Towards Reproducible, Accurately Rounded and Efficient BLAS

Chemseddine Chohra

THÈSE

Pour obtenir le grade de

Docteur

Délivré par

UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale ED305
Et de l'unité de recherche DALI-LIRMM

Spécialité : Informatique

Présentée par Chemseddine Chohra

Towards Reproducible, Accurately Rounded and Efficient BLAS

Soutenance prévue le 10 mars 2017 devant le jury composé de

- **Fabienne Jézéquel** — MCF HDR, Université Panthéon-Assas — Rapporteur
- **Marc Baboulin** — Professeur, Université Paris Saclay — Rapporteur
- **Alfredo Buttari** — Chargé de recherche CNRS, Toulouse — Examinateur
- **Jean-Guillaume Dumas** — Professeur, Université Grenoble-Alpes — Examinateur
- **Marc Daumas** — Professeur, Université de Perpignan — Examinateur
- **David Defour** — MCF HDR, Université de Perpignan — Examinateur
- **Philippe Langlois** — Professeur, Université de Perpignan — Directeur
- **David Parello** — MCF, Université de Perpignan — Co-Directeur

To family
And friends

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my director Prof. Philippe Langlois and my co-director Dr. David Parello for their continuous support, motivation and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis.

I would also like to thank all other thesis jury members: Dr. Alfredo Buttari, Dr. David Defour, Dr. Fabienne Jézéquel, Prof. Jean-Guillaume Dumas, Prof. Marc Baboulin and Prof. Marc Daumas. Their participation to the jury of this thesis is highly appreciated.

My sincere thanks also go to my colleagues in team project team DALI: Hugues de Lassus Saint-Geniès, Bernard Goossens, Sylvia Munoz, Christophe Nègre, Rafife Nheili, Cathy Porada, Guillaume Revy, Jean-Marc Robert, Laurent Thévenoux, Amine Najahi, Manuel Marin, Nasrine Damouche, Djallal Rahmoune, Guillaume Cano, Loic Cellier and Abdelkarim Chaouch Gacem.

Big thanks also to my friends Rawa, Said, Lotfi, Anis and Hadjer.

Last but not the least, I would like to thank my family: my parents Nabil and Nora, my brother Salim and my wife and lover Afef for supporting me spiritually throughout writing this thesis and my my life in general.

And finally, I would like to thank everyone who directly or indirectly had an influence on the completion of this work.

RÉSUMÉ EN FRANÇAIS

Extrait du guide à l'usage du candidat au Doctorat en vue de la soutenance de thèse (version du 25/06/2015) du collège doctoral de l'Université de Perpignan via Domitia: *La langue des thèses et mémoires dans les établissements publics et privés d'enseignement est le français, sauf exceptions justifiées par les nécessités de l'enseignement des langues et cultures régionales ou étrangères ou lorsque les enseignants sont des professeurs associés ou invités étrangers. La maîtrise de la langue française fait partie des objectifs fondamentaux de l'enseignement. (· · ·) Lorsque cette langue n'est pas le français, la rédaction est complétée par un résumé substantiel en langue française. Pour l'ED 305 une synthèse d'une vingtaine de pages en français est intégrée au manuscrit, mettant en évidence les apports principaux du travail de thèse.*

0.1 INTRODUCTION

La puissance de calcul des ordinateurs pour le HPC (High Performance Computing) augmente exponentiellement depuis quelques dizaines d'années. Suivant cette tendance, l'ordinateur exascale, c'est-à-dire 10^{18} opération flottante par seconde sera développé avant 2020. La fréquence des processeurs ne peut plus être augmentée d'une façon efficace. Pour cela, la puissance de calcul est améliorée en développant des nouveaux jeux d'instruction, ou en utilisant plusieurs coeurs (ou processeurs) en parallèle. L'ordonnancement dynamique des taches et le comportement non déterministe de ces environnements parallèles changent l'ordre des opérations d'une exécution à une autre. À cause des erreurs d'arrondi, l'addition flottante est non-associative. Le changement d'ordre des opérations dans les environnements conduit les calculs flottants à fournir des résultats différents d'une exécution à une autre même, pour les mêmes entrées. Nous nous intéressons dans cette thèse à ce problème de non-reproductibilité numérique dans les BLAS (Basic Linear Algebra Subprograms).

Chaque opération flottante génère une erreur d'arrondi qui éloigne le résultat calculé du résultat exact. Le problème de reproductibilité se pose lorsque les mêmes calculs génèrent des erreurs différentes. Nous pouvons définir le problème de reproductibilité sur plusieurs niveaux.

1. La reproductibilité des résultats numériques quand les calculs sont effectués en parallèle sur la même machine avec un nombre de threads[1] fixé.

2. La reproductibilité lorsque les calculs sont effectués sur la même machine en faisant varier le nombre de threads.

3. La reproductibilité des résultats quand les calculs sont effectués sur des machines différentes.

Dans les 3 niveaux cités, l'ordre ou la sémantique des opérations flottantes peuvent changer d'une exécution à une autre. Ce qui va engendrer la génération d'erreurs différentes. Pour cela nous pourrons obtenir des résultats différents pour les mêmes entrées du programme.

La reproductibilité des résultats est très importante pour comprendre le comportement d'un programme ou pour déboguer des erreurs ou même pour faire la différence entre les bugs et les erreurs numériques. Dans certains cas, la reproductibilité des résultats est une condition nécessaire pour accepter qu'un programme est correct. Pour cela elle est listée parmi les 10 premiers défis de l'ordinateur exascale [38].
Des échecs liés au problème de reproductibilité sont dénoncés dans plusieurs applications de simulation scientifique comme

- dynamique des fluides [51];
- simulation hydrodynamique [36];
- simulation d'énergie [66];
- modélisation climatique [23]; et en
- simulation dynamique moléculaire [61].

1 thread est utilisé pour nommer une séquence de calcul susceptible de s'exécuter en parallèle

Plusieurs solutions ont été proposées pour avoir des résultats numériques reproductibles. Les bibliothèques de programmation parallèle comme OpenMP [63] et TBB [64] donnent la possibilité d'ordonnancer les taches statiquement et exécuter les réductions dans un ordre déterministe pour effectuer les opérations flottantes dans le même ordre et garantir des résultats reproductibles. Par contre cette solution reste exposée au changement de jeu d'instructions si les calculs sont effectués sur deux processeurs différents. La bibliothèque MKL de Intel [31] va un peu plus loin et donne la possibilité d'utiliser le même jeu d'instructions sur deux processeurs différents avec la fonctionnalité CNR [33] disponible sur la version 11.0 (ou plus récentes) de la bibliothèque. L'inconvénient de cette solution est l'obligation d'utiliser le même nombre de threads pour que les résultats soient reproductibles. Cela va limiter les performances si un code basé sur MKL est porté d'un processeur à un autre qui à plus de coeurs ou un jeu d'instructions plus récent. D'autres travaux consistent à améliorer la précision du résultat pour avoir des résultats reproductibles [36, 66]. Cependant, cette technique n'est pas une solution universelle et reste exposée au problème de reproductibilité si les données sont mal conditionnées. Demmel et Nguyen proposent de préarrondir les entrées pour commettre toujours la même erreur et fournir le même résultat indépendamment de l'ordre des opérations [12]. Ce qui garantit la reproductibilité des résultats mais pas toujours la précision, en particulier pour, une fois de plus, les problèmes mal conditionnés. des travaux récents [9, 41] proposent des solutions pour calculer une solution reproductible et correctement arrondie pour résoudre à la fois les deux problèmes de reproductibilité et de précision.

L'objectif de cette thèse est de fournir une implémentation reproductible des BLAS (Basic Linear Algebra Subroutines). Les fonctions de BLAS sont classées en 3 niveaux.

1. Le premier niveau définit les fonctions d'une complexité $O(n)$ (opérations vecteur-scalaire ou vecteur-vecteur).

2. Le deuxième niveau définit les fonctions d'une complexité $O(n^2)$ (opérations matrice-vecteur).

3. Le troisième niveau définit les fonctions d'une complexité $O(n^3)$ (opérations matrice-matrice).

Nous proposons ici des solutions pour le premier et le deuxième niveau seulement. Nous nous intéressons à cette bibliothèque parce qu'elle est largement utilisée pour développer des applications scientifiques. Fournir des BLAS reproductibles permettrait de résoudre le problème de reproductibilité pour une grande partie des applications qui en souffrent. De plus, lorsque c'est possible, fournir une implémentation arrondie correctement résout le problème de reproductibilité indépendamment de la configuration matérielle (nombre de threads et architecture de processeur). Cet objectif est atteignable pour une grande partie des BLAS grâce aux algorithmes de sommation et transformations sans erreur. Nous étudions les algorithmes de sommation pour implémenter des BLAS correctement arrondies avec un minimum de surcoût en terme de temps d'exécution par rapport aux BLAS classiques. Nous prenons comme référence la bibliothèque Intel MKL qui fournit une implémentation des BLAS extrêmement bien optimisée pour les architectures Intel.

0.2 ARITHMÉTIQUE FLOTTANTE

Le standard IEEE-754 [26] normalise la représentation, les opérations et les modes d'arrondi pour les nombres flottants. La représentation des nombres flottants est basée sur la notation scientifique. Un nombre flottant x s'écrit:

$$x = (-1)^{signe} \times mantisse \times 2^{exposant-biais},$$

Pour les trois formats binaires ($binary32$, $binary64$ et $binary128$) introduits par le standard IEEE-754, les $signe$, $mantisse$ et $exposant$ sont définis de la façon suivante.

- **signe** prend la valeur 0 pour les nombres positifs et 1 pour les nombres négatifs.
- **mantisse** une chaine binaire:

$$mantisse = b_0.b_1b_2...b_{m-1},$$

où pour m la taille de la mantisse, soit:

- $m = 24$ pour binary32;
- $m = 53$ pour binary64; et
- $m = 113$ pour binary128.

Nous avons $b_0 = 1$ pour les nombres flottants normalisés, et $b_0 = 0$ pour les nombres flottants dénormalisés (ou sous-normaux). Pour cela, la partie fractionnelle seulement est stockée en mémoire et la valeur de b_0 est implicite.

- **exposant** définit le décalage de la virgule sur la mantisse d'un nombre flottant. Les tableaux 2.1 et 2.2 montrent les formats utilisés et comment calculer la valeur d'un nombre flottant à partir de sa présentation binaire.

0.2.1 *Modes d'arrondi*

Quatre modes d'arrondi sont introduites par le standard IEEE-754·

1. Arrondi au plus proche.
2. Arrondi dirigé vers zéro.
3. Arrondi dirigé vers le haut.
4. Arrondi dirigé vers le bas.

Dans ce qui suit, nous supposons que le mode d'arrondi par défaut est l'arrondi au plus proche.

0.2.2 *Arrondi correct et arrondi fidèle*

Pour un nombre réel x, l'arrondi correcte de x est un nombre flottant $\widehat{x}$ qui dépend du mode d'arrondi utilisée. Nous nous intéressons ici et dans toute cette thèse à l'arrondi au plus proche. Donc l'arrondi correcte $\widehat{x}$ est le nombre flottant le plus proche de x. Ainsi l'arrondi correct du x flottant est lui-même.

Si x n'est pas flottant, les deux nombres flottant qui entourent x sont tous les deux des arrondis dits fidèles de x. Par contre, l'arrondi fidèle de x est aussi lui même si ce dernier est un flottant.

0.2.3 *Opérations flottantes*

Le standard IEEE-754 exige que les opérations arithmétiques de base $(+, -, \times, /, \sqrt{})$ soient correctement arrondies. Même si les entrées d'une opération sont des nombres flottants, le résultat ne l'est pas nécessairement. Pour cela les opération flottantes $\oplus, \ominus, \otimes, \oslash$ remplacent respectivement les opérations mathématiques $+, -, \times, /$.

Pour deux nombres flottants a et b, le résultat $a \oplus b$ est l'arrondi correct de $a + b$. De même pour les autres opérations $\ominus, \otimes$ et $\oslash$.

0.2.4 *Erreurs d'arrondi*

Les résultats fournis par des opérations flottantes sont généralement des approximations des résultats exacts qui dépendent du mode d'arrondi. La différence entre le résultat exact et le résultat flottant est ce qu'on appelle *l'erreur d'arrondi*. Pour un résultat exact x, et $\mathrm{round}(x)$ étant l'arrondi correcte de x, nous avons:

$$|x - \mathrm{round}(x)| < u \cdot |x|,$$

ou u est l'unité d'arrondi définie telle que:

$$u = 2^{-m} \text{ en arrondi au plus proche, ou}$$
$$u = 2^{1-m} \text{ en arrondi dirigé.}$$

On rappelle que m est le nombre de bits dans la mantisse de $\mathrm{round}(x)$. La valeur $|x - \mathrm{round}(x)|$ est *l'erreur absolue* et cette erreur peut être exprimé relativement à x avec la formule:

$$\frac{|x - \mathrm{round}(x)|}{|x|} \leqslant u.$$

La valeur de $|x - \mathrm{round}(x)| / |x|$ est *l'erreur relative* entre résultat calculé et résultat exact.

0.2.5 *L'élimination*

L'élimination (cancellation) survient lors de la soustraction de deux nombres flottants qui sont très proches. Sterbenz a démontré [60] que pour $a/2 \leqslant b \leqslant 2a$ la soustraction flottante de a et b est exacte ($a \ominus b = a - b$). Donc l'élimination n'introduit pas de nouvelles erreurs au calcul mais elle amplifie les erreurs générées par les calculs précédents. En d'autres termes, l'élimination augmente l'erreur relative de façon beaucoup plus significative que l'erreur absolue.

0.2.6 *Transformations sans erreur*

Les transformations sans erreur sont des algorithmes qui nous permettent de calculer les erreurs commises en effectuant des opérations flottantes. En arrondi au plus proche, pour deux nombres flottants a et b, les deux algorithmes TwoSum et FastTwoSum (regardez les algorithmes 2.3 et 2.2) peuvent être utilisés pour calculer s et e tel que:

$$\begin{aligned} s &= a \oplus b, \\ a + b &= s + e. \end{aligned}$$

L'algorithme FastTwoSum requiert la condition $|a| \geqslant |b|$ pour garantir la transformation sans erreur. Toutefois, les deux entrées a et b peuvent être permutées pour vérifier que $|a| \geqslant |b|$.

Pour deux nombres flottants a et b, les algorithmes TwoProd et 2MultFMA (algorithmes 2.6 et 2.7) calculent p et e tel que:

$$\begin{aligned} p &= a \otimes b, \\ a \times b &= p + e. \end{aligned}$$

2MultFMA effectue moins d'opérations flottantes que TwoProd et il est par conséquent plus efficace. Mais il est basé sur l'instruction FMA (Fused Multiply and Add) qui n'est pas disponible sur

tous les processeurs.

0.3 SOMMATION ET BLAS

Nous présentons dans cette section nos algorithmes pour des sommations et des BLAS reproductibles. Les solutions proposées fournissent des résultats qui ne dépendent ni du nombre de threads, ni de l'architecture du processeur. La précision des résultats est aussi très importante. Pour cela, nous assurons l'arrondi au plus proche lorsque c'est possible. Autrement, nos algorithmes fournissent des résultats plus précis que les algorithmes classiques.

