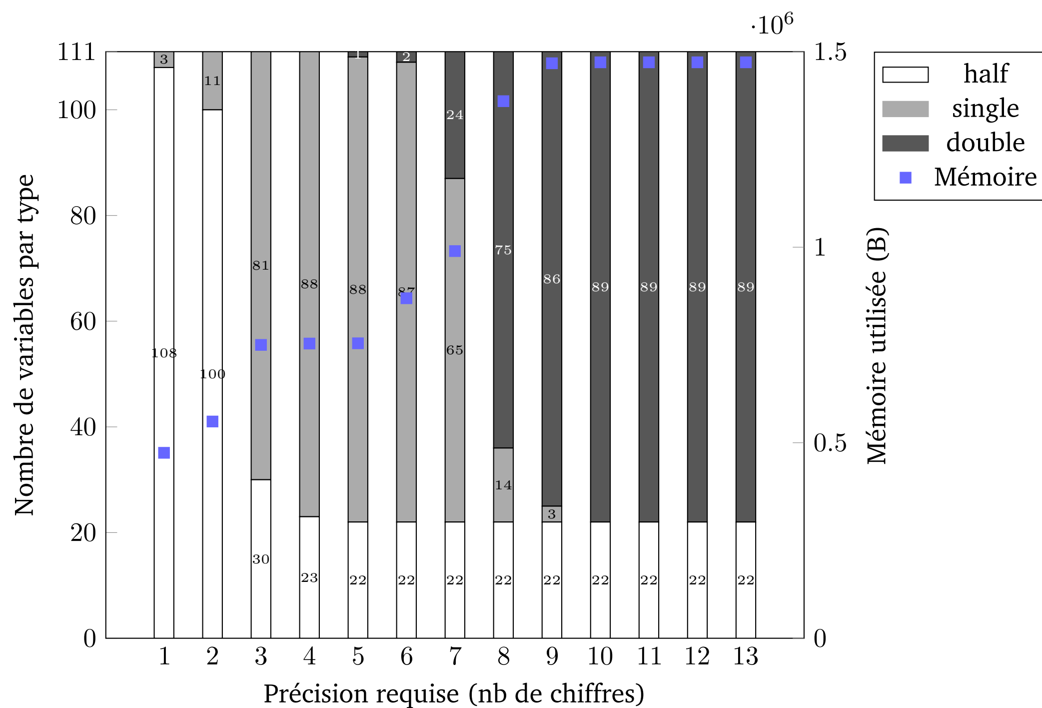

Figure 5.5. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau CIFAR avec comme valeur d'entrée test_data[386]

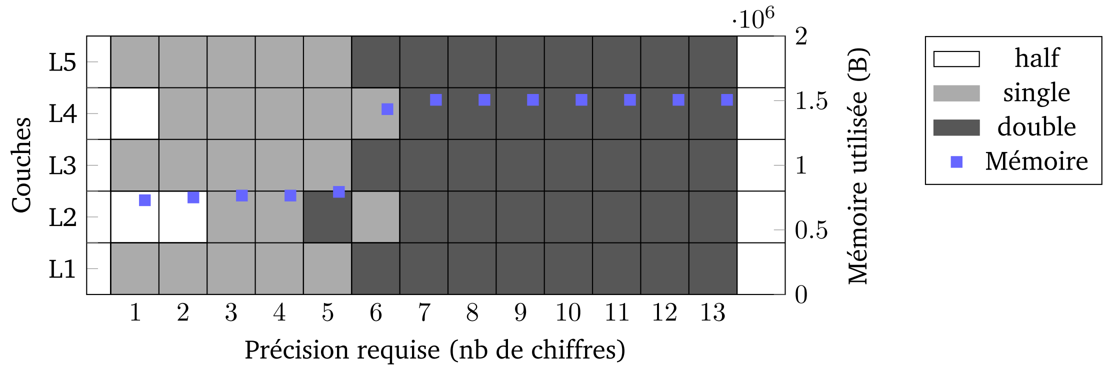

Figure 5.6. Précision de chaque couche et mémoire théorique utilisée pour le réseau CIFAR avec comme valeur d'entrée test_data[386]

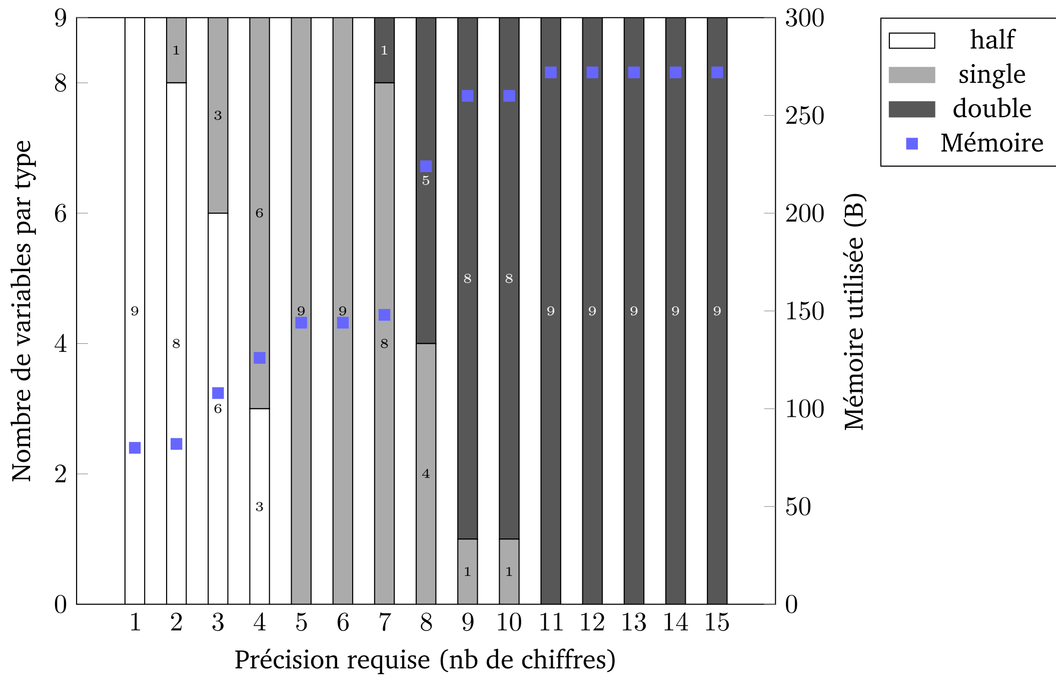

Figure 5.7. Nombre de variables PROMISE de chaque type et mémoire théorique utilisée pour le réseau pendule inversé avec comme valeur d'entrée (0.5,0.5)

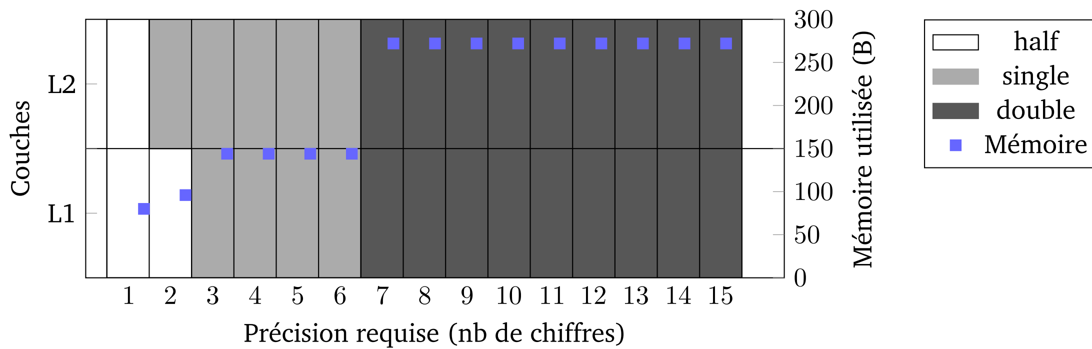

Figure 5.8. Précision de chaque couche et mémoire théorique utilisée pour le réseau pendule inversé avec comme valeur d'entrée (0.5, 0.5)

5.2 Performances en temps

5.2.1 Méthodologie

Nous analysons le gain en temps d'exécution dans les programmes en précision mixte. Pour cette analyse, nous relançons PROMISE sur nos réseaux de neurones en spécifiant utiliser uniquement les précisions simple et double, car la précision half n'est pas disponible en *hardware* sur l'architecture utilisée pour nos expériences, et l'utilisation d'une précision half émulée n'est pas compatible avec une analyse des performances en temps. Nous comparons les temps obtenus avec des codes vectorisés et non vectorisés, car la plupart des CPU modernes supportent les instructions SIMD. Pour ce faire, nous utilisons des instructions SIMD OpenMP et la vectorisation AVX2 sur les boucles qui calculent les différents produits matrice-vecteur (voir code Listing 5.1).

```
#pragma omp simd reduction(+:c)
for(unsigned i = 0; i < n; ++i){
        c += a[i] * b[i];
}
```

Listing 5.1 Exemple de code pour un produit scalaire vectorisé $c = a \cdot b$ avec a et b deux vecteurs de taille n.

Pour cette analyse, le type de l'entrée peut être modifié par PROMISE. En effet, dans nos expériences précédentes, toutes les variables d'entrée étaient en précision double, ce qui forçait toutes les opérations arithmétiques dans la première couche à être effectuées en précision double, avec éventuellement des opérations de cast sur les poids, ce qui peut être coûteux. Par conséquent, permettre l'ajustement des variables d'entrée est bénéfique pour le temps d'exécution des programmes fournis par PROMISE. Pour générer les différentes configurations avec du code vectorisé, nous modifions notre script de traduction du modèle vers le programme C++ pour ajouter les pragmas OpenMP appropriés. Nous pouvons ensuite appliquer directement PROMISE à notre programme C++ vectorisé. Pour activer la vectorisation, nous utilisons les options `-fopenmp-simd -avx2`. Nous compilons aussi avec l'option `-O3`. Nous mesurons les différents temps d'exécution à l'aide de la bibliothèque

C++ `time.h` en considérant la valeur minimale sur 10 000 exécutions du programme. En utilisant AVX2, il est possible d'effectuer simultanément 8 opérations en précision simple, mais seulement 4 en précision double. Une accélération théorique de 2 devrait donc être observée lors du passage d'un code vectorisé de la précision double à la précision simple.