0.3.1 *Précision et reproductibilité de la sommation*

Soit p un vecteur de nombres flottants, et leurs sommes exacte et calculée $S = \sum_{i=1}^{n} p_i$ et $\widehat{S} = p_1 \oplus p_2 \oplus ... \oplus p_n$. Muller et al. rappellent dans [40] que l'erreur relative sur $\widehat{S}$ est majorée comme suit:

$$\frac{\left|S - \widehat{S}\right|}{|S|} \leqslant \gamma_{n-1} \cdot \frac{\sum_{i=1}^{n} |p_i|}{|\sum_{i=1}^{n} p_i|},$$

où

$$\gamma_n = \frac{nu}{1 - nu}.$$

Donc l'erreur relative dépend de n la taille du vecteur p, la précision machine u et le conditionnement de la somme défini par la formule

$$\mathrm{cond}\left(\sum_{i=1}^{n} p_i\right) = \frac{\sum_{i=1}^{n} |p_i|}{|\sum_{i=1}^{n} p_i|}.$$

0.3.2 *Somme correctement arrondie*

Pour la somme séquentielle, nous avons étudié les algorithmes déjà disponibles et observé que `iFastSum` [69] est plus efficace en terme de temps de calcul pour des petits vecteurs (< 5000), tandis que `OnlineExact` [70] doit être préféré pour des grands vecteurs. Les deux algorithmes calculent l'arrondi au plus proche

de la somme, mais de deux façons différentes.

Somme séquentielle

L'algorithme iFastSum repose sur la distillation (expliquée dans la section 3.4.1) pour effectuer des transformations sans erreur sur le vecteur d'entrée et utiliser répétitivement la somme des erreurs pour améliorer la précision du résultat. Le nombre d'itérations effectuées par iFastSum dépend du conditionnement du problème. Comme tous les algorithmes basés sur la distillation, iFastSum est plus efficace quand le vecteur d'entrée tient dans le cache du processeur. Autrement, le coût de recharger les données de la mémoire à chaque raffinement va significativement augmenter le temps d'exécution pour cet algorithme. Plus de détails sur cet algorithme sont présentés dans la section 3.4.1.

De l'autre côté OnlineExact est basé une approche différente. Il utilise aussi iFastSum pour calculer l'arrondi correct de la somme, mais après avoir réduit la taille du vecteur. Pour cela, OnlineExact accumule dans un même accumulateur les entrées qui ont le même exposant. Les accumulateurs sont simulés avec deux nombres flottants standards. OnlineExact utilise un accumulateur pour chaque valeur possible de l'exposant. Pour le format binary64, le vecteur d'entrée p est ainsi transformé sans erreur en un vecteur C de 4096 nombres flottants (2048 accumulateurs de 2 nombres flottants chacun). Un vecteur de cette taille (32kb) tiendrait au moins dans le cache L2 des processeurs actuels. Notez que l'algorithme OnlineExact ne fait qu'un seul passage sur le vecteur d'entrée pour le transformer sans erreur à un vecteur qui tient dans le cache, et sur lequel nous allons appliquer un algorithme de sommation itératif. Cette solution est très efficace pour les grands vecteurs, mais nous utilisons iFastSum directement pour des petits vecteur pour ne pas payer le coût de transformation sans réduire significativement la taille du vecteur. Nous décrivons l'algorithme OnlineExact avec quelques légères modifications pour des raisons d'optimisation dans la section 4.2.2.

Somme parallèle

Nous décrivons notre algorithme de sommation parallèle correctement arrondie en trois étapes comme le montre la figure 0.1.

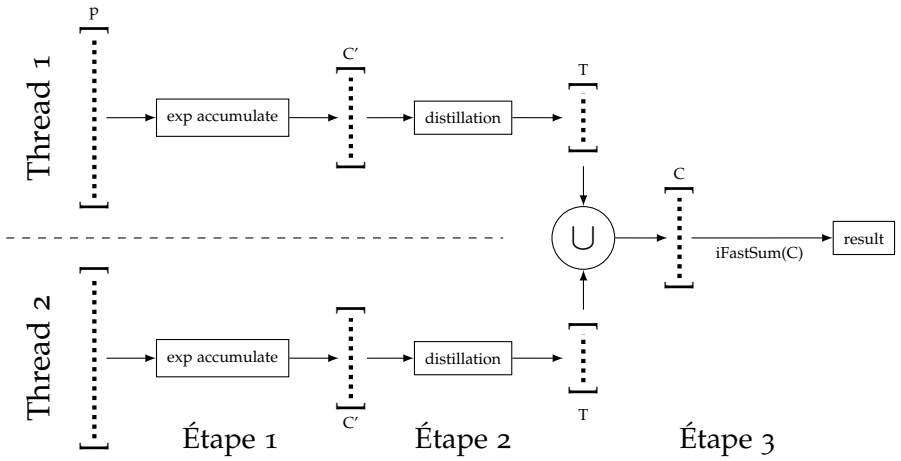


Figure 0.1: Algorithme parallèle pour la sommation correctement arrondie (le vecteur p est distribué sur les threads, ici nous montrons un exemple avec deux threads)

ÉTAPE 1 Dans cette étape, le vecteur d'entrée est divisé entre les threads. Chaque thread fait une transformation sans erreur sur sa partie locale du vecteur. Le but de cette première étape est de réduire la taille de vecteur en entrée. Cela est fait en utilisant la transformation de `OnlineExact` expliquée dans la section précédente. Le résultat de cette étape est un vecteur local C' de taille 4096.

ÉTAPE 2 Cette étape applique un algorithme de distillation sur le vecteur C' généré par l'étape précédente pour transformer les données locales en somme de nombres flottants qui ne se chevauchent pas. Si la taille du vecteur local est suffisamment petite, on peut sauter la première étape et faire directement la distillation pour transformer le vecteur en somme de flottants qui ne se chevauchent pas. La figure 0.1 montre que le résultat de cette étape est écrit dans le vecteur T. La taille de ce dernier dépend du range d'exposant de l'entrée. Dans le pire des cas pour le format `binary64`, cette taille est limitée à 39 nombre flottant: la range d'exposants supportés par le format (2048) divisés par la taille de la mantisse (53).

ÉTAPE 3 Dans cette dernière étape, nous effectuons une opération d'union pour rassembler tous les vecteurs locaux T dans un seul vecteur C. Jusque là, toutes les opérations effectuées sont des transformations sans erreur. Ce qui garantit que

$$\sum_{i=1}^{n} p_i = \sum_{j=1}^{m} C_j.$$

Donc il suffit d'utiliser iFastSum en séquentiel pour calculer l'arrondi correct de C.

0.3.3 *Premier niveau des BLAS*

Après avoir expliqué notre algorithme de sommation nous passons maintenant aux BLAS. Dans cette section, nous nous intéressons au premier niveau des BLAS. Des solutions reproductibles sont proposées pour le produit scalaire (dot), la norme euclidienne (nrm2) et la somme des valeurs absolues (asum).

Produit scalaire correctement arrondi

Le produit scalaire de deux vecteurs x et y de taille n est défini par la formule:

$$S = \sum_{i=1}^{n} x_i \cdot y_i.$$

Pour chaque i, nous utilisons TwoProd ou 2MultFMA pour calculer p_i et r_i tel que:

$$x_i \cdot y_i = p_i + r_i.$$

Cela implique

$$\sum_{i=1}^{n} x_i \cdot y_i = \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} r_i = \sum_{i=1}^{2n} q_i,$$

où $q_i = p_i$ et $q_{n+i} = r_i$. Donc le produit scalaire de deux vecteurs de taille n est transformé sans erreur en une somme d'un vecteur de taille $2n$. Toutes les multiplications $x_i \times y_i$ sont indépendantes et donc peuvent être effectuées en parallèle. L'algorithme de sommation présenté dans la section 0.3.2 est ensuite utilisé

pour calculer la somme du vecteur q.

Norme euclidienne fidèlement arrondie

La norme euclidienne d'un vecteur p est définie par la la formule

$$\left(\sum_{i=1}^{n} p_i^2 \right)^{1/2}.$$

La somme $\sum_{i=1}^{n} p_i^2$ est correctement arrondie en utilisant le produit scalaire présenté dans la section précédente pour multiplier le vecteur p par lui même. Ensuite, nous calculons la racine carrée de cette somme pour avoir une norme euclidienne fidèlement arrondie [21].

Malheureusement, notre algorithme n'assure pas un résultat correctement arrondi. Cependant, ce dernier est reproductible en plus d'être fidèlement arrondi parce qu'il dépend d'un produit scalaire reproductible (correctement arrondi) et une opération de racine carrée qui est aussi reproductible et correctement arrondie comme le garantit le standard IEEE-754.

La somme des valeurs absolues

Le même algorithme de sommation introduit dans la section 0.3.2 pourrait être aussi utilisé pour calculer la somme des valeurs absolues ($asum$) d'un vecteur p. Néanmoins, la somme des valeurs absolues est un problème qui est bien conditionné et ne nécessite pas un algorithme aussi compliqué et coûteux pour atteindre les objectifs de notre bibliothèque. Nous allons plutôt utiliser un algorithme de sommation compensée pour améliorer la précision du résultat, et cela est généralement suffisant pour avoir un résultat correctement arrondi. Nous utilisons pour cela l'algorithme $Sum2$ [44]. La version séquentielle de ce dernier est introduite dans [44], tandis que la version parallèle est présentée dans [68]. $Sum2$ effectue la somme d'un vecteur p en remplaçant l'opération de la somme $\oplus$ par l'algorithme $TwoSum$ qui calcul et écrit les erreurs générées dans un autre vecteur p' (le vecteur p peut être écrasé si nécessaire). Ensuite, la somme du vecteur

p' est utilisée pour raffiner le résultat.

Nous introduisons un petit changement à l'algorithme Sum2. Lorsque la somme de p' est effectuée, nous définissons une valeur maximale pour l'erreur générée en fonction des valeurs intermédiaires de cette somme. Cela nous permettra de savoir si le résultat compensé est correctement arrondi ou pas. Si ce résultat est correctement arrondi, il est retourné par notre asum. Sinon, il est jeté et notre algorithme de sommation présenté dans la section 0.3.2 est alors utilisé pour calculer un résultat correctement arrondi. Notre approche est présentée avec plus de détails dans la section 5.4.

0.3.4 *Deuxième niveau des BLAS*

Dans cette section, nous introduisons nos solutions pour assurer la reproductibilité du deuxième niveau des BLAS. Des solutions reproductibles sont proposées pour la multiplication matrice vecteur (gemv) et la résolution des systèmes triangulaires (trsv).

Multiplication matrice vecteur correctement arrondie

La multiplication matrice vecteur des BLAS 2 est définie par l'expression

$$y = \alpha A \cdot x + \beta y,$$

où A est une matrice de m lignes et n colonnes, x est un vecteur de taille n, y est un vecteur de taille m et α et β sont deux nombres flottants. Notre objectif est de garantir que tous les éléments de y sont correctement arrondis. Donc pour chaque i, il faut assurer que

$$y_i = \alpha a^{(i)} \cdot x + \beta y_i$$

est correctement arrondi, où y_i est le i^{me} élément de y et $a^{(i)}$ est la i^{me} ligne de A. Pour cela nous utilisons l'algorithme décrit

par les trois étapes suivantes.

ÉTAPE 1 Premièrement, le produit scalaire $a^{(i)} \cdot x$ est transformé sans erreur en somme de nombres flottants qui ne se chevauchent pas. Pour cela, nous utilisons d'abord TwoProd ou 2MultFMA pour transformer ce produit scalaire en une somme. Ensuite, nous appliquons le même algorithme décrit pour la somme parallèle correctement arrondie (étapes 1 et 2 de l'algorithme de somme parallèle dans la section 0.3.2). Le résultat de cette étape est écrit dans un vecteur T d'une taille maximale limité à 39 dans le pire des cas.

ÉTAPE 2 Tous les éléments du vecteur T générés par l'étape précédente sont multipliés par α, et y_i est multiplié par β en utilisant TwoProd ou 2MultFMA pour que la transformation soit sans erreur. Cela écrit tous les résultats de la multiplication et les erreurs associées dans un seul vecteur T2.

ÉTAPE 3 Jusqu'ici, $\alpha a^{(i)} \cdot x + \beta y_i$ a été transformé sans erreur en un vecteur T2. Donc finalement, il suffit d'utiliser iFastSum pour calculer la somme arrondie correctement de T2.

La version parallèle de cet algorithme calcule plusieurs y_i en utilisant plusieurs threads après avoir décomposé la matrice comme le montre la figure 0.2. Pour p threads, A est décomposé en p blocs tel que chaque bloc contient m/p lignes, y est aussi décomposé en p sous-vecteurs, tandis que x est atteignable par tous les threads.

Résoudre un système triangulaire

Pour T une matrice triangulaire carrée de taille n et b un vecteur de n nombre flottants, résoudre un système triangulaire consiste à trouver le vecteur x tel que $Ax = b$. Nous supposons ici que la matrice T est une matrice triangulaire inférieure inversible. Les éléments de x peuvent être calculés en utilisant la formule suivante

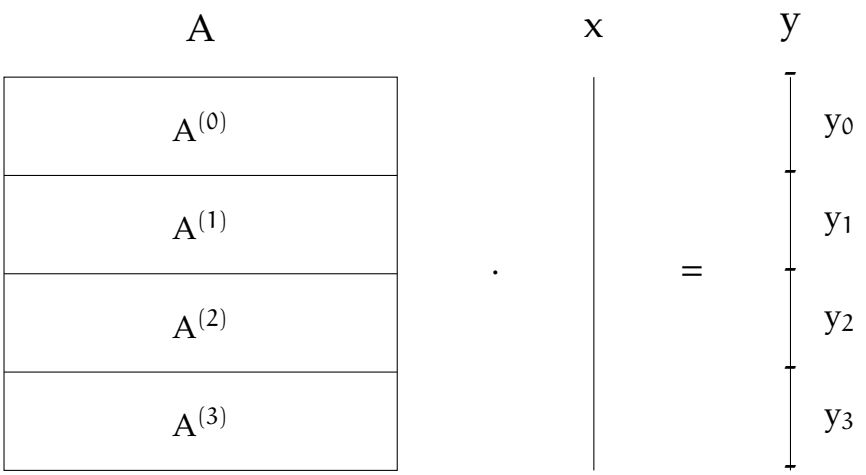


Figure 0.2: Algorithme parallèle pour la multiplication matrice-vecteur correctement arrondie (exemple avec 4 threads)

$$x_i = (b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j)/t_{ii} \text{ pour } i = 1...n,$$

Notez que avant de calculer x_i, tous les éléments x_j où $j < i$ doivent être calculés. Cela crée une dépendance entre chaque x_i et les éléments précédents. L'algorithme séquentiel 6.2 peut être utilisé pour calculer x. Une version parallèle de cet algorithme a été présentée dans [25]. Elle consiste à décomposer la matrice T en blocs comme le montre la figure 0.3. Chaque bloc *trsv* dépend des blocs *gemv* de la même ligne, et les blocs *gemv* dépendent du *trsv* de la même colonne.