5.2.2 Résultats

Nous présentons ici les résultats obtenus sur le réseau MNIST. Les Figures 5.9 et 5.10 montrent les différentes configurations et leur temps d'exécution compte tenu de la précision requise avec l'entrée test_data[61], respectivement avec l'approche par neurone et l'approche par couche. Nous rappelons qu'ici, le type de la variable d'entrée peut également être modifié. Tout d'abord, pour chaque configuration, nous mesurons le temps d'exécution de l'ensemble du calcul, c'est-à-dire le temps de la fonction *main* dans le code. Comme attendu, des différences peuvent être observées entre les codes vectorisés et non vectorisés. De plus, le temps d'exécution dépend de la précision utilisée. D'après les Figures 5.9 et 5.10, l'approche par couche fournit plus de configurations en précision uniforme que l'approche par neurone, comme déjà mentionné dans le chapitre précédent. En effet, en fonction de la précision requise, toutes les variables peuvent être en précision simple ou double. L'accélération d'une exécution en précision simple par rapport à une exécution en précision double est au maximum de 1.60 pour les codes non vectorisés, et de 1.86 pour les codes vectorisés. La meilleure accélération obtenue est donc légèrement inférieure au ratio théorique de 2. Nous pouvons également comparer les codes en précision mixte et en précision double. Nous nous concentrons ici sur la configuration en précision mixte obtenue avec une précision requise de 10 à 13 chiffres et l'approche par neurone (voir Figure 5.9). Le rapport en temps d'exécution en précision mixte par rapport à l'exécution en précision double est au maximum de 1.30 pour les codes non vectorisés et de 1.41 pour les codes vectorisés. Néanmoins, si la configuration comporte une majorité de variables en précision double, le gain de vitesse peut être moins bon. En effet, dans ce cas, la majorité des opérations sont effectuées en double précision et des opérations de cast coûteuses sont réalisées. En comparant le code vectorisé et non vectorisé dans les Figures 5.9 et 5.10, nous observons un facteur d'accélération jusqu'à 1.45. Ce rapport n'est pas en accord avec le facteur d'accélération théorique des codes vectorisés par rapport aux codes non vectorisés sur les unités AVX2 : 8 en précision simple et 4 en précision double. Cela s'explique par le fait qu'une partie importante du code ne bénéficie pas réellement de la vectorisation. C'est notamment le cas de la fonction d'activation *softmax* dans la dernière couche, qui calcule une exponentielle à l'aide de la bibliothèque `cmath` C++.

Par conséquent, nous présentons dans les Figures 5.11 et 5.12 uniquement le temps d'exécution des produits matrice-vecteur exécutés dans le code. Dans ce cas, nous observons une accélération allant jusqu'à 7.2 lorsque nous comparons les codes vectorisés et non vectorisés en simple précision, et une accélération allant jusqu'à 3.9 en double précision. En comparant la précision simple et la précision double dans les codes vectorisés, nous obtenons une accélération allant jusqu'à 2, soit le ratio théorique.

Dans la Figure 5.12, nous n'observons aucune différence de temps entre l'exécution en précision simple et l'exécution en précision double des produits matrice-vecteur non vectori-

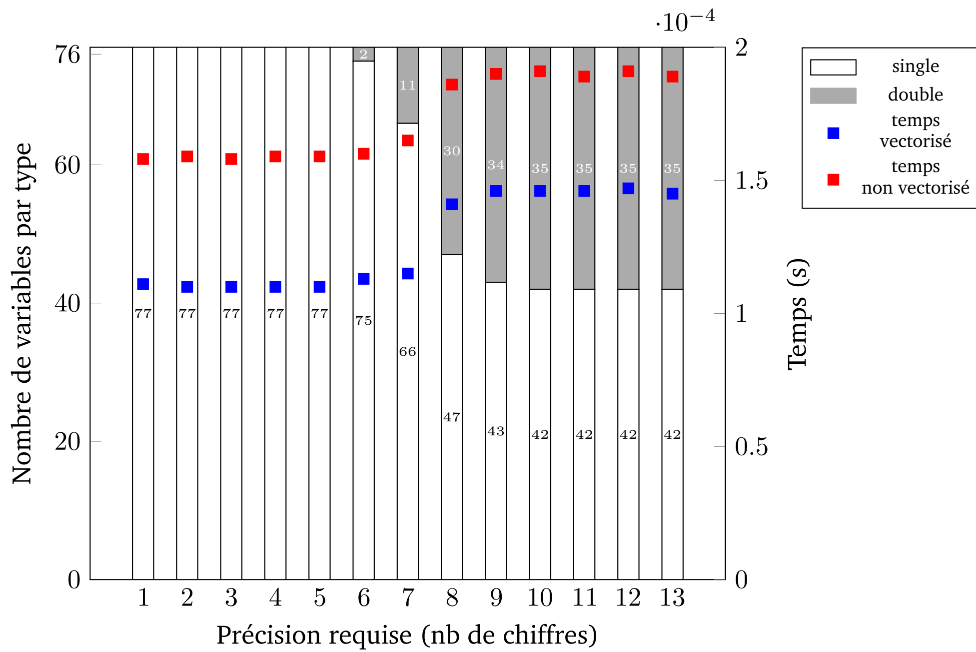

Figure 5.9. Nombre de variables PROMISE de chaque type et temps d'exécution total pour le réseau MNIST avec comme valeur d'entrée test_data[61]

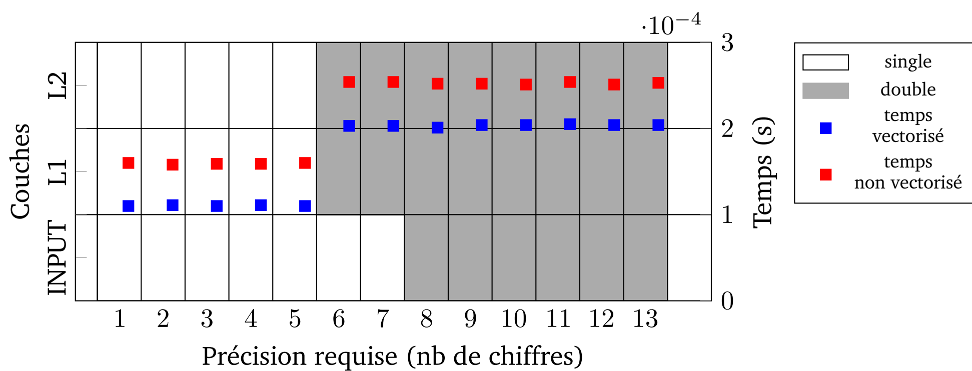

Figure 5.10. Précision de chaque couche et temps d'exécution total pour le réseau MNIST avec comme valeur d'entrée test_data[61]

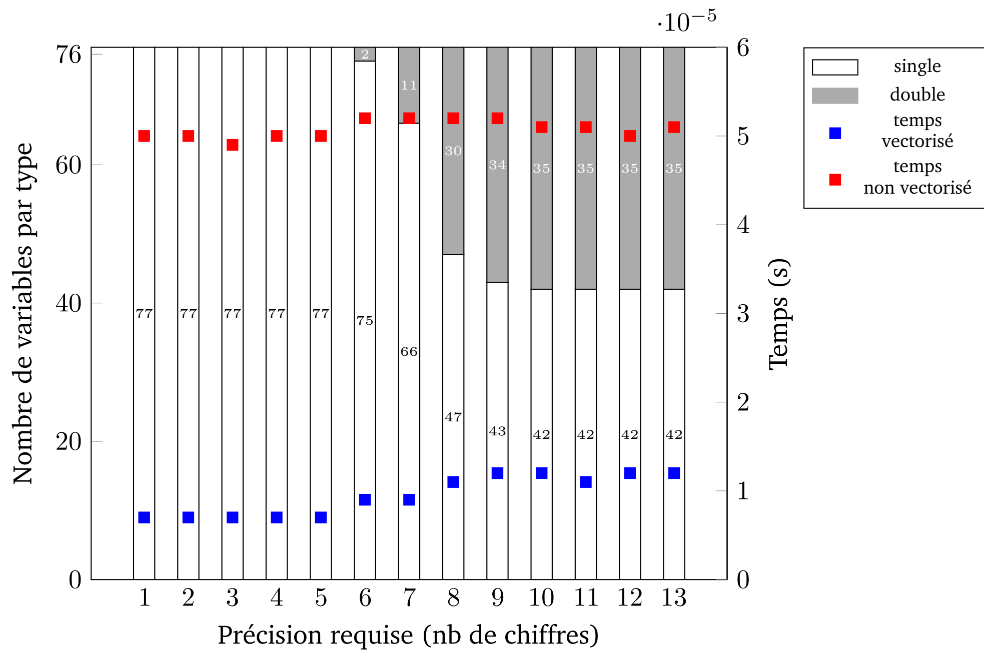

Figure 5.11. Nombre de variables PROMISE de chaque type et temps d'exécution des produits matrice-vecteur pour le réseau MNIST avec comme valeur d'entrée test_data[61]