Les blocs *trsv* sont exécutés en séquentiel tandis que les blocs *gemv* sont exécutés en parallèle. La source principale de la non-reproductibilité numérique pour *trsv* est la non-associativité de l'addition dans le produit scalaire $\sum_{j=1}^{i-1} t_{ij} \cdot x_j$. Les versions séquentielle et parallèle exécutent cette accumulation avec un ordre différent, ce qui va générer des erreurs différentes et donc fournir des résultats non reproductibles. Nous proposons deux approches différentes pour résoudre le problème dans ce cas. (1) la première solution consiste à calculer l'arrondi correct de l'expression $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ pour garantir la reproductibilité indépendamment de l'ordre des opérations. Pour cela nous avons utilisé l'algorithme HybridSum [69] (qui est similaire à OnlineExact

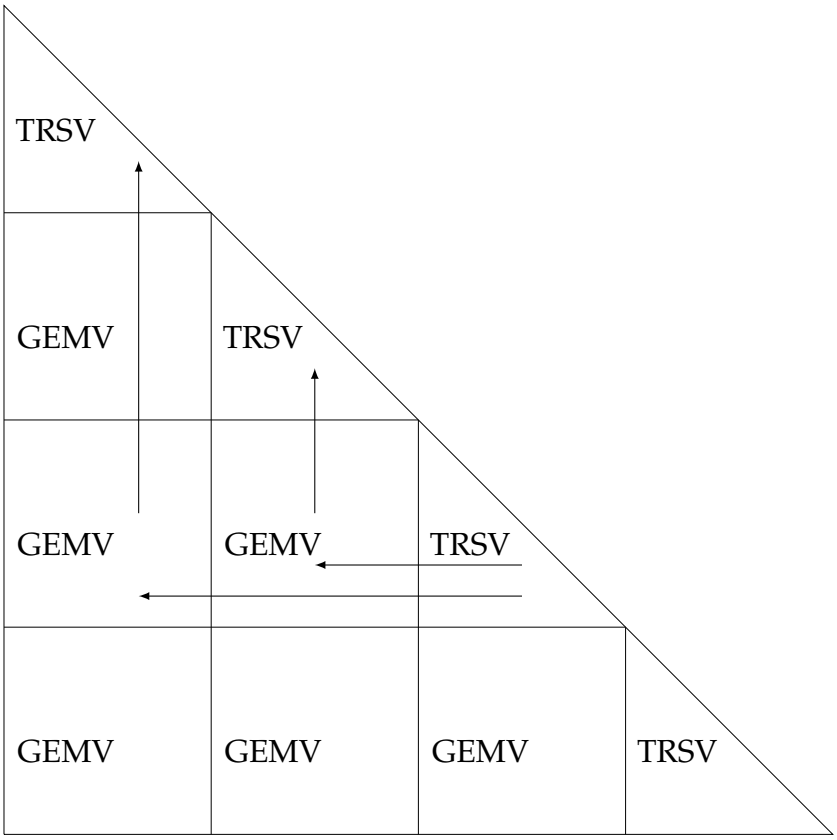


Figure 0.3: Résoudre un système triangulaire en parallèle et exemple de dépendances entre les blocs $trsv$ et $gemv$

mais demande moins d'espace mémoire, plus de détails dans la section 3.4.6). (2) la deuxième solution qui se base sur les nombres flottants indexés introduits dans [12] est présentés dans la section 3.5.3. Le résultat d'accumulation d'un vecteur de nombres flottants dans un nombre indexé est toujours reproductible indépendamment de l'ordre des opérations. Contrairement à la première approche, ici nous n'améliorons pas la précision de calcul, mais nous utilisons des raffinements itératifs après la résolution. Nous rappelons que les étapes du processus de raffinement itératif sont les suivantes.

1. Calculer une solution approchée $\widehat{x}$.
2. Calculer le résidu $r = b - T\widehat{x}$.
3. Calculer la solution du système (T, r) pour avoir un terme correctif d.
4. Mettre à jour $\widehat{x} = \widehat{x} + d$.

5. Répéter depuis l'étape 2 si la solution n'est pas suffisamment précise (dépend de la condition d'arrêt).

Plus de détails sur ces deux approches sont présentés dans la section 6.3.1.

0.4 RÉSULTATS EXPÉRIMENTAUX

Cette section compare nos solutions pour la sommation et pour des BLAS reproductibles aux autres solutions disponibles. Nous prenons les solutions fournies par la bibliothèque MKL comme référence de temps d'exécution, et une implémentation MPFR comme référence pour la précision. Nous comparons la qualité numérique des résultats dans un premier temps. Ensuite nous analysons le surcoût de nos solutions par rapport à MKL sur des architectures Intel.

0.4.1 *Précision*

Les résultats numériques fournis par nos solutions sont d'une part comparés aux algorithmes classiques, et d'un autre côté aux autres solutions reproductibles. Pour la somme, nous considérons les trois algorithmes ReprodSum [11], FastReprodSum [11] et OneReduction [13] (présentés en détail dans les sections 3.5.1, 3.5.2 et 3.5.4 respectivement). Pour le produit scalaire, la multiplication matrice-vecteur et la résolution des systèmes triangulaires, nous considérons seulement des solutions basées sur 1-Reduction et que nous appelons OneReductionDot, OneReductionGemv et OneReductionTrsv.

Nous testons les différentes solutions avec une taille de problème fixe et un nombre de conditionnement variable. La somme des valeurs absolues et la norme euclidienne ne sont pas incluses dans ce test parce que leur conditionnement ne peut pas être augmenté. Les algorithmes utilisés pour générer des instances avec des conditionnements différents sont présentés dans l'annexe B (section B.4.1 pour la sommation et le produit scalaire, section B.4.2 pour la multiplication matrice vecteur et section B.4.3 pour les

systèmes triangulaires).

PRÉCISION DE LA SOMME Pour la somme, la figure 4.6 compare la précision de notre algorithme de sommation Rsum à la somme classique et aux autres solutions reproductibles. Nous montrons sur l'axe des X le log en base 10 du conditionnement du problème et sur l'axe des Y le log en base 10 de l'erreur relative. La taille du vecteur est fixée à 10^5 pour toutes les entrées. L'erreur relative est calculée par rapport au résultat de la somme calculée avec la bibliothèque MPFR.

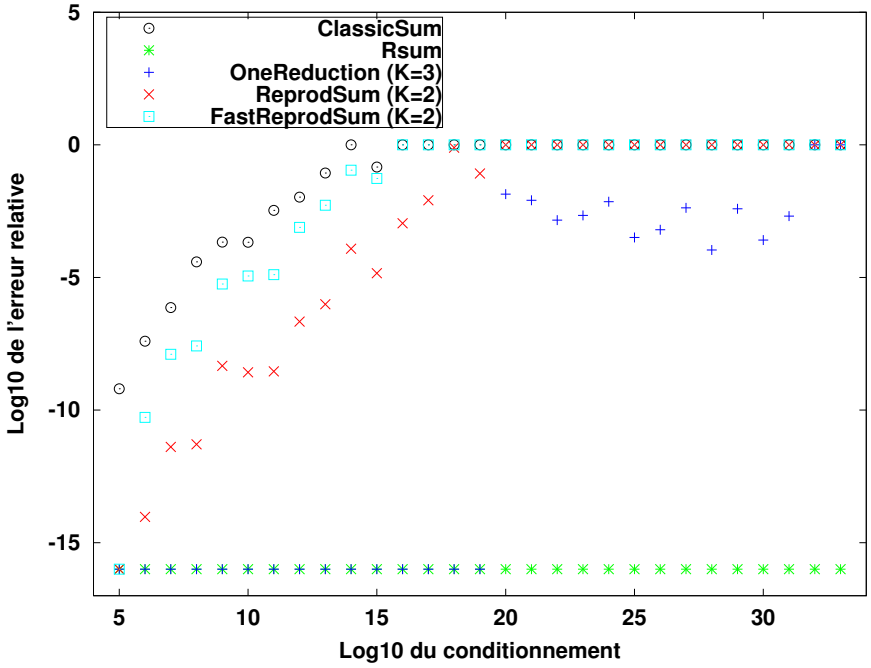


Figure 0.4: Précision de la somme (taille de vecteur $n = 10^5$)

Comme prévu, la précision de la somme classique dépend en grande partie du conditionnement du problème. À cette taille, pour des conditionnements supérieurs à 10^{15}, l'erreur relative est supérieures à 1. Ce qui signifie que le résultat ne contient aucun chiffre significatif. Notre algorithme Rsum donne toujours des résultats correctement arrondis, et l'erreur relative est toujours inférieure à $u \approx 10^{-16}$. Les autres solutions reproductibles sont plus précises que la somme classique mais elles dépendent toujours du conditionnement; ce qui rend leur utilisation difficile à justifier pour des problèmes mal conditionnés.

PRÉCISION DU PRODUIT SCALAIRE Nous avons limité nos tests ici à des conditionnements inférieurs à 10^{16} et nous avons aussi exclu les deux algorithmes ReprodSum et FastReprodSum parce que 1-Reduction est censé améliorer ces derniers.

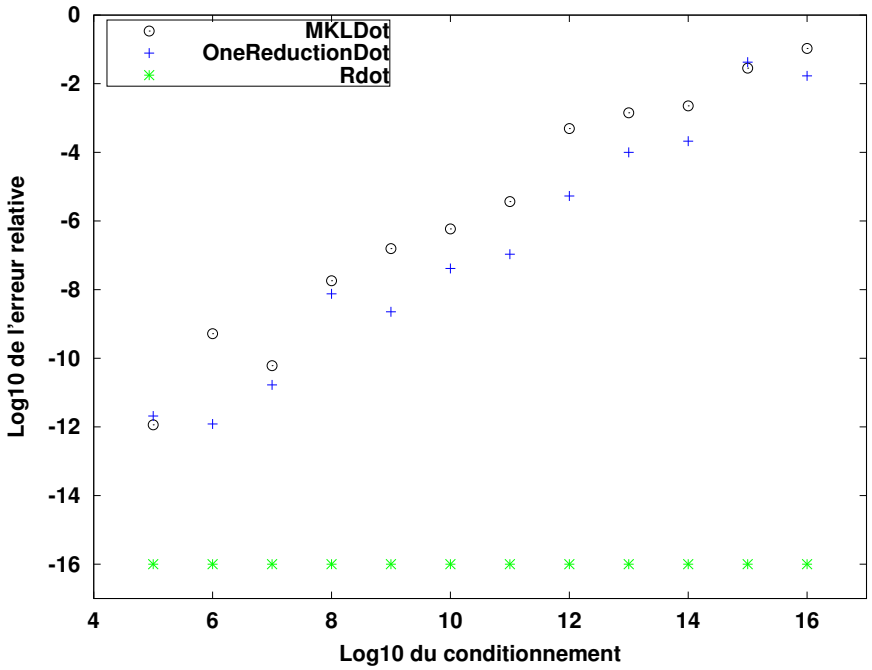


Figure 0.5: Précision du produit scalaire (taille de vecteur $n = 10^5$)

Nous avons aussi le même comportement que pour la somme. Notre algorithme Rdot assure que le résultat est correctement arrondi, tandis que la précision du produit scalaire de MKL dépend du conditionnement. OneReductionDot reste un peu plus précis que MKLDot.

PRÉCISION DU LA MULTIPLICATION MATRICE-VECTEUR La multiplication matrice-vecteur s'appuie sur le produit scalaire. Il est donc tout à fait normal qu'elle exhibe le même comportement. Notre implémentation donne toujours des résultats correctement arrondis, alors que les résultats de la multiplication matrice-vecteur de MKL et celle basée sur 1-Reduction dépendent du conditionnement.

PRÉCISION DE LA RÉSOLUTION DES SYSTÈMES TRIANGULAIRES Nous présentons ici deux mesures de précision pour chaque al-

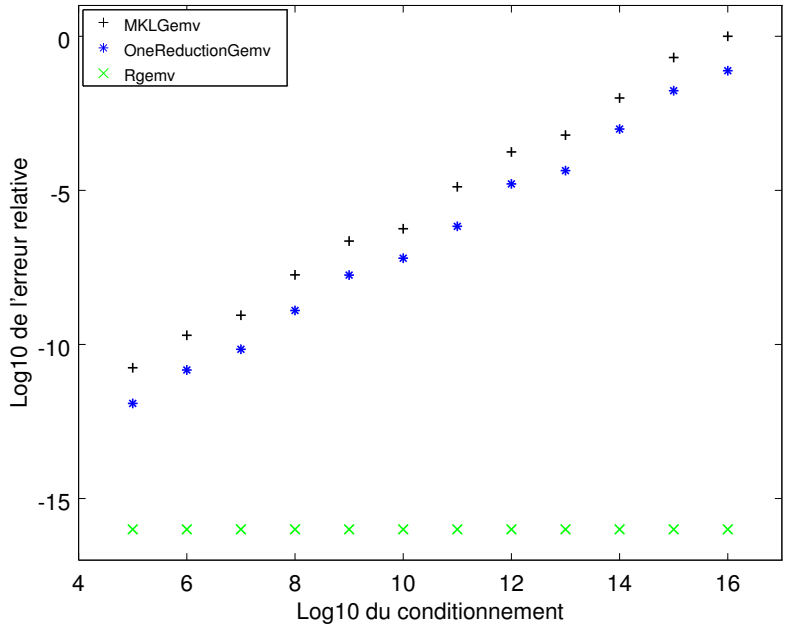


Figure 0.6: Précision de la multiplication matrice-vecteur

gorithme. Pour une solution calculée $\widehat{x}$, nous considérons l'erreur relative évaluée avec la relation

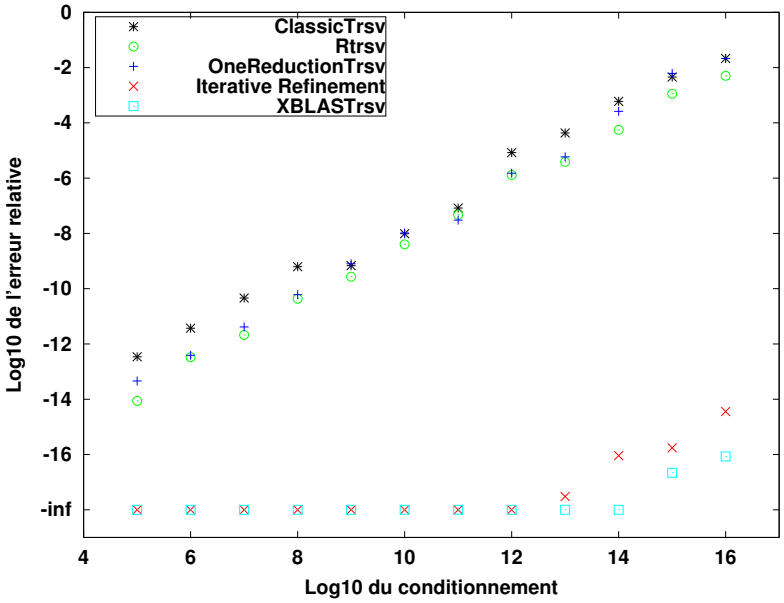
$$\| \widehat{x} - \tilde{x} \|_\infty / \| \tilde{x} \|_\infty,$$

tel que $\tilde{x}$ est la solution de référence calculée en utilisant la bibliothèque MPFR, et le résidu normalisé par b est calculé avec la formule

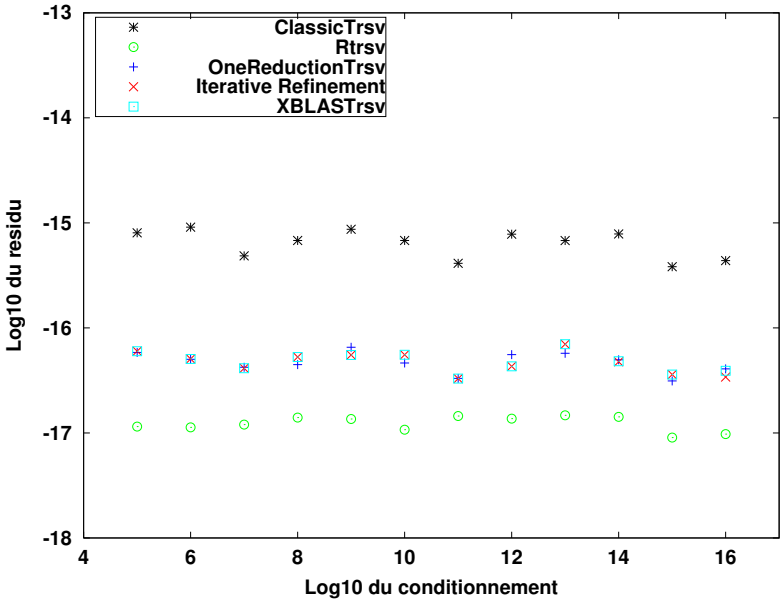
$$\| b - A\widehat{x} \|_\infty / \| b \|_\infty .$$

Nous avons aussi inclus la fonction `trsv` de la bibliothèque XBLAS [37] qui effectue toutes les opérations en double précision. Elle nous servira comme référence pour estimer la précision de nos solutions.

Nous avons expliqué que nous proposons deux algorithmes différents pour la résolution des systèmes triangulaires. Le premier algorithme `Rtrsv` calcule l'arrondi correct de l'expression $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$, alors que le second la calcule d'une façon reproductible et ensuite améliore la précision avec des raffinements itératifs.



(a) Erreur relative



(b) Résidu relativement à b

Figure 0.7: Précision de la résolution des systèmes triangulaires (taille du système = 1000)

La figure 0.7 montre les résultats de précision pour tous les algorithmes testés. Nous montrons sur l'axe des X dans les deux figures 0.7a et 0.7b le log (en base 10) du conditionnement. L'axe des Y, montre montre le log (en base 10) de l'erreur relative sur la figure 0.7a, et le log (en base 10) du résidu normalisé sur la

figure 0.7b.

Comme on l'a vu pour les fonctions précédentes, l'erreur relative de la solution fournie par MKLTrsv augmente linéairement avec le conditionnement. Le même comportement est observé pour OneReductionTrsv et Rtrsv. Cela implique que calculer l'arrondi correct de $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ n'améliore pas forcement la précision de la solution. Par contre, la solution qui réalise des raffinements itératifs fournit les même résultat que MPFR pour des conditionnement inférieurs à 10^{12}, et une erreur relative inférieure à u pour des conditionnement inférieurs à 10^{13}. XBLASTrsv montre aussi le même comportement, mais il est un peu plus précis pour des grands conditionnements.

La figure 0.7b montre que le résidu de la solution calculée ne dépend pas du conditionnement mais dépend principalement de l'algorithme utilisé. Les plus grands résidus sont donnés par MKLTrsv. Rtrsv assure des petites valeurs pour ce résidu alors que pourtant il calcule des solutions presque aussi précises que celles de MKLTrsv, Les trois autres algorithmes testés donnent des résidus d'une qualité intermédiaire entre ceux de Rtrsv et ceux de MKLTrsv.

Il est bien connu qu'un petit résidu n'implique pas forcément une meilleure précision [20]. Pour une solution calculée $\widehat{x}$, un petit résidu signifie que $A\widehat{x}$ prédit b précisément, tandis qu'avoir une meilleur précision exige que $\widehat{x}$ soit proche de la solution exacte x. Pour deux solutions calculées $\widehat{x_1}$ et $\widehat{x_2}$, il est possible que $\| \widehat{x_1} - x \|_\infty > \| \widehat{x_2} - x \|_\infty$ alors que $\| A\widehat{x_1} - b \|_\infty < \| A\widehat{x_2} - b \|_\infty$.

0.4.2 *Temps d'exécution*

Assurer la reproductibilité et améliorer la précision nécessite l'utilisation d'algorithmes qui sont compliqués, et qui introduisent plus d'opération. Ce qui à priori engendre un énorme surcoût en terme de complexité. Mais en pratique, un algorithme qui fait plus d'opérations n'est pas forcement plus lent. Nous avons testé nos solutions pour la sommation et les BLAS sur plusieurs

environnements différents. Nos tests ont été effectués sur une station de travail, un accélérateur Intel Xeon Phi et un super calculateur (seul le produit scalaire a été testé sur ce dernier). Pour plus d'informations sur l'environnement de test regardez le tableau B.1. Notre méthodologie est expliquée en détail dans l'annexe B. Le temps d'exécution a été mesuré en cycles en utilisant l'instruction assembleur RDTSC.

Les figures 0.8, 0.9, 0.11, 0.12, 0.13 et 0.14 montrent respectivement le surcoût de nos algorithmes de somme, produit scalaire, norme euclidienne, somme des valeurs absolues, multiplication matrice-vecteur et résolution des systèmes triangulaires chacun comparé par rapport à MKL sur les deux environnements A et B. La figure 0.10 montre la scalabilité du produit scalaire sur un super calculateur à mémoire distribuée.

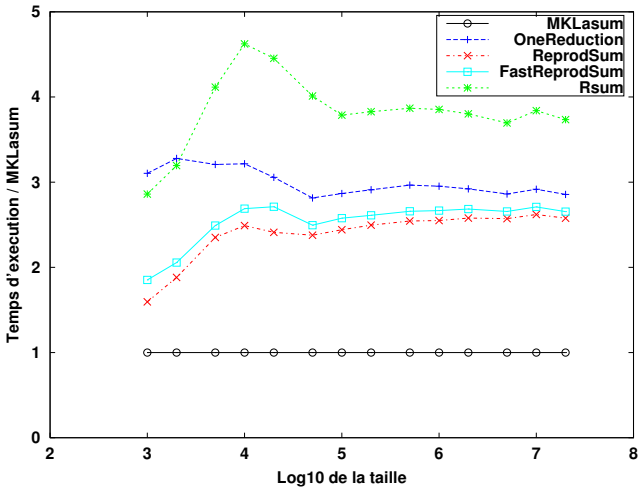
Le tableaux 0.1 récapitule le surcoût de ces derniers par rapport à MKL sur les trois environnements testés.

	Séquentiel	parallèle CPU	super calculateur	Xeon Phi
somme	4×	1×	N.D.	6×
dot	4.5×	1×	1×	4×
nrm2	8×	2×	N.D.	7×
asum (worst)	1.5 (7)×	1 (2)×	N.D.	2 (8)×
gemv	8×	2×	N.D.	7×
trsv	25-30×	8-10×	N.D.	3-4×

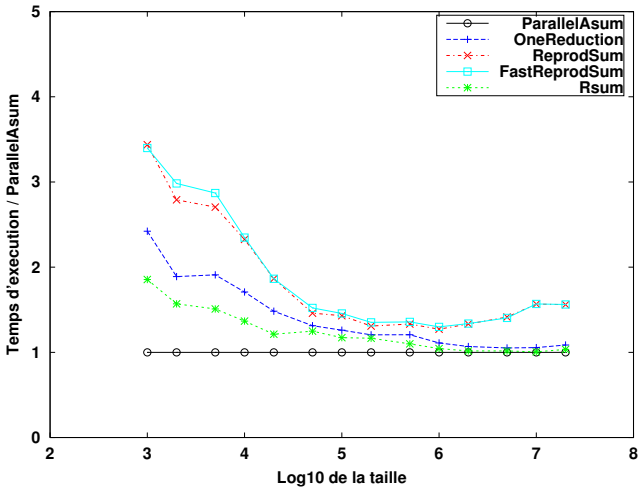
Table 0.1: Le surcoût de nos solutions par rapport à MKL

0.5 CONCLUSION

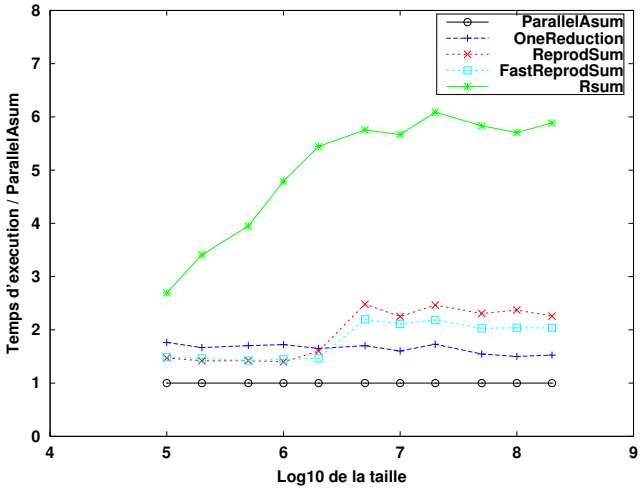
Obtenir des résultats reproductibles dans un environnement massivement parallèle est très important pour les calculs scientifiques. Nous avons présenté dans cette thèse quelques solutions pour assurer la reproductibilité numérique en parallèle. Une implémentation reproductible et précise des BLAS a été proposée. Les BLAS sont largement utilisées pour développer des



(a) Version séquentielle (env. A)

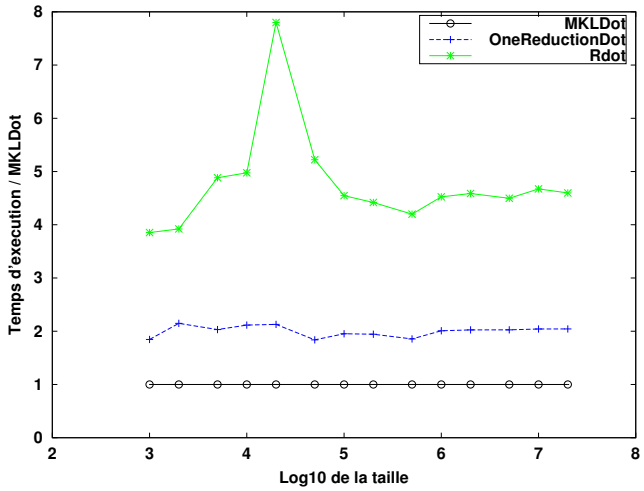


(b) CPU parallèle (env. A, 16 threads)

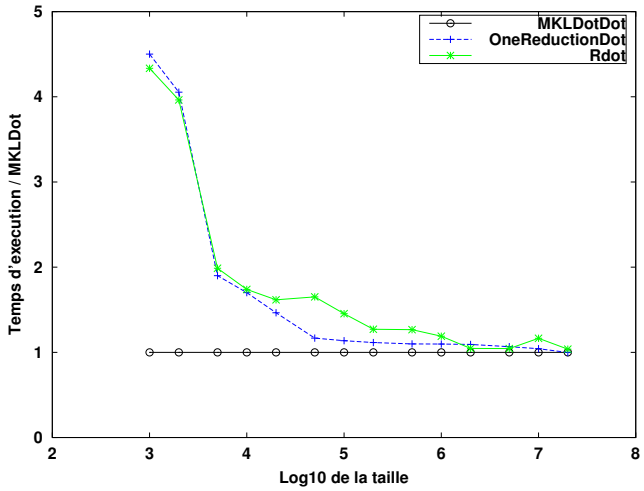


(c) Xeon Phi parallèle (env. B, 240 threads)

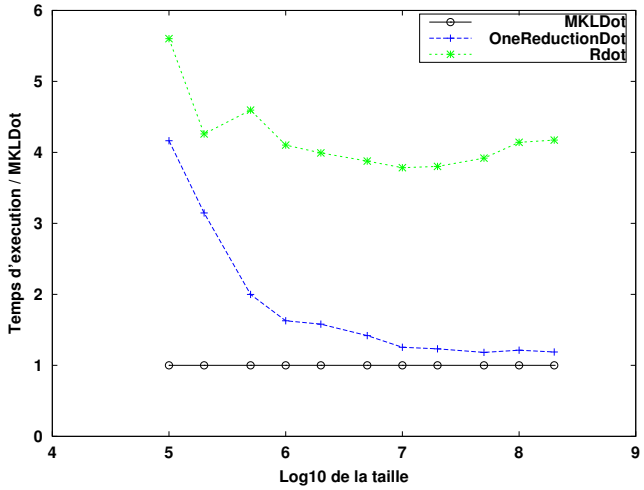
Figure o.8: Sommation (Cond = 10^8)



(a) Version séquentielle (env. A)

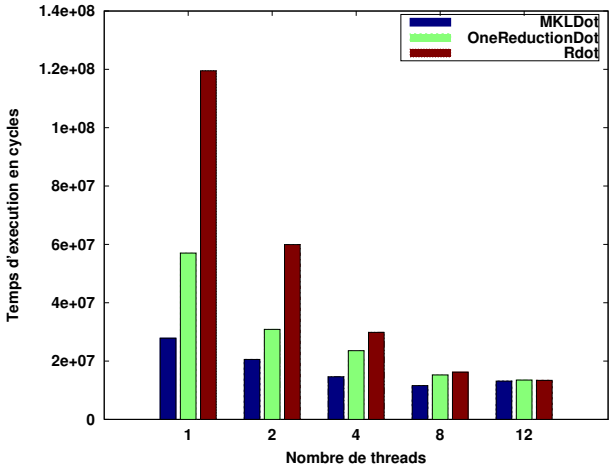


(b) CPU parallèle (env. A, 16 threads)

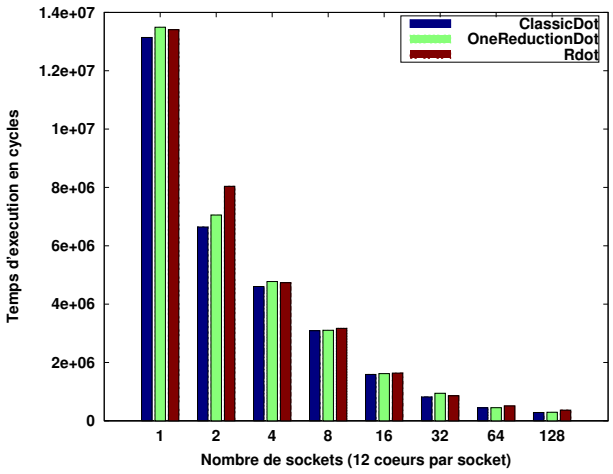


(c) Xeon Phi parallèle (env. B, 240 threads)

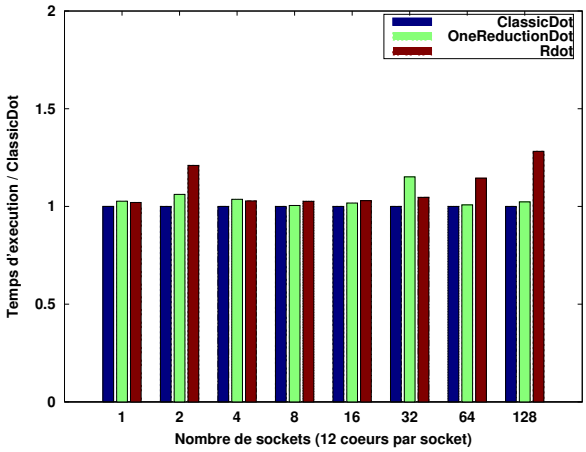
Figure 0.9: Produit scalaire (Cond = 10^8)



(a) Scalabilité sur un socket

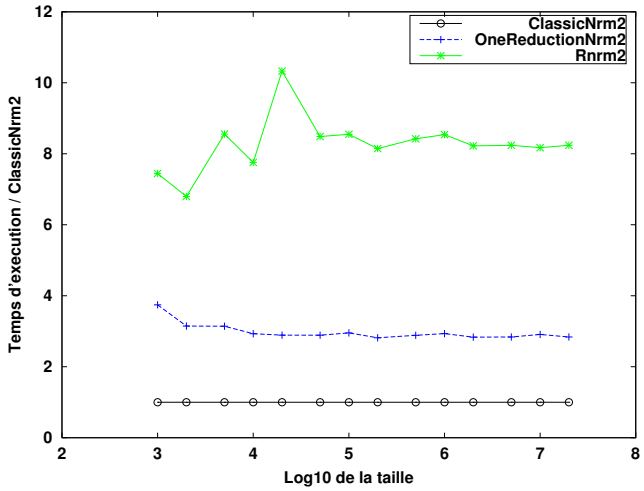


(b) Configuration multi sockets

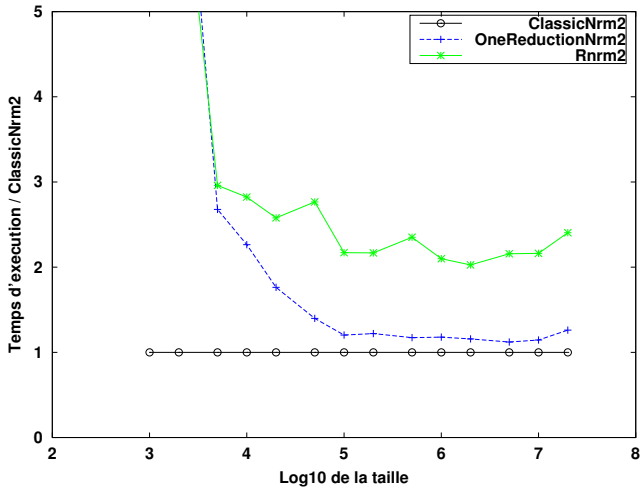


(c) Configuration multi sockets (normalisé par ClassicDot)