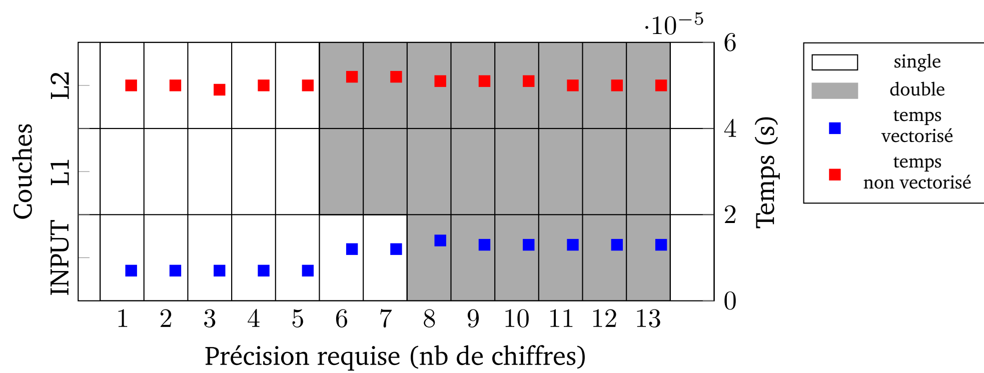

Figure 5.12. Précision de chaque couche et temps d'exécution des produits matrice-vecteur pour le réseau MNIST avec comme valeur d'entrée test_data[61]

sés. Toutefois, la Figure 5.10 met en évidence une accélération entre le temps d'exécution en simple et le temps d'exécution en double du code non vectorisé jusqu'à 1.60. Cette accélération est due aux avantages de l'utilisation de la précision simple lors de la déclaration et de l'initialisation des variables (dans notre cas, tous les poids et les biais). Dans cette partie du code, le rapport de temps d'exécution en précision simple par rapport à l'exécution en précision double est en fait de 2, le ratio théorique.

5.3 Amélioration des performances de PROMISE

Alors que la Section 5.2 analysait les performances des codes générés par PROMISE, cette section est consacrée à l'amélioration des performances de l'outil PROMISE lui-même. La recherche d'une configuration appropriée par l'algorithme de Delta-Debug nécessite de multiples compilations et exécutions de versions en précision mixte du code utilisateur. C'est pourquoi nous avons parallélisé l'algorithme de Delta-Debug dans PROMISE.
Pour cela, nous utilisons la parallélisation décrite dans [?] et l'implémentation correspondante dans l'outil Picire [2]. Cette parallélisation repose sur le module `multiprocessing` de Python qui permet d'exécuter une fonction avec plusieurs valeurs d'entrée, en distribuant les données dans différents processus en parallèle.

La Figure 3.1 dans le Chapitre 3 présente le fonctionnement de l'algorithme de Delta-Debug pour la création de différentes configurations avec deux types différents. L'idée de la parallélisation est de tester plusieurs configurations en parallèle à un niveau de l'algorithme de Delta-Debug. Lorsqu'une configuration répond à nos exigences, nous pouvons arrêter de passer en boucle sur les différentes configurations. Le nombre de configurations à tester en parallèle peut être spécifié. Dans notre cas, nous testons 6 configurations en parallèle, car 6 cœurs sont disponibles sur notre machine.

Les Figures 5.13 et 5.14 montrent le temps d'exécution de PROMISE pour MNIST et CIFAR en utilisant les versions séquentielles et parallèles de l'algorithme de Delta-Debug. Nous utilisons ici l'approche par neurone, plus coûteuse que celle par couche. Pour MNIST, nous observons une accélération allant jusqu'à 3.2, et pour CIFAR une accélération allant jusqu'à 2.7.

Nous pouvons remarquer que, pour MNIST, si 1 ou 2 chiffres sur le résultat sont requis, l'algorithme séquentiel est plus performant que l'algorithme parallèle (voir Figure 5.13). Pour CIFAR, lorsqu'un chiffre est requis, les temps d'exécution en séquentiel et en parallèle sont très proches (voir Figure 5.14). En effet, la version parallèle peut ne pas être satisfaisante lorsque la précision demandée est faible, car, dans ce cas, l'algorithme de Delta-Debug n'exécute que quelques étapes, ce qui s'avère plus long à cause de la surcharge liée à la parallélisation.

2. https://github.com/renatahodovan/picire

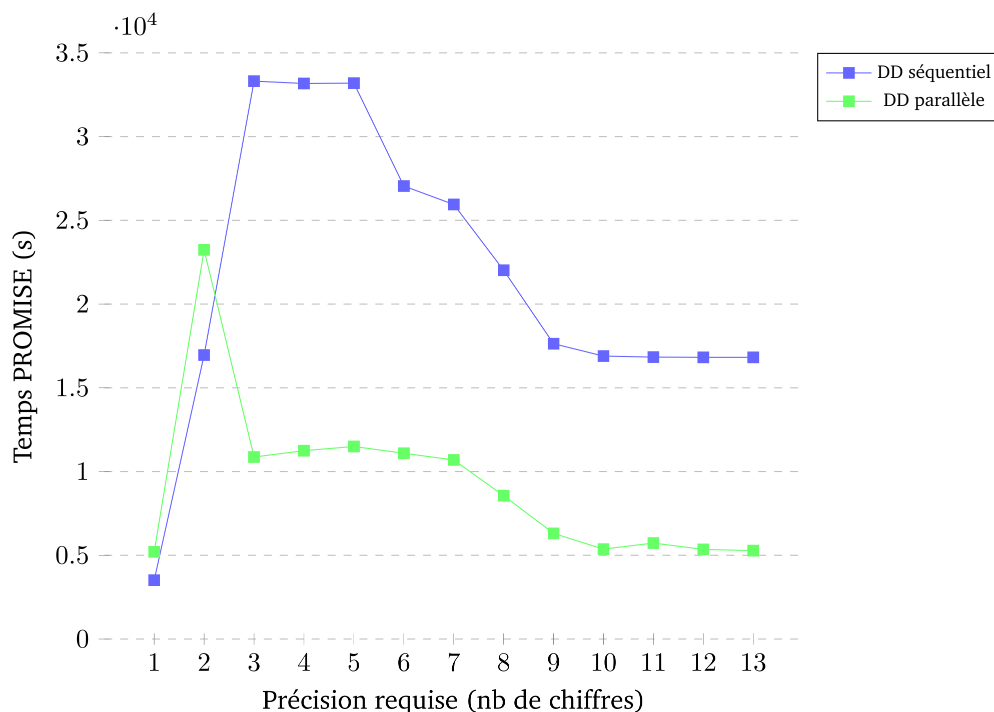

Figure 5.13. Temps de calcul de PROMISE en utilisant les différentes versions du Delta-Debug sur le réseau de neurones MNIST avec comme valeur d'entrée test_data[61]

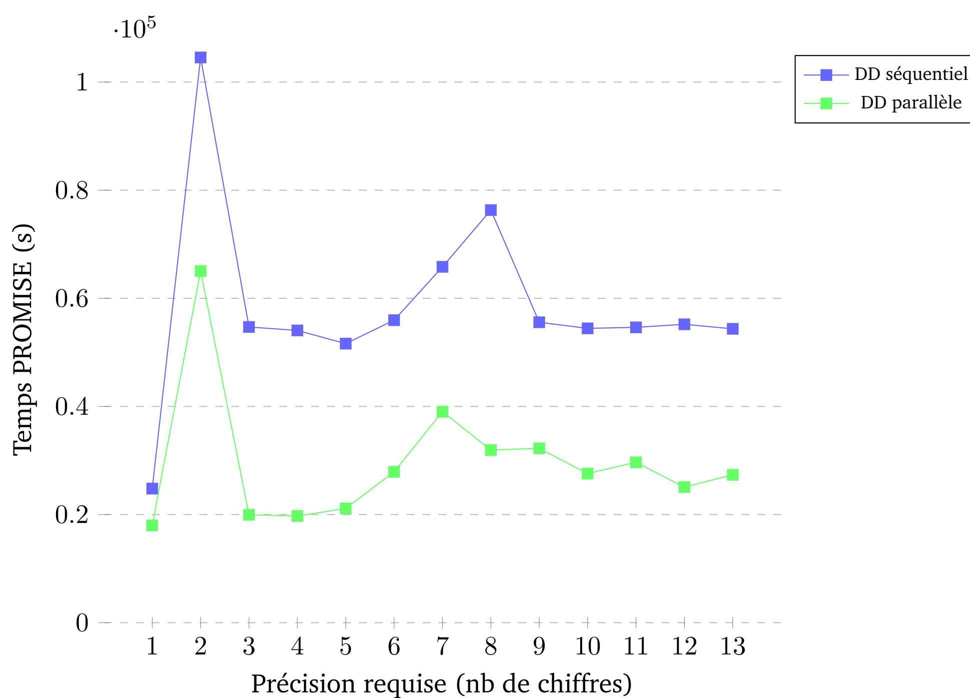

Figure 5.14. Temps de calcul de PROMISE en utilisant les différentes versions du Delta-Debug sur le réseau de neurones CIFAR avec comme valeur d'entrée test_data[386]

5.4 Conclusion

Grâce à l'application de l'auto-tuning de précision sur différents réseaux de neurones, nous avons montré les avantages en termes de consommation de mémoire et de temps d'exécution de la réduction de la précision des variables à virgule flottante. Des gains significatifs en mémoire ont pu être déduits de la taille de chaque format flottant. En effet, la précision simple permet de consommer deux fois moins de mémoire que la précision double, et la précision half quatre fois moins.
Pour les mesures de temps, PROMISE a été utilisé pour générer des codes mélangeant les deux précisions simple et double disponibles en *hardware* dans le matériel utilisé pour nos expériences. L'utilisation de la précision simple est bénéfique pour la déclaration et l'initialisation des variables, et particulièrement avantageuse dans les parties vectorisées des codes avec dans le deux cas un gain de temps de facteur 2 par rapport à la précision double.
Enfin, nous avons vu que l'algorithme du Delta-Debug dans PROMISE qui teste les différentes configurations de types dans le code utilisateur a un fort impact sur le temps d'exécution de PROMISE. La parallélisation de celui-ci permet de réduire cet impact par un facteur jusqu'à 3.2 sur l'architecture utilisée (processeur Intel Core i5-8400 2.80 GHz, 6 curs, 16Go de RAM).

6 Instrumentation de codes pour la validation et l'auto-ajustement de la précision

Outre les résultats présentés jusqu'ici, des améliorations plus techniques ont été apportées à PROMISE afin de rendre ce dernier plus performant. Celui-ci a notamment été rendu compatible avec les codes MPI, et peut désormais considérer le format bfloat16 à la place du format fp16, ou encore bénéficier de la parallélisation du Delta-Debug présentée en Section 5.3.

Nous présentons ici des améliorations liées à l'instrumentation de codes, que ce soit pour CADNA ou pour PROMISE. En effet, à l'origine, l'instrumentation pour CADNA se faisait grâce à un script Perl et l'instrumentation pour PROMISE se faisait à la main. Le script Perl n'est pas robuste et constitue une solution *ad hoc*, pouvant présenter des manquements. L'instrumentation à la main pour PROMISE n'est tout simplement pas pratique, surtout pour l'application sur des codes HPC de taille importante. Nous présentons donc ici un outil d'instrumentation basé sur la représentation syntaxique du compilateur Clang. Celui-ci est adapté en deux versions, une pour CADNA, une pour PROMISE. De la même manière, le *parsing* de codes écrit en Python dans PROMISE, c'est-à-dire l'analyse de codes pour récupérer les informations des variables PROMISE et la création de nouveaux codes, est lui aussi remplacé par une version de notre outil basé sur Clang. Encore une fois, le script Python est *ad hoc* et peut présenter des manquements. De plus, ouvrir la représentation syntaxique Clang pour un fichier et l'utiliser tout au long de l'exécution de PROMISE peut permettre un gain de temps considérable. De manière générale, l'utilisation de Clang représente une solution robuste basée sur la représentation syntaxique, prenant en compte les possibles mises à jour C/C++, et enfin accessible.

Nous commençons par décrire les outils LLVM et Clang, ainsi que le fonctionnement de la bibliothèque permettant l'instrumentation de codes, en nous concentrant sur les composantes utilisées. La documentation de la bibliothèque complète peut être trouvée sur `https://clang.llvm.org/docs/LibTooling.html`. Nous décrivons ensuite l'instrumentation et les spécificités des versions pour CADNA puis PROMISE de notre outil.
Dans tout ce chapitre, nous utilisons comme verbe l'anglicisme *matcher*, du verbe *to match* qui peut être traduit par "trouver et récupérer la correspondance" [1]. Par exemple, on peut matcher la fonction `f` dans le code, ce qui permet non seulement de la trouver, mais aussi d'en récupérer les informations.

1. la traduction littérale peut être "correspondre (à)" ou "concorder", mais cette traduction perd l'idée de récupérer l'information de ce qui est "matché".

6.1 LLVM et Clang

LLVM est un ensemble de bibliothèques, compilateurs et éléments d'optimisation de la chaîne de compilation basés sur la modularité. À l'origine, c'était un projet de compilateur moderne utilisant une forme statique à affectation unique, ou SSA (*Static Single Assignment form*), représentation intermédiaire d'un code source forçant chaque variable à être affectée une seule fois. Aujourd'hui, LLVM est devenu un énorme projet englobant de nombreux sous-projets utilisés dans l'industrie comme dans la recherche académique, et permettant aussi le développement open source de sous-projets. Les bibliothèques principales de LLVM fournissent un outil d'optimisation indépendant de la source et de la cible ainsi qu'un générateur de code construit autour de la représentation intermédiaire LLVM (IR LLVM). Le générateur de code permet de traduire cette représentation intermédiaire en code assembleur ou en code binaire. Parmi les sous-projets les plus importants de LLVM, nous retrouvons notamment une implémentation de la bibliothèque C++, libc++, un "débugger", LLDB, ainsi qu'un compilateur de codes C/C++/Objective-C, Clang.
En plus d'être un compilateur rapide et peu gourmand en mémoire, Clang fournit aussi de nombreuses bibliothèques C/C++ comme Clang Static Analyzer, un outil d'analyse statique au niveau du code source. Celui-ci permet de trouver automatiquement des bogues dans un code. Nous retrouvons aussi des frameworks comme Clang-Tidy qui permet une analyse de code poussée, et qui inclut notamment Clang Static Analyzer. Clang-Tidy est fondé sur la bibliothèque LibTooling de Clang, qui permet d'écrire des outils autonomes basés sur Clang et travaillant notamment sur l'arbre syntaxique (AST). C'est cette bibliothèque que nous utilisons pour l'instrumentation de codes au sein de notre outil.

6.2 Instrumentation LLVM

6.2.1 Arbre de la Syntaxe Abstraite (AST) Clang

La bibliothèque LLVM LibTooling permet d'écrire un outil autonome basé sur Clang et de travailler sur l'arbre de la syntaxe abstraite d'un code, aussi appelé arbre syntaxique, ou AST de l'anglais *Abstract Syntax Tree*. L'AST est une version allégée du *concrete syntax tree* qui représente de manière exacte et détaillée la grammaire du code analysé sous forme d'arbre. Ce dernier n'est cependant jamais construit et utilisé, car très formel et donc peu pratique. L'AST suffit au compilateur pour que celui-ci génère la représentation intermédiaire après une phase d'analyse sémantique et de vérification des règles de grammaire.
L'AST se compose de nœuds représentant les différentes composantes (déclaration de fonctions, déclaration de variables, opération, instruction `if`, appel de fonction, etc.) du code source analysé. Ces derniers fonctionnent sur un principe hiérarchique. Parmi les nœuds de base, nous retrouvons les `Stmt`, pour *Statement*, qui correspondent aux fragments d'un programme qui sont exécutés en séquence. Toutes les lignes d'une fonction sont par exemple de type `Stmt`. Nous retrouvons aussi les déclarations, `Decl`, parent notamment des déclarations de fonctions `FunctionDecl` et des déclarations de variables `VarDecl`, et les expressions, `Expr`, parent notamment des opérations de cast `ExplicitCastExpr` et des appels aux fonc-

tions `CallExpr`. Si un `Decl` ou `Expr` est aussi un `Stmt`, alors les types `DeclStmt` et `ExprStmt` peuvent être utilisés. Nous pouvons remarquer que les noms des types dans l'AST sont explicites.

Nous donnons en Figure 6.1 un exemple d'AST simplifié pour l'expression "`double result = x * 42.0 + 12.0`". Comme expliqué, nous pouvons voir que la ligne entière est un `DeclStmt`, qui inclut une déclaration de variable `VarDecl` (de la variable `result`), dont la valeur est définie par les opérations qui suivent. L'arbre complet de cette même opération placé dans une simple fonction est donné dans le Listing 6.1, obtenu avec l'option `-ast-dump`. Dans celui-ci, nous pouvons voir qu'un AST clang commence toujours par une `TranslationUnitDecl`, puis nous retrouvons notre arbre avec la hiérarchie symbolisée par des tabulations et le caractère "`|`". Les nœuds présentent aussi beaucoup plus d'informations, notamment leur position ainsi que le type pour les déclarations et les opérations.

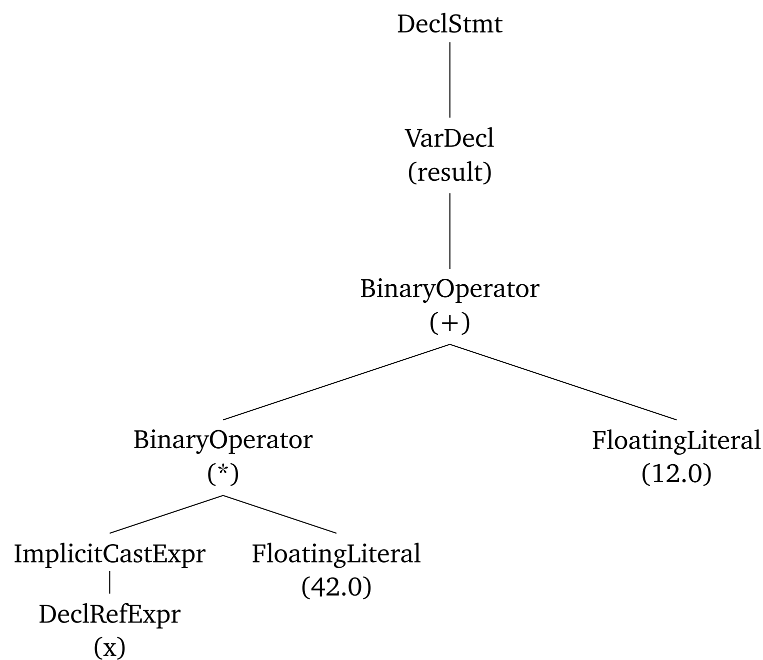

Figure 6.1. Représentation graphique simplifiée de l'AST pour l'expression "`double result = x * 42.0 + 12.0;`"

Pour parcourir l'AST, il suffit de partir de la `TranslationUnitDecl`, la déclaration d'origine, et de parcourir récursivement tous les nœuds suivants. Pour trouver un nœud dans l'AST, il est aussi possible d'utiliser les matchers, qui récupèrent dans le code le nœud désiré.