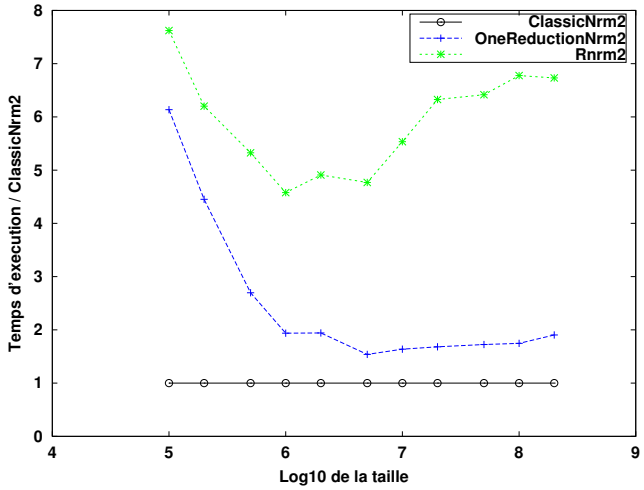
Figure 0.10: Produit scalaire (size = 10^7, Cond = 10^{32})



(a) Version séquentielle (env. A)

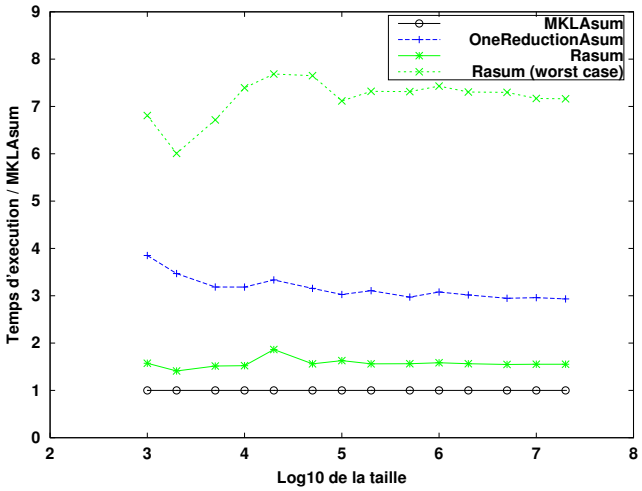


(b) CPU parallèle (env. A, 16 threads)

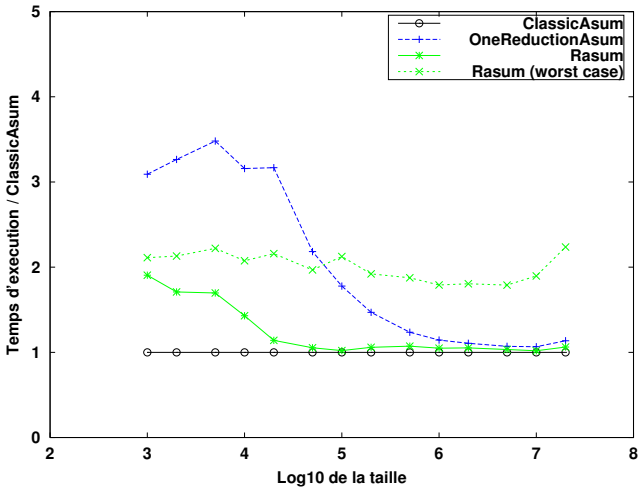


(c) Xeon Phi parallèle (env. B, 240 threads)

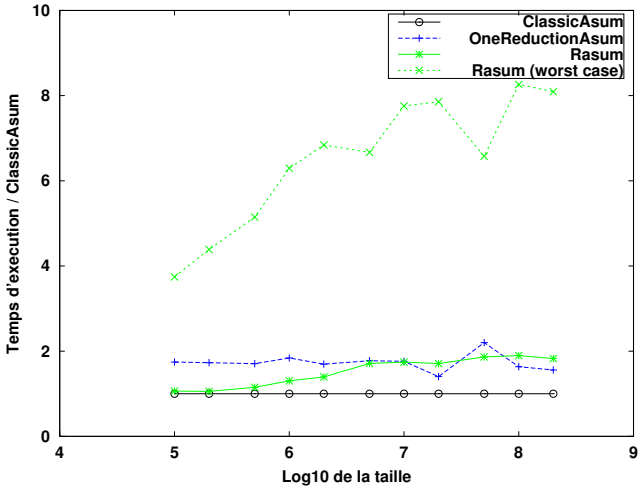
Figure 0.11: Norme euclidienne



(a) Version séquentielle (env. A)

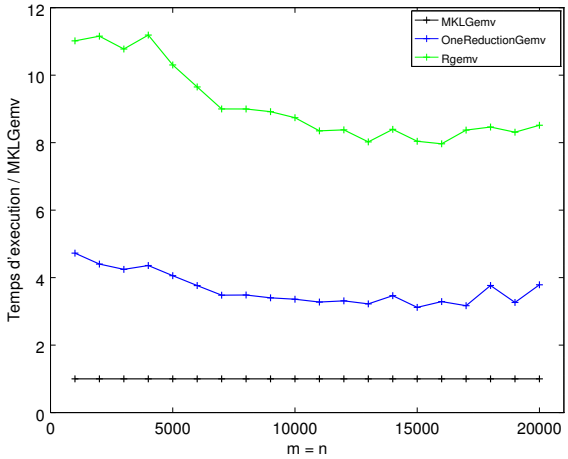


(b) CPU parallèle (env. A, 16 threads)

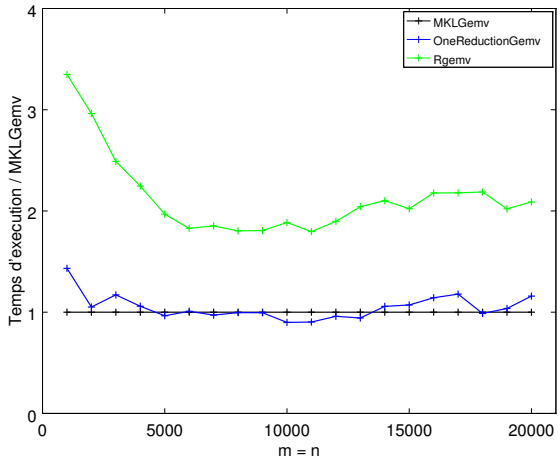


(c) Xeon Phi parallèle (env. B, 240 threads)

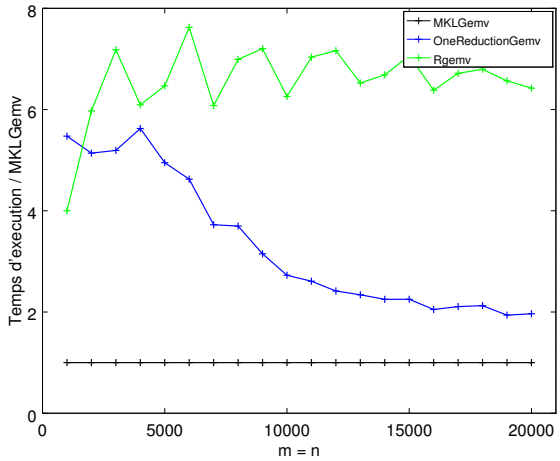
Figure 0.12: Somme des valeurs absolues



(a) Version séquentielle (env. A)

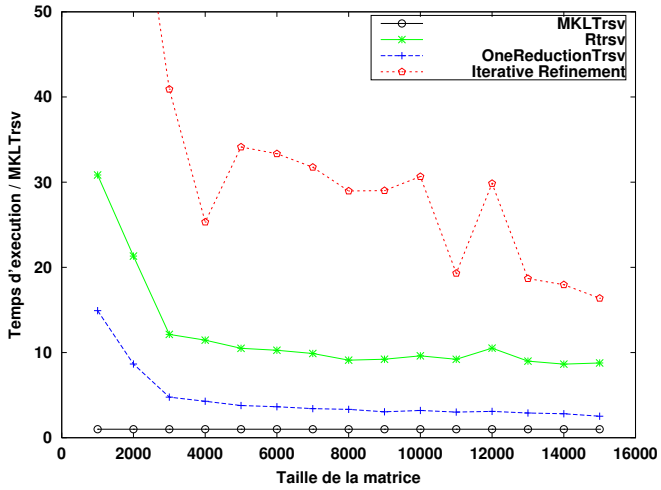


(b) CPU parallèle (env. A, 16 threads)

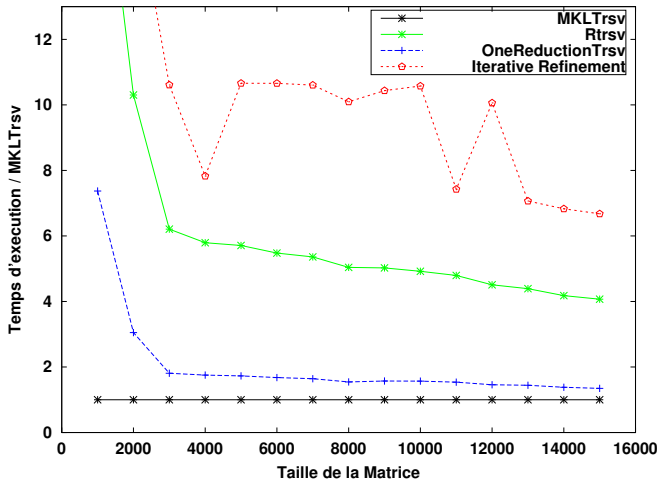


(c) Xeon Phi parallèle (env. B, 240 threads)

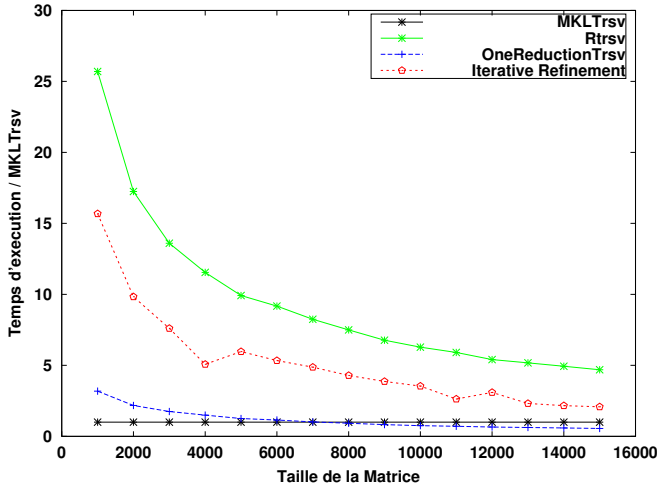
Figure 0.13: Multiplication matrice-vecteur (Cond = 10^8)



(a) Version séquentielle (env. A)



(b) CPU parallèle (env. A, 16 threads)



(c) Xeon Phi parallèle (env. B, 240 threads)

Figure 0.14: Résolution de système triangulaire (Cond = 10^8)

applications de simulation et calcul scientifique. Alors, fournir une implémentation reproductible et précise des BLAS aidera les développeurs à assurer que leurs applications se comportent de même façon indépendamment de l'environnement d'exécution. Nous rappelons les objectifs de notre implémentation des BLAS:

- Reproductibilité : la reproductibilité des résultats jusqu'au dernier bit.
- Précision: les résultats fournis par notre bibliothèque doivent être plus précis que ceux fournis par les BLAS classiques.
- Efficacité: Un surcoût minimal par rapport aux BLAS classiques et optimisées.

Des approches différentes ont été utilisées selon la nature du problème abordé. Les transformations sans erreur et les algorithmes de sommation correctement arrondie nous permettent d'implémenter un produit scalaire et une multiplication matrice-vecteur correctement arrondis ainsi qu'une norme euclidienne fidèlement arrondie. Pour la somme des valeurs absolues nous avons utilisé l'algorithme Sum2 qui assure l'arrondi correct dans la majorité des cas. Et finalement, pour les systèmes triangulaires, nous nous appuyons sur un algorithme de somme reproductible pour faire les calculs et les raffinements itératifs et assurer à la fois la reproductibilité et la précision du résultat. Malheureusement, nos approches ne sont à priori pas convenables pour le troisième niveau des BLAS à cause du surcoût mémoire qu'elles nécessitent. Pour cela, nous avons présenté des solutions pour les deux premiers niveaux seulement.

Le tableau 0.1 récapitule le surcoût de notre bibliothèque par rapport à MKL. Ce dernier varie entre 1 et $\times 30$. Dans les cas où le surcoût est acceptable pour l'utilisateur, notre bibliothèque pourrait facilement remplacer les BLAS classiques qui ne garantissent ni la reproductibilité ni la précision. Par contre, si le surcoût est trop élevé, l'utilisation de notre bibliothèque restera limitée au débogage et la validation des applications seulement, ou pour des applications dont la reproductibilité ou la précision des résultats est une forte exigence.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 3
- 1.2 How to Ensure numerical Reproducibility 3
- 1.3 Our Goal 5
- 1.4 Contributions 6

I STATE OF THE ART 9

2 BACKGROUND 11
- 2.1 Introduction 11
- 2.2 The IEEE-754 standard 11
  - 2.2.1 Rounding Modes 13
  - 2.2.2 Floating-Point Operations 14
- 2.3 Correctly or Faithfully Rounded 15
- 2.4 Rounding Errors 15
- 2.5 Cancellation 16
- 2.6 ufp, ulp and lnb 17
- 2.7 Sum of Floating-Point Numbers 18
- 2.8 Error-Free Transformations 19
  - 2.8.1 Error-Free Transformations for Addition 19
  - 2.8.2 Error-Free Transformations for Multiplication 21
- 2.9 Extra Precision 24
  - 2.9.1 Double-Double Arithmetic 24
  - 2.9.2 Floating-Point Expansions 25
  - 2.9.3 Superaccumulators 25
- 2.10 Hardware Implementation and Performance 25
  - 2.10.1 Floating-Point Addition 26
  - 2.10.2 Floating-Point Multiplication 26
  - 2.10.3 Fused Multiply and Add 27
  - 2.10.4 Pipelining 27
  - 2.10.5 SIMD Instructions 28
- 2.11 Conclusion 30

3 ACCURACY AND NUMERICAL REPRODUCIBILITY 31
- 3.1 Introduction 31
- 3.2 Non Reproducibility of the Summation 31

3.3 Reproducibility VS Accuracy 32
3.4 Accurate Summation Algorithms 33
3.4.1 Distillation Algorithms 33
3.4.2 SumK 34
3.4.3 iFastSum 35
3.4.4 AccSum 38
3.4.5 FastAccSum 39
3.4.6 HybridSum 40
3.4.7 OnlineExact 41
3.4.8 How to Choose the Right Summation Algorithm? 42
3.5 Solutions for Reproducible Summation 43
3.5.1 ReprodSum 43
3.5.2 FastReprodSum 44
3.5.3 Indexed Floating-Point Numbers 44
3.5.4 1-Reduction 45
3.5.5 Static Scheduling and Deterministic Reduction 46
3.6 Reproducible BLAS and Summation 47
3.6.1 Conditional Numerical Reproducibility 47
3.6.2 ReproBLAS Library 47
3.6.3 ExBLAS library 47
3.7 Conclusion 48