6.2.2 Matchers et Replacements

Il existe 3 types de matchers :

— les *Node Matchers*, qui matchent un type de nœud dans l'AST ;

— les *Narrowing Matchers*, qui matchent des attributs d'un nœud dans l'AST ;

— les *Traversal Matchers*, qui permettent la correspondance entre deux nœuds de l'AST.

```
$ cat test.cc
double f(double x) {
    double result = x * 42.0 + 12.0;
    return result;
}

# Clang est un front end pour de nombreux outils ; -Xclang permet de
# passer des options directement au front end C++.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x12d844e08 <<invalid sloc>>
... declarations internes de clang ...
`-FunctionDecl 0x12d8e37a0 <test.cc:1:1, line:4:1> line:1:8 f 'double (doub
le)'
  |-ParmVarDecl 0x12d8e36d0 <col:10, col:17> col:17 x 'double'
  `-CompoundStmt 0x12d8e3a20 <col:20, line:4:1>
    |-DeclStmt 0x12d8e39c0 <line:2:5, col:29>
    | `-VarDecl 0x12d8e38a8 <col:5, col:30> col:12 result 'double' cinit
    |   `-BinaryOperator 0x1318095a8 <col:21, col:32> 'double' '+'
    |     |-BinaryOperator 0x131809568 <col:21, col:25> 'double' '*'
    |     | |-ImplicitCastExpr 0x131809550 <col:21> 'double' <LValueToRValue
>
    |     | | `-DeclRefExpr 0x131809510 <col:21> 'double' lvalue ParmVar 0x
1318092d0 'x' 'double'
    |     | `-FloatingLiteral 0x131809530 <col:25> 'double' 4.200000e+01
    |     `-FloatingLiteral 0x131809588 <col:32> 'double' 1.200000e+01
    `-ReturnStmt 0x131809618 <line:3:5, col:12>
      `-ImplicitCastExpr 0x131809600 <col:12> 'double' <LValueToRValue>
        `-DeclRefExpr 0x1318095e0 <col:12> 'double' lvalue Var 0x1318094a8
'result' 'double'
```

Listing 6.1 Exemple d'AST (documentation Clang (https://clang.llvm.org/docs/IntroductionToTheClangAST.html) modifié avec type flottants)

Un matcher spécifique est toujours constitué d'un Node Matcher, permettant de matcher un nœud dans l'AST, éventuellement combiné à des Narrowing Matchers et Traversal Matchers rajoutant des conditions sur les nœuds à matcher. Par exemple, il est possible de matcher spécifiquement la fonction de nom `f` grâce au matcher ci-dessous, composé d'un Node Matcher qui matche tous les nœuds FunctionDecl, et d'un NarrowingMatcher ajoutant comme condition que la fonction ait comme nom `f`.

```
auto myMatcher = functionDecl(hasName("f"));
```

Notons que les Node Matchers sont souvent nommés comme le type qu'ils permettent de matcher, avec simplement un caractère minuscule au début, au lieu d'une majuscule. Par exemple ici le matcher `functionDecl` permet de matcher les types `FunctionDecl`. Les Narrowing Matchers et Traversal Matchers sont aussi souvent bien nommés, permettant de faire des sortes de phrases au sein du matcher et de comprendre facilement ce qu'il matche, comme ici avec le matcher `hasName`.

Il est aussi possible d'ajouter un identifiant aux matchers de nœuds grâce à la fonction `bind("id")`. Cela permet de récupérer facilement les informations du matcher depuis le *callback*. En effet, à chaque fois qu'un matcher trouve un nœud lui correspondant, une fonction callback est appelée permettant d'exécuter les actions désirées lors du match en question. Associer une chaîne de caractères à un matcher permet de facilement savoir dans le callback, commun aux matchers, quel match a engendré l'appel. Par exemple, on peut ajouter l'identifiant `"f"` à notre matcher sur la fonction `f` comme suit :

```
auto myMatcher = functionDecl(hasName("f")).bind("f");
```

Dans notre cas, le callback sera principalement utilisé pour construire des `Replacement`, permettant d'appliquer un changement textuel dans le fichier source. Pour cela, le premier argument pour construire un `Replacement` est le `SourceManager` qui gère le chargement et la mise en cache des fichiers source en mémoire. Nous pouvons ainsi effectuer le remplacement d'un nœud par une chaîne de caractères donnée en argument, ou de même un remplacement sur la plage [`Start`, `Start + length`) dans le fichier source avec une position `Start` (de type `SourceLocation`) et une taille de remplacement. Dans certains cas, nous utiliserons des `Replacement` sans `SourceManager`, qui prennent alors comme argument le chemin absolu du fichier et appliquent le remplacement sur la plage [`Offset`, `Offset+length`).
Le Listing 6.2 donne un exemple de callback pour notre matcher sur la fonction `f`, qui change la fonction `f` en une fonction `g` qui retourne simplement 0.

```
virtual void run(const MatchFinder::MatchResult &Result) {
    auto SM = Result.SourceManager;

    if(auto fDecl = Result.Nodes.getNodeAs<clang::FunctionDecl>("f")) {
        Replacement repl(*SM, fDecl, "double g(){\n    return 0;\n}");

        if((*repls)[filename].add(repl)){
            llvm::outs() << "replacement failed\n";
        }
    }
}
```

Listing 6.2 Example de callback pour un matcher sur une fonction `f`

Ce `Replacement` est ajouté à la liste des `Replacements` du `RefactoringTool`, qui n'est rien d'autre qu'une `map` associant les remplacements au nom de fichier. Le `RefactoringTool` est l'outil Clang qui gère ensuite les remplacements textuels au sein des fichiers, appliquant simplement les `Replacements` donnés dans la liste.

Dans notre cas, nous cherchons à remplacer principalement les types des variables à virgule flottante. Pour cela, nous définissons un matcher appelé `FloatTypeLocMatcher` qui matche les types flottants présents dans le code source via le Node Matcher `typeLoc`, pour *type location*. Nous lui donnons l'identifiant `"type"`. Ce dernier devient une condition pour les matchers suivants grâce au Narrowing Matcher `hasTypeLoc()` prenant en paramètre notre matcher sur les `TypeLoc` ayant un type flottant. Par exemple, dans le matcher défini précédemment sur la fonction `f`, nous pouvons ajouter cette condition ainsi qu'un identifiant comme dans l'exemple ci-dessous. Nous obtenons ainsi un matcher sur la fonction `f` seulement si celle-ci a un type flottant en type de retour.

```
auto myMatcher = functionDecl(hasTypeLoc(FloatTypeLocMatcher), hasName("f")
    ).bind("function");
```

Par la suite, dans le callback, il nous est facile de récupérer le `TypeLoc` de la déclaration de fonction en utilisant la fonction `Result.Nodes.getNodeAs` et l'identifiant `"type"`, et ainsi de créer un remplacement du type comme suit :

```
Replacement myRepl(*SM, Result.Nodes.getNodeAs<clang::TypeLoc>("type")), "
    float");
```

Ici, le nœud récupéré par la fonction `getNodeAs` est remplacé par la chaîne de caractères "float".

De nombreux matchers ont été ainsi créés selon les besoins, pouvant récupérer le type d'une déclaration de variable, globale ou non, le type d'une déclaration de fonction, le type d'un paramètre de fonction, le type d'un paramètre de structure, le type d'une définition d'alias ou encore le type d'un cast explicite. Des matchers ont aussi été créés sans condition sur le type, permettant de matcher les fonctions d'affichage de C/C++ (`printf`, `asprintf`, `fprintf`, etc.) ou les fonctions MPI.
Les fonctions du `Lexer` de Clang nous seront aussi utiles, transformant le code source en flux de *token* [2] et permettant de le parcourir caractère par caractère, et ainsi d'y faire des modifications plus précises.
Notons enfin que l'AST considère aussi les fichiers et bibliothèques inclus ainsi que les définitions des fonctions internes à Clang avant le fichier source concerné, comme montré dans l'exemple en Listing 6.1. Ne voulant pas modifier des fichiers non concernés, nous avons fait le choix d'ignorer cette partie de l'AST. Notre outil s'applique uniquement sur le fichier source donné en paramètre.

6.3 Instrumentation pour CADNA

L'instrumentation de code pour CADNA consiste principalement à modifier les types flottants du code source en types stochastiques comme vu dans la Section 2.2.3. Un exemple est donné dans le code du Listing 6.4, version instrumentée du code source donné dans le Listing 6.3. Cet exemple montre que nous devons aussi inclure la bibliothèque `cadna.h`, ce qui est fait grâce à un `Replacement` portant au début du fichier source. De même, nous avons aussi rajouté les appels aux fonctions `cadna_init()` et `cadna_end()`, respectivement au début et fin de la fonction `main`. Un matcher sur la fonction `main` combiné aux fonctions du `Lexer` nous permet d'ajouter ces deux directives.

2. jeton en anglais, terme utilisé pour indiquer une unité lors de l'analyse lexicale, transformant une chaîne de caractères en une liste de symboles (tokens)

```
#include <stdio.h>

using namespace std;

int main() {
    double x = 42;
    double y = 193;
    double res;

    res = 4*x*x+9*y*y*y;

    printf("res: %.2e\n", res);

    return 0;
}
```

Listing 6.3 Code source

```
#include <cadna.h>
#include <stdio.h>

using namespace std;

int main() {
    cadna_init(0);
    double_st x = 42;
    double_st y = 193;
    double_st res;

    res = 4*x*x+9*y*y*y;

    printf("res: %s\n", strp(res)
        );

    cadna_end();
    return 0;
}
```

Listing 6.4 Code instrumenté pour CADNA

Le Listing 6.4 montre aussi l'utilisation de la fonction `strp()` de la bibliothèque `cadna.h` renvoyant une chaîne de caractères et permettant d'afficher le résultat d'un nombre stochastique (i.e. avec le bon nombre de chiffres significatifs exacts). Notre outil doit donc modifier le *specifier*[3] à l'intérieur de la chaîne de caractères (`%.3e` en `%s` par exemple), ainsi que l'argument associé. Pour cela, un matcher pour chacune des fonctions `printf`, `asprintf`, `fprintf`, `snprintf` et `sprintf` a donc été créé. Afin de trouver la chaîne de caractères dans les arguments de la fonction, nous parcourons ces derniers en explorant simplement les nœuds enfants dans l'AST grâce au pointeur `child_begin()`. Lorsque nous trouvons la chaîne de caractères, nous parcourons celle-ci afin de trouver les différents specifiers et récupérons leurs positions par rapport aux autres specifiers ce qui permet de modifier l'argument associé s'il s'agit d'un specifier flottant (f, F, e, E, g ou G). Nous créons de plus en parallèle la nouvelle chaîne de caractères avec le specifier remplacé, puis créons le `Replacement` associé lorsque nous atteignons la fin de la chaîne de caractères.
De même, l'outil permet aussi de modifier les types stochastiques présents lors de l'appel à la fonction `sizeof()` ainsi que lors de casts explicites.