II TOWARDS REPRODUCIBLE AND ACCURATELY ROUNDED BLAS 51

4 PARALLEL REPRODUCIBLE AND CORRECTLY ROUNDED SUMMATION 53
4.1 Introduction 53
4.2 Implementation and optimization of Sequential Summation 53
4.2.1 Optimization of HybridSum 54
4.2.2 Optimization of OnlineExact 61
4.2.3 Runtime Performance for Summation Algorithms 63
4.2.4 Our Hybrid Summation Algorithm Rsum 65
4.3 Parallel Correctly Rounded Sum 66
4.4 Runtime Performance for our Parallel Summation 67
4.5 Accuracy Results 70

4.6 Compare to ExBLAS 72
4.7 Conclusion 76
5 REPRODUCIBLE LEVEL 1 BLAS 77
5.1 Introduction 77
5.2 Dot Product 77
5.2.1 Sequential Dot Product 78
5.2.2 Parallel Dot Product 81
5.2.3 Runtime Performance Results 83
5.2.4 Accuracy Results 87
5.3 Euclidean Norm 87
5.4 Sum of Absolute Values 90
5.4.1 Parallel Version 92
5.4.2 Performance Results 93
5.5 Conclusion 95
6 REPRODUCIBLE LEVEL 2 BLAS 97
6.1 Introduction 97
6.2 Matrix Vector multiplication 97
6.2.1 Sequential Correctly Rounded Algorithm 97
6.2.2 Parallel Correctly Rounded Algorithm 98
6.2.3 Running Time Results 99
6.2.4 Accuracy Results 101
6.3 Triangular Solver 102
6.3.1 Reproducible and Accurate trsv 105
6.3.2 Accuracy Results 108
6.3.3 Performance Results 111
6.4 Conclusion 113
7 CONCLUSION AND FUTURE WORK 115
7.0.1 Future Work 117
III APPENDICES 119
A ABOUT INTEL MKL CONDITIONAL NUMERICAL REPRODUCIBILITY 121
A.1 Introduction 121
A.1.1 How to Use CNR Feature 121
A.2 Reproducibility and Performance 122
A.2.1 Test Methodology 122
A.2.2 Numerical Results 123
A.2.3 Performance Results 127
A.3 Conclusion 132

B WORK METHODOLOGY 133
B.1 Introduction 133
B.2 Test Environments 133
B.3 Performance Evaluation 135
B.4 Data Generation 135
B.4.1 Summation and Dot Product Data Generator 135
B.4.2 Matrix Vector Multiplication Data Generator 136
B.4.3 Triangular Systems Data Generator 136

BIBLIOGRAPHY 146

LIST OF FIGURES

Figure 2.1 Rounding to nearest 14
Figure 2.2 Directed rounding towards zero 14
Figure 2.3 Rounding error for addition 15
Figure 2.4 Faithfully and correctly rounding (Rounding to nearest) values 16
Figure 2.5 ulp , ufp and lnb of a floating-point number (only the mantissa is presented) 18
Figure 3.1 Input and output of the ExtractSclalar 39
Figure 3.2 Algorithm AccSum 40
Figure 3.3 Algorithm 1-Reduction (K = 2) 45
Figure 3.4 The principle of summation used within ExBLAS 48
Figure 4.1 Performance results for summation algorithms 64
Figure 4.2 Performance according to condition number (vector size = 50000) 65
Figure 4.3 Parallel algorithm for correctly rounded sum (entry vector p is distributed to every thread, here 2) 66
Figure 4.4 Rsum: scaling (environment A, cond = 10^8) 68
Figure 4.5 Rsum: Performances (Cond = 10^8) 69
Figure 4.6 Accuracy results for different summation algorithms (vector size = 100000) 71
Figure 4.7 Rsum vs ExBLAS: relative performance (ratio/size), summation, sequential case 73
Figure 4.8 Rsum vs ExBLAS: relative performance (ratio/size), summation, parallel case (threads = 16) 74
Figure 4.9 Impact of the exponent range on summation algorithm efficiency (vector size = 10^5) 75
Figure 5.1 Algorithm for a sequential correctly rounded dot product 78

Figure 5.2 Parallel algorithm for correctly rounded dot product (input vectors x and y are distributed to every thread, here 2) 82
Figure 5.3 Performance results for dot product on shared memory systems (Cond = 10^8) 85
Figure 5.4 Performance results for dot product on distributed memory system (size = 10^7, Cond = 10^{32}) 86
Figure 5.5 Accuracy results for dot product (size = 10^5) 88
Figure 5.6 Performance results for euclidean norm 89
Figure 5.7 Performance results for sum of absolute values 94
Figure 6.1 Parallel algorithm for correctly rounded matrix-vector multiplication (4 threads case) 99
Figure 6.2 Performance results for matrix vector multiplication (Cond = 10^8) 100
Figure 6.3 Accuracy results for matrix vector multiplication ($m = n = 12000$) 102
Figure 6.4 Parallel triangular solver and example of dependencies between $trsv$ and $gemv$ 104
Figure 6.5 Parallel triangular solver relying on correctly rounded dot product 106
Figure 6.6 Recursive parallel triangular solver 107
Figure 6.7 Accuracy of triangular solvers (size = 1000) 109
Figure 6.8 Performance results for triangular solver (Cond = 10^8) 112
Figure A.1 Numerical reproducibility for two different environments with CNR turned off 124
Figure A.2 Numerical reproducibility for two different environments with MKL_CBWR set to AVX 125
Figure A.3 Numerical reproducibility for different thread configurations (environment A) 126
Figure A.4 Non reproducibility of $gemm$ based on SSE2 for different number of threads, Size = 1500 (Focus of Figure A.3c) 127
Figure A.5 Performance of sequential MKL on environment A 128

Figure A.6 Performance of parallel MKL on environment A 129
Figure A.7 Performance results for gemm (environment D) 130
Figure A.8 Floating Point Capabilities of a Single Core on Intel Processors 131
Figure B.1 Matrix 180° rotation 137

LIST OF TABLES

Table 2.1 Numerical values according to exponent (inspired from [45]) 12
Table 2.2 Standard floating-point number formats 13
Table 2.3 Latency and throughput of floating-point operations [30] 28
Table 7.1 Extra-cost ratio of our reproducible and accurate solutions 116
Table B.1 Experimental frameworks 134

LIST OF ALGORITHMS

Algorithm 2.1 The classical accumulation algorithm 18
Algorithm 2.2 Algorithm FastTwoSum (Dekker, 1971) 20
Algorithm 2.3 Algorithm TwoSum (Knuth, 1969) 20
Algorithm 2.4 Priest's transformation algorithm (1992) 21
Algorithm 2.5 Veltkamp's Split algorithm (1968) 22
Algorithm 2.6 Algorithm TwoProd 23
Algorithm 2.7 Algorithm 2MultFMA 24
Algorithm 2.8 Pipelined summation algorithm 29
Algorithm 3.1 Algorithm VecSum 34
Algorithm 3.2 Algorithm SumK 35
Algorithm 3.3 Algorithm iFastSum 37
Algorithm 3.4 Algorithm ExtractScalar 39
Algorithm 3.5 Split algorithm 41
Algorithm 3.6 HybridSum transformation 41
Algorithm 3.7 OnlineExact transformation 42
Algorithm 4.1 HybridSum transformation with single exponent computation 60
Algorithm 4.2 OnlineExact transformation using FastTwoSum 62
Algorithm 5.1 Correctly rounded sum of absolute values 91
Algorithm 6.1 Correctly rounded matrix-vector multiplication 98
Algorithm 6.2 Forward substitution algorithm 103
Algorithm 6.3 Reproducible iterative refinement 108

1

INTRODUCTION

Supercomputer computational power is increasing exponentially. Given this trend, exascale computing, i.e. 10^{18} floating-point operations per second is likely to be reached before 2020. Since the CPU clock speed is hitting energy and heat walls, it can no more be increased efficiently. The computational power of recent computers is increased by several sophisticated architectural improvements supporting more ILP (Instruction Level Parallelism), refining instruction sets to introduce fused operations (like Fused Multiply and Add) and larger registers for SIMD (Single Instruction Multiple Data) instructions, or by using a huge number of processors and perform parallel computations using a lot of threads or processes.

The growth of this computational power allows us to solve more and more complex problems. However it also increases the amount of rounding errors, their propagation and possibly their maleficent effects. Tasks are likely to be scheduled dynamically to balance irregular workload. Therefore, repeated runs of the same program on the same input would not perform operations in the same order. Due to round off errors, non-associativity of floating-point addition and dynamic behavior in massively parallel systems, numerical reproducibility and accuracy of large scale scientific simulations are becoming critical issues; examples are presented in Section 1.1. In this thesis, we focus in particular on accuracy and numerical reproducibility of parallel BLAS which is ubiquitous in scientific computing.

We define numerical reproducibility as getting the same result up to the last bit from multiple runs of the same program for the same input. The main source of non-reproducibility is the non-associativity of floating-point addition. Furthermore, various factors can change the order and/or the semantic of floating-

point operations. It is informative to distinguish 3 levels for non-reproducibility problem.

1. **Obtaining reproducible results for a given processor with the same number of threads**
   In parallel environments, dynamic task scheduling and non deterministic reductions are employed to balance workloads dynamically and reduce the running time. However, they introduce a non-deterministic behavior that might modify the operation order from one run to another even if the same number of threads is used.

2. **Obtaining reproducible results for a given processor with different thread configurations**
   Even on the same processor, running a program with different numbers of threads would also change data distribution, and therefore changes the operation order, which leads to non-reproducible numerical results.

3. **Obtaining reproducible results for different processors with different architectures**
   Another factor that might influence numerical results is the processor architecture and the supported instruction set. Some instructions are supported on some processors only, register and floating-point unit sizes for SIMD operations also vary from one processor to another, which leads to change operation order. Therefore, porting even a serial source code across different architectures would lead to binaries that generate different results for the same input data.

There are other potential sources of non-reproducibility. For example compiling the same source with different compilers, different compiler versions, different compiler flags or linking with different math libraries would also lead to non-reproducible results. However, most of these compiler dependencies are beyond the scope of our work and corresponds to classical (but not simple) porting issues. We consider only the case where we control the compiler flags and prevent the use of aggressive optimizations, especially those that are opaque and not documented.

1.1 MOTIVATION

Obtaining reproducible results in parallel is already a challenge on today's computers. Issues have been reported in fluid dynamics [51], hydrodynamics [36] and energy [66] simulations, climate modeling [23], GPU accelerated molecular dynamic simulations [61] and a lot of other scientific applications. Numerical reproducibility is very important for debugging since it is very hard to differentiate numerical errors and implementation errors if the results are not reproducible. Published results should also be reproducible so they can be replicated and studied carefully. Numerical reproducibility is listed as one of the top ten challenges for future exascale supercomputers and it is even considered in some cases as a measure of code correctness [38].

1.2 HOW TO ENSURE NUMERICAL REPRODUCIBILITY

In response to non-reproducibility failures, several techniques have been proposed. The most prevalent ones are presented here.

- **Deterministic Behavior**
  This technique ensures that the order of operations does not change from one run to another. For this, data should be statically scheduled, parallel reductions have to be deterministic and the used instruction set should be the same even on different processors. Parallel programming libraries as OpenMP [63] and TBB [64] implement static scheduling and deterministic reductions. Intel MKL library [31] starting with its 11.0 release has introduced the CNR (Conditional Numerical Reproducibility) [33] feature that allows users to specify the used instruction set for its functions. Therefore the same instruction set could be used for different architectures to ensure reproducible results. The drawback of this technique is that the number of threads and the used instruction set should remain the same between different runs for the data to be scheduled in the same way. This defeats the purpose of using more recent processor and leads to a significant performance drop if such imple-

mentation is ported to a processor with more cores or with a more recent instruction set.

- **Improving Result Accuracy**
  It might solve the reproducibility problem for a specific application [36] and is not considered as a generic solution. Higher accuracy could be achieved either by using more accurate algorithms [44, 55] or using higher precision. Note that even for improved accuracy, the result still depends on operation order. Therefore, if the problem is ill conditioned this technique does not recover the full bitwise reproducibility, yet by improving accuracy, even if results are not exactly reproducible the difference is small enough to trust the program [66].

- **Deterministic Error**
  Demmel and Nguyen [12] came up with this technique that consists in pre-rounding the inputs to commit the same errors in any different run configuration. The sum of pre-rounded inputs is exact independently from the order of operations. Therefore the result is the same for any order and does not depend on thread configuration nor register size. Despite ensuring numerical reproducibility, this solution does still have accuracy issues, especially when the problem is ill conditioned. This solution is explained in more details in chapter 3.

- **Correctly Rounded Results**
  Recent works [9, 41] present different algorithms to calculate correctly rounded parallel sum. Although, the authors tackle the problem using different approaches, they agree on the fact that parallel correctly rounded summation can be performed at minimal or even no extra-cost compared to classical summation. Being unique, providing the correctly rounded results systematically solves the reproducibility problem.

1.3 OUR GOAL

Our work does not focus on one specific application. A parallel library that ensures reproducible results independently from the running environment is provided instead. We choose to implement reproducible BLAS (Basic Linear Algebra Subroutines) since this library is commonly used to develop high quality scientific applications. Note also that most major hardware manufacturers provide their own BLAS implementations that are very well optimized for their architectures. We mention here Intel MKL library [31], ACML [2] provided by AMD, ESSL [18] which is optimized for IBM architectures and cuBLAS [72] from Nvidia. Therefore, relying on BLAS allows developers to implement portable and efficient applications with minimal effort. Providing reproducible BLAS implementation that return the same results independently from the running environment would solve the reproducibility problem or at least improve result reproducibility in most applications that rely on BLAS: It suffices to link the application with our library which of course provides the same interface. We define next the goals that should be satisfied by such a reproducible BLAS implementation, and that will be ours in this work.

- **Bitwise Numerical Reproducibility**
  Whenever a subroutine is called with the same input data it should always provide the bitwise identical result. The only requirement is that the running environment should implement the IEEE-754 standard for floating-point arithmetic [26].

- **Maximum Accuracy**
  Computing a non-accurate answer is hard to justify even if it is reproducible. We aim at providing a reproducible answer with maximum accuracy. The correctly rounded result is provided when possible. However, for some BLAS routines (like triangular solver) it is very hard if not impossible to ensure that the result is the correctly rounded value of the solution. Therefore, when correctly rounded results can not be provided we ensure at least that our subroutines besides being reproducible are more accurate than

classical implementations. More details about our BLAS accuracy are given in Chapters 5 and 6.

- **Running Time Performance**
  In this scope, running time is as important as result accuracy. Since most BLAS implementations are focused on performance, our implementation should be competitive enough to be actually useful in practice. So, we try to ensure maximum performance without compromising the result accuracy.

1.4 CONTRIBUTIONS

The aim of this work is to provide a reproducible BLAS implementation. BLAS functions are classified into 3 levels.

1. The level 1 BLAS contains functions of $O(n)$ complexity (scalar-vector and vector-vector operations).
2. The level 2 BLAS contains functions of $O(n^2)$ complexity (matrix-vector operations).
3. The level 3 BLAS contains functions of $O(n^3)$ complexity (matrix-matrix operations).

We focus in this work on level 1 and level 2 subroutines. Since most reproducibility problems in BLAS are originated from the non-associativity of floating-point summation we design first a parallel correctly rounded summation algorithm. Afterwards, we rely on this latter to implement reproducible, correctly or faithfully rounded level 1 BLAS for shared memory parallel systems. Our first results are published in [6]. Then, our approach is extended to level 2 BLAS. We provide implementations for both distributed memory parallel systems and Intel Xeon Phi accelerator to validate our approach on different environments. These latter cover all platforms that are significant of today's practice of floating-point computing. These results have been published in [8, 7].
This document is organized as following.