Enfin, l'outil d'instrumentation pour CADNA a aussi été adapté pour instrumenter les codes MPI. Rappelons que CADNA a été adapté à MPI grâce à la bibliothèque `cadna_mpi.h` présentée dans [?]. MPI (*Message Passing Interface*) est un standard de parallélisation de codes sur des machines multiprocesseurs, ou sur des clusters à mémoire distribuée, largement utilisé dans les codes HPC. Adapter l'outil pour la version MPI de CADNA est ainsi primordial. En plus de remplacer les types flottants par des types stochastiques, les types MPI `MPI_FLOAT` et `MPI_DOUBLE` sont aussi remplacés par les types MPI stochastiques correspondants, `MPI_FLOAT_ST` et `MPI_DOUBLE_ST`, définis dans la bibliothèque CADNA MPI. Pour réaliser cela, nous créons des matchers sur les fonctions `MPI_Send`, `MPI_ISend`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Bcast` et `MPI_Scatter`, qui prennent en paramètre ces types.

3. nom donné aux lettres suivant un % dans une chaîne de caractères

Celles-ci sont les fonctions principales de MPI qui permettent respectivement d'envoyer un message, d'envoyer un message non bloquant, de recevoir un message, d'envoyer et recevoir un message simultanément, d'envoyer un message à tous les processus (Bcast = *broadcast*), et de découper un tableau dans tous les processus.
La contrainte dans cette situation est qu'il est difficile de matcher les types MPI directement, car ceux-ci n'apparaissent pas tels quels dans l'AST Clang, mais en tant que `MPI_Datatype`. Pour remplacer les types MPI, nous récupérons donc toute la ligne correspondante grâce au `Rewriter` Clang pouvant récupérer le texte d'origine sous forme de chaîne de caractères. Cela nous permet de créer une nouvelle chaîne de caractères avec le type MPI modifié, puis d'effectuer le `Replacement` adéquat.

Notre outil ajoute aussi les appels aux fonctions `cadna_mpi_init()` et `cadna_mpi_end()`, nécessaires au fonctionnement de CADNA dans du code MPI. Ces deux fonctions appellent `cadna_init()` et `cadna_end()`. La fonction `cadna_mpi_init()` prend en paramètre obligatoire le numéro du processus courant.

Un exemple de transformation de code MPI tiré de la bibliothèque CADNA est donné dans les Listings 6.5 et 6.6. Celui-ci fait la somme de trois résultats obtenus dans trois processus différents. Nous pouvons voir l'argument `p`, qui correspond au numéro du processus, dans la fonction `cadna_mpi_init()`.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {

  int p, np;
  MPI_Status status;

  /* MPI Initialization */
  MPI_Init( &argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &p);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  if( p ==0 ) {
     fprintf( stderr, "Computation with MPI without CADNA:\n");
     fprintf( stderr, "I am the Master\n");
  }
  if( np !=4 ) {
    if( p ==0 ) fprintf( stderr, "4 processes are necessary\n");
    MPI_Finalize();
    return 0;
  }

  double x=10864., y=18817.;

  if (p!=0) {
    double res;
    switch(p){
    case 1 : res=9.*x*x*x*x; break;
    case 2 : res=- y*y*y*y; break;
    case 3 : res= 2.*y*y; break;
```

```
    }
    printf("I am thread %d and I send %.15e\n",p,res);
    MPI_Send( &res, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  }
  else{
    double res=0.;
    double tmp;
    int i;
    for (i=1; i<=3; i++) {
      MPI_Recv( &tmp, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
      res+=tmp;
      printf("I have received %.15e\n",tmp);
    }
    printf("Final result: %.15e\n", res);
    printf("Exact result: 1.000000000000000e+00\n");
  }

  MPI_Finalize();
  return 0;
}
```

Listing 6.5 Code source MPI

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#include <cadna.h>
#include <cadna_mpi.h>

int main(int argc, char *argv[]) {

int p, np;
MPI_Status status;

 /* MPI Initialization */
MPI_Init( &argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &p);
MPI_Comm_size(MPI_COMM_WORLD, &np);

cadna_mpi_init(p, -1);

  if( p ==0 ){
     fprintf( stderr, "Computation with MPI using CADNA:\n");
     fprintf( stderr, "I am the Master\n");
  }
  if( np !=4 ) {
    if( p ==0 ) fprintf( stderr, "4 processes are necessary\n");
    MPI_Finalize();
    return 0;
  }

  double_st x=10864., y=18817.;

  if (p!=0) {
```

```
    double_st res;
    switch(p){
    case 1 : res=9.*x*x*x*x; break;
    case 2 : res=- y*y*y*y; break;
    case 3: res= 2.*y*y; break;
    }
   printf("I am thread %d and I send %s\n",p,strp(res));
   MPI_Send( &res, 1, MPI_DOUBLE_ST, 0, 0, MPI_COMM_WORLD);
  }
  else{
    double_st res=0.;
    double_st tmp;
    int i;
    for (i=1; i<=3; i++) {
      MPI_Recv( &tmp, 1, MPI_DOUBLE_ST, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &status);
      res+=tmp;
      printf("I have received %s\n",strp(tmp));
    }
  printf("-->Final result: %s\n", strp(res));
  printf("Exact result:  1.000000000000000e+00\n");
 }

  cadna_mpi_end();
  MPI_Finalize();
  return 0;
}
```

Listing 6.6 Code MPI instrumenté pour CADNA

Enfin, l'outil permet aussi de modifier les types stochastiques présents lors de l'appel à la fonction `sizeof()` ainsi que lors de casts explicites.

6.4 Instrumentation pour PROMISE

L'instrumentation pour PROMISE a donné naissance à deux fonctionnalités différentes : la première instrumente un code pour PROMISE, en remplaçant les variables de types flottants par des variables PROMISE, la seconde permet aussi de parser le code au sein de PROMISE afin d'enregistrer les différentes variables et d'appliquer les changements liés au Delta-Debug via l'instrumentation LLVM.

6.4.1 Remplacement des types

Le remplacement des types flottants dans le code peut se faire de la même manière que pour les types stochastiques dans le cas de CADNA. Pour chaque type rencontré, l'outil le remplace par un type PROMISE `__PR_I__` avec `I` un numéro que nous incrémentons au fur et à mesure. Un exemple est donné dans le Listing 6.8, version instrumentée pour PROMISE du code source donné dans le Listing 6.7. Cette instrumentation automatique est activée dans PROMISE grâce à l'option `-auto` lors de l'exécution de ce dernier.

```
#include <stdio.h>
#include <promise.h>

using namespace std;

int main() {
    double x = 42;
    double y = 193;
    double res;

    res = 4*x*x+9*y*y*y;

    printf("res: %.2e\n", res);

    PROMISE_CHECK_VAR(res);

    return 0;
}
```

Listing 6.7 Code source

```
#include <stdio.h>
#include <promise.h>

using namespace std;

int main() {
    __PR_0__ x = 42;
    __PR_1__ y = 193;
    __PR_2__ res;

    res = 4*x*x+9*y*y*y

    printf("res: %.2e\n", res);

    PROMISE_CHECK_VAR(res);

    return 0;
}
```

Listing 6.8 Code instrumenté pour PROMISE

De plus, l'utilisateur peut désormais choisir des parties du code à instrumenter en ajoutant les macros `PROMISE_START()` et `PROMISE_END()` autour de la (ou des) zone(s) à instrumenter, comme montré dans les Listings 6.9 et 6.10. Ces deux macros sont définies dans la bibliothèque `promise.h`. Cela permet à l'utilisateur de définir des zones du code à passer en précision mixte, et donc de conserver le code en dehors de ces macros dans une précision fixée. Pour réaliser cela, nous créons des matchers sur les appels aux macros `PROMISE_START()` et `PROMISE_END()` et nous récupérons les numéros des lignes respectives de ces appels. Ces macros peuvent apparaître au plus 100 fois. Par la suite, nous vérifions avant d'effectuer le callback que le match qui l'appelle se situe entre les deux macros en comparant les numéros de lignes.

```
#include <stdio.h>
#include <promise.h>

using namespace std;

int main() {
    PROMISE_START();
    double x = 42;
    double y = 193;
    PROMISE_END();

    double res;

    res = 4*x*x+9*y*y*y;

    printf("res: %.2e\n", res);

    PROMISE_CHECK_VAR(res);

    return 0;
}
```

Listing 6.9 Code source

```
#include <stdio.h>
#include <promise.h>

using namespace std;

int main() {
    PROMISE_START();
    __PR_0__ x = 42;
    __PR_1__ y = 193;
    PROMISE_END();

    double res;

    res = 4*x*x+9*y*y*y

    printf("res: %.2e\n", res);

    PROMISE_CHECK_VAR(res);

    return 0;
}
```

Listing 6.10 Code instrumenté pour PROMISE

Afin de garder un code complet cohérent, les cast explicites sont modifiés et remplacés par le type PROMISE qui convient. Par exemple, un cast dans le `return` d'une fonction aura forcément le même type PROMISE que le type de retour de la fonction, comme montré dans les Listings 6.11 et 6.12. Afin d'effectuer ce remplacement, un matcher sur les casts explicites de types flottants a été créé, ainsi que des fonctions permettant de remonter dans l'AST. Une première fonction nous permet de récupérer le `ReturnStmt`, et nous confirme que nous sommes dans une instruction `return` de fonction, une seconde nous permet de remonter jusqu'à la déclaration de la fonction. On cherche ensuite à récupérer non pas le type de la fonction dans le code, mais le type qu'elle va prendre après application des `Replacements`. Pour cela, on crée d'abord un `Replacement` sur la fonction que l'on a récupérée. On parcourt ensuite la liste des remplacements du `RefactoringTool` afin de trouver le remplacement ayant le même `Offset`, c'est-à-dire la même position dans le même fichier. Étant donné les remplacements que l'on effectue, l'`Offset` est bien unique pour nos `Replacements`. On peut ainsi récupérer le type que l'on a donné à la fonction.

```
double f(float x) {
    float y;
    y = 4*x*x;

    return (double)y;
}
```

Listing 6.11 Code source

```
__PR_0__ f(__PR_1__ x) {
    __PR_2__ y;
    y = 4*x*x;

    return (__PR_0__)y;
}
```

Listing 6.12 Code instrumenté pour PROMISE

La même stratégie est utilisée pour des déclarations de variables, comme présenté dans les Listings 6.13 et 6.14. Dans ce cas, la fonction qui remonte l'AST nous permet de récupérer la

déclaration de variable VarDecl. Enfin, la même chose est réalisée pour l'affectation d'une valeur à une variable, présentée dans les Listings 6.15 et 6.16. Dans ce cas, la fonction qui remonte l'AST nous permet de récupérer l'opération binaire "=", de type BinaryOperator, puis de redescendre dans ses nœuds enfants pour avoir l'élément à gauche du signe égal. Celui-ci nous permet enfin de récupérer la déclaration de la variable correspondante de type ValueDecl.

```
double b = (double)a;
```

Listing 6.13 Code source

```
__PR_0__ b = (__PR_0__)a;
```

Listing 6.14 Code instrumenté pour PROMISE

```
double b;
b = (double)a;
```

Listing 6.15 Code source

```
__PR_0__ b;
b = (__PR_0__)a;
```

Listing 6.16 Code instrumenté pour PROMISE

6.4.2 *Parsing* de codes

Notre outil d'auto-instrumentation a aussi été modifié en une API Python permettant de réaliser le *parsing* du code source nécessaire au sein de PROMISE.

En effet, à l'origine, un parsing du code source était effectué en Python afin de récupérer les variables PROMISE présentes dans le code source. Celles-ci étaient enregistrées dans un tableau, tandis que les types PROMISE étaient enregistrés dans un dictionnaire associant à chacun une précision. En parallèle de la récupération des variables, le code source était lui-même récupéré, découpé autour des types PROMISE des déclarations de variables et stocké dans plusieurs chaînes de caractères. Les types PROMISE étaient ainsi retirés. Cela permettait de recréer le code source avec les types désirés pour chaque variable PROMISE à chaque nouvelle configuration établie par l'algorithme du Delta-Debug.

Ce parsing, réalisé en Python, peut s'avérer instable et incomplet, et deviendra caduc. Une première amélioration a par exemple été de mieux prendre en compte les déclarations de tableaux avec affectation directe comme ci-dessous.

```
double tab[4] = {1.41, 2.71, 3.14, 42.0};
```

Ces déclarations étaient mal gérées si le tableau était déclaré avec un type PROMISE avec chaque valeur castée en type PROMISE, comme dans l'exemple ci-dessous. Or, cela était nécessaire pour obtenir des tableaux utilisant la précision half émulée.

```
__PR_0__ tab[4] = {(__PR_0__)1.41, (__PR_0__)2.71, (__PR_0__)3.14, (
    __PR_0__)42.0};
```

Cela montre l'importance de rendre cet outil plus robuste, ce qui est réalisé en utilisant les fonctionnalités de Clang via l'API créée. En effet, l'utilisation de LLVM/Clang, un outil très répandu, qui est maintenu et mis à jour, permet un parsing plus précis et global. Elle doit aussi apporter une solution plus efficace, nous évitant de lire le fichier source ligne par ligne et de le stocker dans des chaînes de caractères, comme effectué dans la version d'origine du parser.

Cette API reprend les principes de notre outil et utilise la représentation de l'AST Clang, mais permet d'en récupérer des informations et d'appliquer des `Replacements` depuis du code Python. Elle permet plus précisément de créer un objet appelé `instrumentizer` prenant en entrée un fichier source qui sera ouvert par Clang pour travailler dessus. Un `instrumentizer` se compose des matchers présentés précédemment, ce qui offre la possibilité de matcher les différents types PROMISE présents dans le code. Au lieu de créer directement des `Replacement` dans le callback, nous récupérons, selon ce qui est matché, les informations nécessaires au bon déroulement de PROMISE, et à la création ultérieure de `Replacement`. Pour les variables notamment, nous récupérons celles-ci dans un tableau que nous passons à PROMISE comme dans le parsing réalisé auparavant. Nous récupérons aussi leurs types PROMISE, ce qui permet de créer le dictionnaire associant une précision à chaque type PROMISE, et sert aussi à l'algorithme du Delta-Debug. De plus, les informations de position de chaque variable sont aussi stockées. À chaque nouvelle configuration examinée par l'algorithme du Delta-Debug, ces informations servent à créer les `Replacement` nécessaires au sein du code source, permettant ainsi de tester la configuration correspondante.

Outre le gain en robustesse, l'utilisation de Clang pour le parsing de code avait comme objectif de rendre ce dernier plus performant. Tout d'abord, nous espérions pouvoir ouvrir une seule fois l'AST pour notre fichier source et, en travaillant sur celui-ci, ne pas passer par la re-création de fichier à chaque nouvelle configuration du Delta-Debug. Cependant, la modification du fichier par les `Replacement` Clang corrompt les informations de l'AST précédemment ouvert, et oblige donc à réouvrir un AST. Le choix a donc été fait de garder la création de fichier pour chaque nouvelle configuration. Par ailleurs, la récupération des différentes variables et de leurs informations via Clang est normalement plus rapide que le code Python. Cependant, en recréant un fichier à chaque fois, différent du fichier source, nous forçons notre objet `instrumentizer` à changer de fichier pour l'application des `Replacement`, et devons donc ensuite le réinitialiser, puis le réappliquer sur le fichier source pour récupérer les variables et leurs informations pour l'itération suivante. Cela semble dupliquer les appels aux matchers, ce qui fait que le nombre de matchs double à chaque nouvelle application de l'`instrumentizer` sur notre fichier source. Cette erreur, dont l'origine n'a pas encore été identifiée, rallonge significativement le temps d'exécution. Pour le moment, nous prenons tout de même soin de ne pas récupérer les différentes variables en double, ne perturbant pas ainsi le bon fonctionnement de PROMISE.

6.5 Conclusion

L'outil d'instrumentation utilisant la bibliothèque LLVM LibTooling nous permet de changer les outils d'instrumentation et de parsing existant dans PROMISE. Une première version permet d'instrumenter du code pour CADNA sans s'appuyer sur un script Perl moins robuste. Celle-ci prend de plus en compte les codes MPI, pris en charge par CADNA. Une seconde version permet d'autoriser l'instrumentation automatique de code dans l'outil PROMISE, au lieu d'avoir à l'effectuer à la main. Enfin, cette version modifiée en API Python permet de

remplacer le parser présent au sein de PROMISE le rendant ainsi plus robuste et plus facile à maintenir.

7 Conclusion générale et perspectives

Le travail présenté dans cette thèse a porté sur l'auto-ajustement de la précision notamment via l'outil PROMISE. Comme expliqué dans le Chapitre 3, les outils d'auto-ajustement de la précision permettent d'obtenir un code en précision mixte, et de bénéficier ainsi des avantages de l'utilisation de formats flottants de tailles plus faibles : temps de lecture et d'écriture plus court, temps de calcul plus court, moins d'espace utilisé et moins d'énergie consommée. Cependant, comme vu au Chapitre 1, l'utilisation des nombres flottants s'accompagne d'erreurs d'arrondi inévitables dues à l'arithmétique flottante elle-même, manipulant un ensemble fini de nombres pour représenter un ensemble infini, les réels. À ces erreurs intrinsèques s'ajoutent les cas particuliers d'absorption et d'annulation catastrophique. Ces erreurs numériques s'avèrent d'autant plus importantes lors de l'utilisation de formats plus faibles. Les outils d'auto-tuning de précision fournissent alors toujours une solution qui respecte un critère sur la précision du résultat. La plupart de ces outils comparent le résultat d'un configuration à un résultat de référence, obtenu de manière analytique ou calculé avec la précision la plus élevée. PROMISE a quant à lui la particularité d'utiliser CADNA, un outil de validation numérique. Ce dernier permet d'obtenir un résultat de référence avec une estimation à 95% du nombre de chiffres significatifs exacts.

Le Chapitre 4 présente les résultats de l'application de PROMISE sur des réseaux de neurones. En effet, les outils d'auto-tuning de précision ont très peu été appliqués aux réseaux de neurones, malgré les besoins de compression de ces derniers (i.e. diminution de leur taille en mémoire). L'application de PROMISE sur 4 réseaux de neurones différents a permis de montrer que l'auto-tuning de précision parvient à diminuer les formats des poids utilisés dans chaque réseau. Deux approches ont pu être explorées : en considérant 1 type par neurone ou 1 type par couche. Tandis que la première permet de diminuer le plus la précision des différents paramètres, la seconde, en considérant moins de configurations possibles, permet d'obtenir un résultat plus rapidement. L'application sur différentes données en entrée a aussi permis de montrer que malgré une tendance globale identique selon l'entrée, ce paramètre doit être pris en compte selon le besoin de l'utilisateur.