CHAPTER 2: In this chapter we present the basic concepts about floating-point numbers and their main properties as formats, rounding modes and hardware implementations. Some advanced concepts about error-free transformations are also presented.

CHAPTER 3: This chapter focuses on summation and explain the source of numerical reproducibility problem. Existing solutions to obtain numerical reproducibility are also addressed in more details.

CHAPTER 4: We present in this chapter our correctly rounded summation algorithm. We study its accuracy and its performance on CPU and on Xeon Phi accelerator. Our summation algorithm is also compared to other available solutions in different scenarios to exhibit the advantages and the drawbacks of each solution.

CHAPTER 5: In this chapter, parallel, reproducible and accurate algorithms are introduced for level 1 BLAS. The addressed subroutines are the dot product (dot), the sum of absolute values (asum) and the euclidean norm (nrm2). Our implementation ensures correctly rounded dot and asum, while for nrm2 we ensure that the result is faithfully rounded besides being reproducible.

CHAPTER 6: The last chapter of this document deals with level 2 BLAS. we focus on the two subroutines: matrix-vector multiplication (gemv) and triangular solver (trsv). We rely on the dot product from the previous chapter to implement a correctly rounded gemv.

For trsv unfortunately we do not ensure that the result is correctly rounded. However we show that it is reproducible and much more accurate than classic BLAS implementation.

Part I

STATE OF THE ART

2

BACKGROUND

2.1 INTRODUCTION

Floating-point numbers are used to approximate real numbers on today's computers. In this chapter we describe the IEEE-754 [26] standard for floating-point arithmetic. Hardware implementation and performance of floating-point operations are presented here. We also introduce the floating-point errors for elementary operations and for the sum of a floating-point vector. Since we aim at providing a reproducible implementation of BLAS (Basic Linear Algebra Subprograms) [3], this summation is a very important building block. And finally we tackle some more advanced concepts as error-free transformations and extra precision.

2.2 THE IEEE-754 STANDARD

The IEEE-754 [27] is a widely adopted standard that defines the formats, the exceptions, the operations and the rounding rules for floating-point numbers. This standard has been a successful attempt to provide a consistent floating-point number representation to be used by most of machines.

A floating-point number is composed from a sign, a mantissa and an exponent. The representation is based upon the scientific notation using the radix two nearly on all computers. Three primary binary formats are defined in the IEEE-754 standard: binary32, binary64 and binary128 are encoded on 32, 64 and 128 bits respectively. The 2008 version of the standard introduced also two formats for decimal floating-point numbers. However these formats are mainly motivated by financial applications (for example when the exchange rate is defined in decimal and can not be exactly presented in binary formats) and will not be con-

sidered in this document.

The value of a floating-point number x is computed using the following formula :

$$x = (-1)^{sign} \times mantissa \times 2^{exponent-bias}, \tag{2.1}$$

where,

- **sign** takes the value 0 for a positive or 1 for a negative floating-point number. A single bit is used for sign in all three formats.
- **mantissa** is a binary string

$$mantissa = b_0.b_1b_2...b_{m-1}, \tag{2.2}$$

  Such that the value of the mantissa size m depends on the different IEEE-754 formats.
  - $m = 24$ for $binary32$.
  - $m = 53$ for $binary64$.
  - $m = 113$ for $binary128$.

Exponent value E	Numerical value
$(0...00)_2 = 0$	$2^{sign} \times 0.b_1b_2...b_{m-1} \times 2^{1-bias}$
$(0...01)_2 = 1$	$2^{sign} \times 1.b_1b_2...b_{m-1} \times 2^{1-bias}$
$(0...10)_2 = 2$	$2^{sign} \times 1.b_1b_2...b_{m-1} \times 2^{2-bias}$
.	.
.	.
$(1...10)_2 = X$	$2^{sign} \times 1.b_1b_2...b_{m-1} \times 2^{X-bias}$
$(1...11)_2$	if $mantissa = 0$ than $\pm\infty$ else NaN (Not a Number)

Table 2.1: Numerical values according to exponent (inspired from [45])

In binary, $b_i \in \{0, 1\}$ for $i > 0$, while $b_0 = 1$ for normalized floating-point numbers and $b_0 = 0$ for subnormal ones (see Table 2.1). Therefore, only the fractional part is stored and there is no need to store b_0.

- **exponent** defines the shift of the floating-point in the mantissa. If the value of "exponent - bias" is positive the point is shifted to the right. Otherwise, the point is shifted to the left. Table 2.1 shows the possible values corresponding to the exponent of a floating-point number. The properties of the defined formats by the IEEE-754 standard are presented in Table 2.2.

Format	Exponent size	Mantissa size m	Bias value
binary32	8 bits	24 bits (23 stored)	127
binary64	11 bits	53 bits (52 stored)	1023
binary128	15 bits	113 bits (112 stored)	16383

Table 2.2: Standard floating-point number formats

The machine precision ϵ is defined to be the distance between 1 and its next larger floating-point neighbor. We have

$$\epsilon = 2^{1-m},$$

where m is the mantissa length in Relation (2.2).

2.2.1 *Rounding Modes*

The standard defines two types of rounding modes, roundings to nearest and directed roundings.

2.2.1.1 *Roundings to Nearest*

ROUNDING TO NEAREST, TIES TO EVEN rounds to the nearest floating-point value. If the number falls at the same distance from 2 successive floating-point numbers, it is rounded to the even (zero least significant bit) floating-point number (see Figure 2.1).

ROUNDING TO NEAREST, TIES AWAY FROM ZERO also rounds to the nearest floating-point value. If the number falls at the same distance from 2 successive floating-point numbers, it is here rounded to the farthest from zero, i.e. the largest in absolute value.

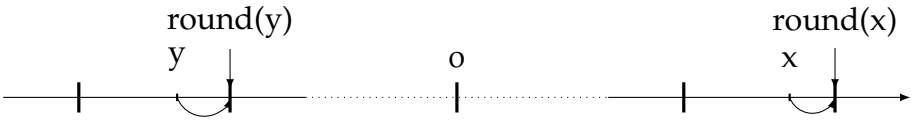


Figure 2.1: Rounding to nearest

2.2.1.2 *Directed Roundings*

ROUNDING TOWARDS POSITIVE INFINITY rounds to the closest larger floating-point number.

ROUNDING TOWARDS NEGATIVE INFINITY rounds to the closest smaller floating-point number.

ROUNDING TOWARDS ZERO rounds to the closest floating-point number which is smaller in absolute value (see Figure 2.2).

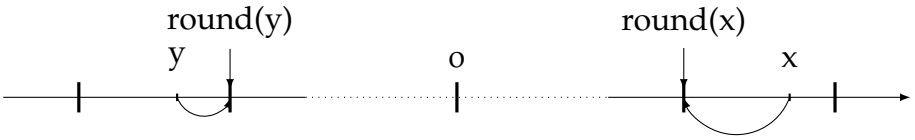


Figure 2.2: Directed rounding towards zero

2.2.2 *Floating-Point Operations*

The IEEE-754 standard requires the five basic algebraic operations $(+, -, \times, /, \sqrt{})$ to be correctly rounded according to the used rounding mode. In this document we denote the four standard floating-point operations $\oplus, \ominus, \otimes, \oslash$, which respectively correspond to the real arithmetic operations $+, -, \times, /$.

For two floating-point numbers a and b, the result of the addition $+$ is not necessarily a floating-point number. The floating-point operation $\oplus$ guarantees that the result is a representable floating-point number such that:

$$a \oplus b = \mathrm{round}(a + b),$$

where $round(x)$ is a rounding map that computes the floating-point number corresponding to the real number x. When $a \oplus b \neq a + b$, the floating-point addition introduces a rounding error into its result. See Figure 2.3 for a visual explanation of the add result and its rounding error.

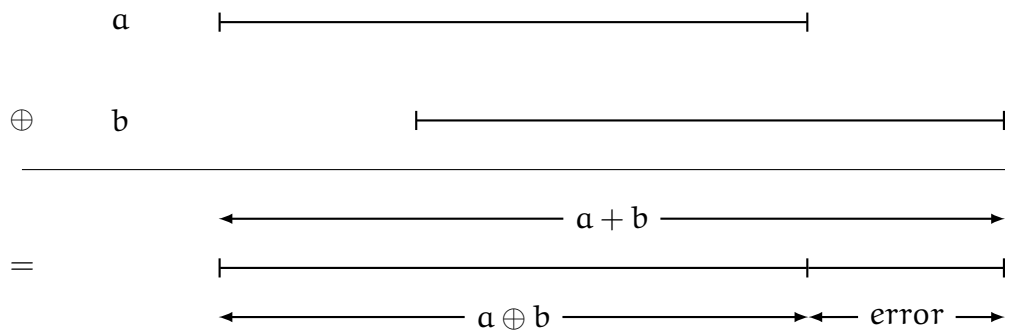


Figure 2.3: Rounding error for addition

2.3 CORRECTLY OR FAITHFULLY ROUNDED

For a given real number x, the correctly rounded value depends the used rounding mode. We focus in this document on rounding to nearest (except when we state explicitly another rounding mode). Therefore we define the *correctly rounded* result as the closest floating-point number to the real number x.

A *faithfully rounded* value of x is defined as one of the two floating-point numbers that surround it if it is not a floating-point number, while it equals x if this latter is a floating-point number.
Figure 2.4 shows a visual explanation for correctly and faithfully rounded results.

2.4 ROUNDING ERRORS

Floating-point operations approximate the result to one appropriate floating-point number. The generated error depends on the used rounding mode. If x is the exact result we have:

$$|x - round(x)| < u \cdot |x|,$$

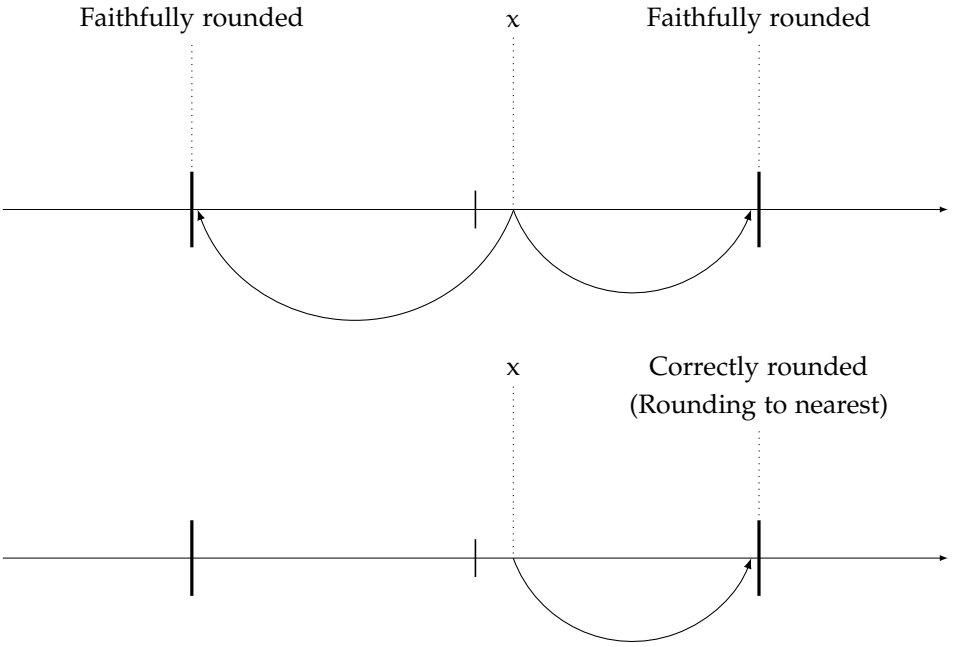


Figure 2.4: Faithfully and correctly rounding (Rounding to nearest) values

where the rounding unit u is defined as

$$\begin{aligned} u &= 2^{-m} \text{ for roundings to nearest, and} \\ u &= 2^{1-m} \text{ for directed roundings.} \end{aligned} \tag{2.3}$$

The value of $|x - \mathrm{round}(x)|$ is the absolute error which can also be estimated relatively to $|x|$ as

$$\frac{|x - \mathrm{round}(x)|}{|x|} \leqslant u.$$

The value of $|x - \mathrm{round}(x)| / |x|$ is known as the *relative rounding error*.

2.5 CANCELLATION

Cancellation occurs when we subtract a floating-point number from another that is approximately equal to it, i.e. the addition of two floating-point numbers with very close values and different signs.

Sterbenz has shown in [60] that if $a/2 \leqslant b \leqslant 2a$ then the subtraction of a and b is exact ($a \ominus b = a - b$). Accordingly the

subtraction of two floating-point numbers that have the same exponent can be performed with no error. Therefore, the cancellation itself does not generate errors, yet it magnifies the errors already introduced by the previous operations when the highest significant bits (or digits) cancel. Note that the cancellation increases the relative error while the absolute error is not affected.

2.6 UFP, ULP AND LNB

We present in this section the three functions ulp, ufp and lnb of a floating-point number. Those functions will be used later for our demonstrations and error analysis.

ufp : stands for "Unit in the First Place". This concept has been introduced in [54]. For a given real number x we have:

$$\begin{aligned} \text{if } x \neq 0 \Rightarrow \mathrm{ufp}(x) = 2^{\lfloor \log_2 |x| \rfloor}, \\ \text{if } x = 0 \Rightarrow \mathrm{ufp}(x) = 0. \end{aligned}$$

In other words, we could define $\mathrm{ufp}(x)$ as the weight of the leading bit in the mantissa of x.

ulp : stands for "Unit in the Last Place". For a normalized floating-point number x, it verifies:

$$\mathrm{ulp}(x) = \mathrm{ufp}(x) \cdot 2u. \tag{2.4}$$

lnb : stands for "Last Non-zero Bit". Introduced in [46], $\mathrm{lnb}(x)$ is the weight of the last bit with positive value in the mantissa of x. For a floating-point number x we know that:

$$\mathrm{ulp}(x) \leqslant \mathrm{lnb}(x) \leqslant \mathrm{ufp}(x).$$

Note that both ufp and lnb are defined for every real number while ulp depends on the used floating-point format.

Given two real numbers x, y, we have

$$\begin{aligned} \mathrm{lnb}(x) \geqslant \mathrm{ufp}(x) \cdot 2u &\iff \mathrm{round}(x) = x, \\ \mathrm{lnb}(x) < \mathrm{ufp}(x) \cdot 2u &\iff \mathrm{round}(x) \neq x. \end{aligned} \tag{2.5}$$

and

$$\begin{aligned} \mathrm{lnb}(x+y) &\geqslant \min(\mathrm{lnb}(x), \mathrm{lnb}(y)), \\ \mathrm{lnb}(x \times y) &= \mathrm{lnb}(x) \times \mathrm{lnb}(y). \end{aligned} \tag{2.6}$$

Figure 2.5 illustrates those different concepts.

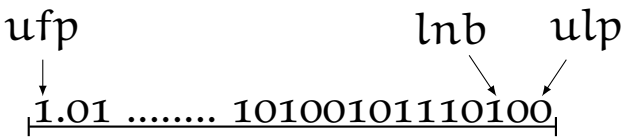


Figure 2.5: ulp , ufp and lnb of a floating-point number (only the mantissa is presented)

2.7 SUM OF FLOATING-POINT NUMBERS

Let p be a vector of n floating-point numbers and let $S = \sum_{i=1}^{n} p_i$ be the exact sum. We have seen that the IEEE-754 standard requires addition to be correctly rounded. However this rule does not apply to the accumulation of n floating-point numbers. Let $\widehat{S}$ be the result of the floating-point accumulation of p_i computed using Algorithm 2.1.