Le Chapitre 5 réemploie les réseaux de neurones utilisés précédemment pour mettre en avant les gains en temps comme en mémoire obtenus grâce à l'application de l'auto-tuning de précision. En mémoire, les résultats s'approchaient des valeurs théoriques d'utilisation mémoire (16 bits pour chaque half, 32 bits pour chaque simple, 64 bits pour chaque double), avec un léger surcoût négligeable ($\sim$1%). Pour les gains en temps, des versions vectorisées et non vectorisées du code ont été étudiées, mettant en avant les avantages de la précision faible, en particulier grâce à la vectorisation. Dans les codes vectorisés, nous observons une accélération d'un facteur 2 par rapport à la précision double pour les parties vectorisées (produits matrice-vecteur) ainsi que pour la déclaration et l'initialisation des variables en précision simple. Ce résultat est égal à l'accélération théorique. Les parties non vectorisées

(appel à des fonctions de la bibliothèque `math.h` par exemple) viennent ternir le résultat du temps d'exécution du code complet, mais nous obtenons tout de même un gain en temps de facteur 1.86 pour la précision simple par rapport à la double, et un gain en temps de facteur 1.41 pour la pire configuration mixte obtenue par rapport à la précision double.
La performance de l'outil PROMISE a aussi été améliorée grâce à la parallélisation du Delta-Debug, permettant d'obtenir un résultat jusque 3.2x plus vite.

Enfin, le Chapitre 6 s'intéresse lui aussi à l'amélioration de PROMISE grâce à l'utilisation d'un outil d'instrumentation basé sur LLVM. Cet outil permet d'instrumenter des codes pour l'utilisation de CADNA, instrumentation réalisée jusqu'alors par un script Perl, non robuste et pouvant devenir obsolète. L'outil est aussi utilisé pour PROMISE, permettant d'instrumenter les codes d'entrée donnés à PROMISE de manière automatique. Il permet aussi à l'utilisateur de fixer les zones de son code dans lesquelles il souhaite intégrer de la précision mixte. De plus, cet outil a été transformé en une version API permettant de manipuler un objet d'instrumentation appelé `instrumentizer` en Python. Cette API est utilisée au sein de PROMISE pour l'analyse du code donné en entrée, permettant de récupérer les différentes variables sur lesquelles PROMISE doit travailler, ainsi que de créer les fichiers nécessaires au test de chaque nouvelle configuration. Ce *parsing*, ainsi que la création du code pour chaque nouvelle configuration, étaient réalisés auparavant par un script Python. L'utilisation de LLVM permet de rendre cette partie plus robuste et durable.

Le travail réalisé dans le Chapitre 6 ainsi que la parallélisation du Delta-Debug dans la Section 5.3 mettent en avant des améliorations de l'outil PROMISE rendant celui-ci plus efficace et plus robuste. Dans la continuité, différentes améliorations sont envisageables.

Premièrement, une amélioration récente dans PROMISE a été la prise en charge du format bfloat16, très répandu dans les réseaux de neurones. Pour cela, une option lors du lancement du logiciel a été ajoutée. Les tests ont été réalisés avec le récent format `std::bfloat16_t` du C++23. La prise en charge d'autres formats, ou même de formats personnalisés (qui ne sont pas définis par une norme), permettrait aussi à PROMISE de pouvoir être appliqué à des situations plus spécifiques qu'actuellement. L'application aux précisions arbitraires, notamment utilisées sur les FPGA (*Field Programmable Gate Arrays*), pourra aussi s'avérer très intéressante. Par ailleurs, il existe déjà SAM [**?**], une version de CADNA pour les précisions arbitraires.

Secondement, l'extension de PROMISE aux codes GPU permettrait d'étendre encore plus le champ d'action de l'outil. En effet, ces codes sont aussi largement utilisés dans l'industrie, car, étant massivement parallèles, ils permettent d'obtenir des performances bien supérieures à leurs équivalents sur CPU.

D'autres axes de développement moins spécifiques à PROMISE peuvent aussi s'avérer intéressants.

Tout d'abord, alors que nous avons dans notre cas utilisé LLVM pour l'instrumentation, celui-ci comporte aussi de nombreuses bibliothèques et de nombreux outils intégrés à sa chaîne de compilation. Il s'agit de plus d'un *framework* très utilisé et très répandu, répondant à de nombreux besoins. En particulier, LLVM intègre des *sanitizers* (littéralement "assainisseurs") qui permettent de trouver automatiquement des bugs au sein des programmes passés

dans LLVM, en utilisant notamment la représentation intermédiaire. Les différents sanitizers s'intéressent à différentes sources de bugs comme la fiabilité des *threads* (tsan), la mémoire (asan, msan, lsan) ou les comportements indéfinis (ubsan). NSan [**?**] est un sanitizer intégré à LLVM, développé par Clément Courbet, qui se charge de trouver et de corriger des erreurs liées à l'utilisation de l'arithmétique flottante. Pour cela, NSan utilise ce qu'on appelle une *mémoire shadow*, créée en parallèle de l'exécution, contenant les mêmes variables que le programme, et effectuant les mêmes opérations, mais dans une précision supérieure (le double de bits de précision). Si une différence est observée entre la précision plus faible et la précision supérieure, alors un bug est détecté. L'utilisation de la mémoire shadow permet aux tests de se faire de manière simultanée à l'exécution et ainsi d'obtenir une analyse jusqu'à 4x plus rapide par rapport aux outils existants. L'intégration dans LLVM représente de plus un énorme avantage : l'inclusion de ce débuggage dans des routines de tests. De ce fait, l'idée d'implémenter une version de l'Arithmétique Stochastique Discrète en utilisant une mémoire shadow a été étudiée. INSanE [1] (*Interface for Numerical Stability Sanitizer Extension*) développée par Mathys Jam et al. est une interface permettant d'implémenter des codes C++ d'analyse spécifique, appelés *runtime*, tout en utilisant les avantages de NSan : l'instrumentation et la manipulation de la mémoire shadow. L'analyse ainsi effectuée peut s'avérer plus précise, détecter plus d'instabilités que NSan, mais peut aussi donner l'occasion de valider un code, là où NSan se contente de détecter les instabilités. Cette interface, qui possède elle-même des axes d'améliorations, pourrait être utilisée pour l'implémentation de l'Arithmétique Stochastique Discrète avec utilisation d'une mémoire shadow, et intégrée à LLVM, permettant une utilisation simplifiée et accessible.

Sur un autre point, l'utilisation de différentes techniques ayant démontré leur intérêt pour l'auto-tuning de précision peut être une piste intéressante dans le développement de PROMISE. En particulier, la différentiation automatique peut par exemple être utilisée. Cette méthode permet d'obtenir pour chaque variable la dérivée de la fonction décrivant son processus de calcul, et donc son conditionnement. Connaître le conditionnement de chaque variable permet, par extension, d'estimer pour chaque variable l'impact sur l'erreur finale de l'erreur introduite par le passage à une précision plus faible. C'est ce qui est utilisé dans ADAPT [**?**], où les auteurs proposent un algorithme qui abaisse en premier la précision des variables dont le passage à une précision plus faible impacte le moins l'erreur finale. L'algorithme s'arrête lorsqu'il n'est plus possible de diminuer la précision de la variable suivante sans dépasser un seuil sur l'erreur finale.
PROMISE pourrait tirer bénéfice de la différentiation automatique en s'inspirant de ce qui est fait dans ADAPT. L'algorithme du Delta-Debug pourrait être utilisé et appliqué à la liste des variables ordonnées selon l'impact de la diminution de leur précision sur le résultat. Cela permet de rajouter un cas d'élimination de configuration : pour un découpage de la liste en n parties dans le Delta-Debug, si l'ajout du bloc i ne satisfait pas la condition sur la précision, alors il n'est pas nécessaire de tester l'ajout du bloc $i + 1$, constitué de variables dont l'abaissement de la précision aurait plus d'impact sur l'erreur. Au lieu de ça, il faut tester l'ajout du bloc i lui-même découpé. Cela permettrait d'améliorer les performances de PROMISE.
De plus, la différentiation automatique pourrait aussi être utilisée afin d'appliquer l'auto-

1. https://www.interflop.fr/documents/061021_Jam.pdf

tuning sur différentes valeurs d'entrées. En effet, en la combinant à des méthodes de validation numérique, il serait possible de considérer un ensemble d'entrées : un intervalle pour l'arithmétique par intervalles ou plusieurs données perturbées pour l'arithmétique stochastique. L'application de la différentiation automatique à l'instar de ce qui est fait dans APAPT permettrait d'obtenir un code en précision mixte non pas pour une donnée particulière, mais pour un ensemble de données.

Enfin, l'explosion récente des performances des intelligences artificielles, notamment génératives, c'est-à-dire capable de générer du texte, des images, des vidéos ou d'autres médias en réponse à des requêtes, vient offrir de nouvelles possibilités de recherche tant ces IA semblent pouvoir s'appliquer à tous les domaines. Une intelligence artificielle capable de détecter des instabilités dans un code est alors envisageable. De même pour l'auto-tuning de précision, une IA s'entraînant à reconnaître, selon les calculs effectués dans un code, les précisions adéquates pour chaque variable, ou du moins l'impact sur l'erreur finale de chaque variable, se présente comme une possibilité. Quelle que soit sa forme finale, l'utilisation de l'IA au service de la validation numérique ou de l'auto-tuning de précision constitue un axe de recherche prometteur.