Algorithm 2.1 The classical accumulation algorithm

Input: p: a vector of n floating-point numbers
Output: $\widehat{S}$: the evaluated sum of p_i

1: $\widehat{S} = 0$
2: **for** $(i = 1 : n)$ **do**
3: $\quad \widehat{S} = \widehat{S} \oplus p_i$
4: **end for**

The difference between S and $\widehat{S}$ is the absolute error of the classic summation algorithm. It has been shown in [40] that for rounded to nearest elementary additions we have:

$$\left| S - \widehat{S} \right| \leqslant \gamma_{n-1} \cdot \sum_{i=1}^{n} |p_i| \,, \tag{2.7}$$

where

$$\gamma_n = \frac{nu}{1-nu}. \tag{2.8}$$

The relative error of Algorithm 2.1 can be calculated from Relation (2.7). Dividing the both sides by $|S|$, we have:

$$\frac{\left|S-\widehat{S}\right|}{|S|} \leqslant \gamma_{n-1} \cdot \frac{\sum_{i=1}^{n}|p_i|}{|S|} = \gamma_{n-1} \cdot \frac{\sum_{i=1}^{n}|p_i|}{|\sum_{i=1}^{n} p_i|}. \tag{2.9}$$

The relative error depends on the machine precision, the vector size and the condition number. Here the condition number of the sum is defined by the expression:

$$\mathrm{cond}\left(\sum_{i=1}^{n} p_i\right) = \frac{\sum_{i=1}^{n}|p_i|}{|\sum_{i=1}^{n} p_i|}. \tag{2.10}$$

Therefore, Relation (2.9) can be written as:

$$\frac{\left|S-\widehat{S}\right|}{|S|} \leqslant \gamma_{n-1} \cdot \mathrm{cond}\left(\sum_{i=1}^{n} p_i\right).$$

Due to cancellations the value of $|\sum_{i=1}^{n} p_i|$ can be very small compared to $\sum_{i=1}^{n}|p_i|$. This leads to an *ill conditioned problem*. If there is not much cancellations, the value of $|\sum_{i=1}^{n} p_i|$ would be close to $\sum_{i=1}^{n}|p_i|$ and the condition number will be small. In this case, the problem is said to be well conditioned.

2.8 ERROR-FREE TRANSFORMATIONS

We focus in this section on error-free transformations for addition and multiplication operations. In this case, an error-free transformation is an algorithm that provides an exact result for an elementary floating-point operation as the sum of more than one floating-point numbers.

2.8.1 *Error-Free Transformations for Addition*

2.8.1.1 *Algorithm* FastTwoSum

This algorithm has been introduced by Dekker [10]. Given two floating-point numbers a and b, such that the exponent of a is

larger than or equal to the exponent of b[1] [40], and a rounding to nearest mode is used. FastTwoSum computes the two floating-point numbers s and e such that:

$$\begin{aligned} s &= a \oplus b, \\ a + b &= s + e. \end{aligned} \tag{2.11}$$

Algorithm 2.2 Algorithm FastTwoSum (Dekker, 1971)

Input: a, b: floating-point numbers
Output: s, e: floating-point numbers
Requires: $exponent(a) \geqslant exponent(b)$

1: $s = a \oplus b$
2: $t = s \ominus a$
3: $e = t \ominus b$

2.8.1.2 *Algorithm* TwoSum

This algorithm has been introduced by Knuth [34]. It also computes floating-point values s and e that satisfy Relation (2.11), and works properly only in roundings to nearest. Nevertheless, TwoSum does not require any preliminary knowledge of the order of magnitude of a and b. It requires 6 floating-point operations instead of 3 floating-point operations with FastTwoSum.

Algorithm 2.3 Algorithm TwoSum (Knuth, 1969)

Input: a, b: floating-point numbers
Output: s, e: floating-point numbers

1: $s = a \oplus b$
2: $t = s \ominus a$
3: $e = (a \ominus (s \ominus t)) \oplus (b \ominus t)$

2.8.1.3 *Priest's Addition EFT*

The two previous algorithms TwoSum and FastTwoSum work perfectly in rounding to nearest. In directed roundings neither

1 the condition $|a| \geqslant |b|$ is easier to check

TwoSum nor FastTwoSum guarantee that the transformation is error-free. Priest's algorithm [48] does not require to round to nearest and transform two given floating-point numbers a, b into s, e such that $a + b = s + e$. However, this algorithm does not always guarantee that $s = a \oplus b$.

Algorithm 2.4 Priest's transformation algorithm (1992)

Input: a, b: floating-point numbers
Output: s, e: floating-point numbers

1: **if** $|a| < |b|$ **then**
2: $\quad (a, b) = (b, a)$
3: **end if**
4: $s = a \oplus b$
5: $w = s \ominus a$
6: $x = s \ominus w$
7: $y = x \ominus a$
8: $z = b \ominus y$
9: $e = z \ominus w$
10: **if** $e \oplus w \neq z$ **then**
11: $\quad s = a$
12: $\quad e = b$
13: **end if**

It also requires that $|a| \geqslant |b|$ so the inputs a and b should be permuted if necessary.

2.8.2 *Error-Free Transformations for Multiplication*

We present here the two algorithms TwoProd and 2MultFMA that perform an error-free transformation for multiplication. Algorithm 2MultFMA is more efficient, yet it requires a FMA floating-point unit which is not available in all processors.

2.8.2.1 *Algorithm* TwoProd

For two floating-point numbers a and b, this algorithm computes floating-point values p and e such that:

$$\begin{aligned} p &= a \otimes b, \\ a \times b &= p + e. \end{aligned} \tag{2.12}$$

Algorithm TwoProd is introduced by Dekker [10]. It relies on Veltkamp's Split algorithm [40].

VELTKAMP'S Split: Let m be the mantissa size, x be a given floating-point number, and s be a positive integer. This algorithm splits x into two floating-point numbers x_h and x_l such that :

- $x = x_h + x_l$;
- the mantissa of x_h fits in $m - s$ bits;
- the mantissa of x_l fits in s bits.

Algorithm 2.5 Veltkamp's Split algorithm (1968)

Input: x: floating-point number, s: positive integer
Output: x_h, x_l: floating-point numbers
1: $a = \beta^s + 1$ {*β is the radix*}
2: $b = x \otimes a$
3: $c = x \ominus b$
4: $x_h = a \oplus c$
5: $x_l = x \ominus x_h$

TwoProd uses Veltkamp's algorithm to split the inputs a and b using $s = \lceil m/2 \rceil$. If m is even then the mantissa of both split results fit in $m/2$ bits. Therefore, the pairwise products of (a_h, a_l) and (b_h, b_l) is performed with no error since their result mantissas fit in m bits. If m is odd then the mantissa of the low part from split results fits in $\lceil m/2 \rceil$ bits. Therefore, the mantissa of $a_l \times b_l$ in general does not fit in m bits.

However, if the radix $\beta = 2$, then the lower part computed by Veltkamp's algorithm fits in $s - 1$ bits [40]. Note that if $s = \lceil m/2 \rceil$ and m is odd then $\lfloor m/2 \rfloor = s - 1$. Since the mantissas of all split results fit in $\lfloor m/2 \rfloor$ bits, pairwise products of (a_h, a_l) and (b_h, b_l) can be performed exactly.

Algorithm 2.6 Algorithm $TwoProd$

Input: a, b: floating-point numbers
Output: p, e: floating-point numbers
1: $(a_h, a_l) = Split(a, \lceil m/2 \rceil)$
2: $(b_h, b_l) = Split(b, \lceil m/2 \rceil)$
3: $p = a \otimes b$
4: $p_{hh} = a_h \otimes b_h$ {*Exact multiplication*}
5: $p_{hl} = a_h \otimes b_l$ {*Exact multiplication*}
6: $p_{lh} = a_l \otimes b_h$ {*Exact multiplication*}
7: $p_{ll} = a_l \otimes b_l$ {*Exact multiplication*}
8: $t_1 = p_{hh} \ominus p$
9: $t_2 = p_{hl} \oplus t_1$
10: $t_3 = p_{lh} \oplus t_2$
11: $e = p_{ll} \oplus t_3$

2.8.2.2 *Algorithm* $2MultFMA$

The error-free transformation of two floating-point numbers product is easier when a FMA instruction is available.

THE FMA (FUSED MULTIPLY-ADD) INSTRUCTION: The FMA instruction evaluates the expression $a \times b + c$ with one final rounding. In other words, given the floating-point numbers a, b and c,

$$FMA(a, b, c) = round(a \times b + c).$$

The numerical results of the FMA instruction can be different from the results of two separate "addition then multiplication". On one side, the FMA performs a single rounding, while two separate operations generate two roundings. Therefore, replacing a sequence of a separate floating-point "multiplication then addition" by a FMA instruction could lead to different numerical results.

The FMA instruction is available on almost all recent CPUs, e.g. all Intel CPUs based on Haswell (released in 2013) or more recent microarchitecture support it. It is also included in the IEEE-754 standard since the 2008 revision [27]. Thus, it is gainful to consider FMA when designing algorithms. More details

about the FMA instruction are given in Section 2.10.3.

Algorithm 2.7 Algorithm 2MultFMA

Input: a, b: floating-point numbers
Output: p, e: floating-point numbers
1: $p = a \otimes b$
2: $e = \mathrm{FMA}(a, b, -p)$

Algorithm 2MultFMA uses the FMA instruction to reduce the number of floating-point operations required to perform an error-free transformation of the product. Only 3 operations are required or 2 operations if we consider FMA as a single operation. Algorithm 2MultFMA is much more efficient compared to Dekker's TwoProd that requires 17 floating-point operations.

Algorithm 2MultFMA consists in two instructions. The first instruction computes p, and the second one computes e as the difference between exact $a \times b$ and computed $a \otimes b$ (stored in p). If no overflow nor underflow occurs, e is calculated exactly such that the error-free transformation verifies Relation (2.12).

2.9 EXTRA PRECISION

Currently, the binary128 format is not implemented on mainstream CPUs, but some software tweaks are introduced to implement higher precisions than binary64.

2.9.1 *Double-Double Arithmetic*

Double-double arithmetic aims at doubling the precision offered by the standard binary64 format. A double-double number x is a non-evaluated sum of two standard binary64 numbers x_h and x_l such that:

$$\begin{aligned} x_h &= x_h \oplus x_l \\ x &= x_h + x_l. \end{aligned} \tag{2.13}$$

Basic double-double arithmetic operations $+, -, \times$ and $/$ are implemented in software relying on standard binary64 arith-

metic. The library XBLAS Li et al. [37] (available at [1]) uses the double-double format to provide a more accurate BLAS implementation.

2.9.2 *Floating-Point Expansions*

Expansions store a number of an arbitrary precision as a non-evaluated sum of a floating-point array. The double-double format presented in the previous section can be considered as expansion of size 2. The expansion size can be set to an integer n to simulate n times the machine precision. However, the larger the expansion is, the more resource intensive it will be. One should find a compromise between accuracy and cost depending on application requirements.

Algorithms for addition, multiplication and division of two expansions are proposed in the works of Priest and Shewchuk [47, 48, 57].

2.9.3 *Superaccumulators*

Superaccumulators are introduced to accumulate floating-point numbers without generating any error. A superaccumulator is a fixed-point number that is long enough to cover the range of all possible inputs. It is mainly composed of three parts, a sign, an integer part and a fractional part (a status part for exceptions and special values can also be considered [35]). U. Kulisch and V. Snyder in [35] suggested a superaccumulator of size 4288 bits to perform a correctly rounded dot product of two binary64 vectors (with size $\leqslant 2^{88}$). Superaccumulators are also used in [9] to perform a correctly rounded accumulation.

2.10 HARDWARE IMPLEMENTATION AND PERFORMANCE

We present in this section some basic algorithms for performing elementary operations. Latency and throughput of operations are also presented. We focus only on Intel processor perfor-

mance because all our performance tests have been performed on Intel architectures.

2.10.1 *Floating-Point Addition*

According to [43], a basic floating-point addition (or subtraction) requires the following steps

1. Compute the difference between input exponents.
2. Shift the mantissa of the input with smaller exponent to the right to align the bits with the same weight.
3. Perform an integer addition on the mantissas.
4. If the integer result is negative, it is converted from two's complement to a sign and mantissa.
5. In the case of cancellation, the first leading bit is shifted to the left to keep a single bit before the point.
6. Normalization of the mantissa, and exponent update.
7. Round the mantissa.

This is not the algorithm used on today's hardware due to its high latency. Improvements can be made to improve this latter [43]. However it is presented for the sake of explanation. The latency of a floating-point addition on current Intel CPUs is 3 or 4 cycles depending on the microarchitecture [30].

2.10.2 *Floating-Point Multiplication*

The floating-point multiplication is simpler than addition, it consists in [39]:

1. Exclusive or (xor) between the signs.
2. Integer addition of the exponents.
3. Integer multiplication for the mantissas.

Note that a denormalization of the mantissa before the multiplication and a result normalization are required. Despite being simple, the latency of the floating-point multiplication is usually higher than addition due to the high latency of integer multiplication. On current Intel CPUs the latency of multiplication is 4 or 5 cycles depending on the microarchitecture [30].

2.10.3 *Fused Multiply and Add*

FMA occurs frequently in a wide range of numerical applications, especially when linear algebra operations are used. Therefore, a hardware implementation of a single instruction that performs a multiply-add operation is very important for performance. As explained in Section 2.8.2.2 a single rounding operation is performed here and the result for the FMA could be different from separated multiplication and addition. An efficient algorithm to perform a FMA is presented in [4]. In practice addition and multiplication steps can overlap with each other. Consequently, a FMA operation when it is implemented in hardware costs the same as a single multiplication. FMA is available on Intel CPUs starting from Haswell architecture [29] released in 2013. Its latency on Intel CPUs is 4 or 5 cycles depending on the architecture. FMA operation is also available on the Intel Xeon Phi accelerator which is important for our experimentation results in chapters 4, 5 and 6.

2.10.4 *Pipelining*

A floating-point operation usually requires several stages and takes more than one cycle. The latency of a floating-point operation varies from 3 cycles (for addition) up to more than 14 cycles (for division) [30]. Pipelining increases the throughput of the floating-point unit. Therefore even if a floating-point operation requires more than one cycle, a new operation is started before ending the previous one. Pipelining ensures that the throughput of a floating-point unit is (significantly) less than its latency. The latency and throughput of floating-point operations on recent

Intel microarchitectures are summarized in Table 2.3.[2]

Archi	Sandy Bridge		Haswell		Skylake	
	lat	thr	lat	thr	lat	thr
Add	3	1	3	1	4	1
Mult	5	1	5	1	4	1
FMA	N/A	N/A	5	1	4	1
Div	15-22	14-22	14-20	13	14	4
SQRT	21-22	14-22	20	13	18	6

Table 2.3: Latency and throughput of floating-point operations [30]

Note that pipelined operations should not depend on each other. For example the addition operations in Algorithm 2.1 can not be pipelined, since $\widehat{S}$ is in the same time one input for the addition in iteration i and the output of the addition in the iteration $i-1$. It is up to the developer's responsibility to implement a program that benefits well from the pipelining to make the floating-point units operating at a consistent throughput.

A possible solution to benefit from pipelining is to use different accumulators inside the loop so the accumulations to different variables do not depend on each other. Algorithm 2.8 is a pipelined version of Algorithm 2.1.

The four floating-point operations inside the loop in Algorithm 2.8 are independent and can be pipelined. Theoretically this algorithm should be about 4 times faster than Algorithm 2.1. However in practice there are some other bottlenecks that reduce these benefits.

2.10.5 *SIMD Instructions*

SIMD stands for Single Instruction on Multiple Data. SIMD instructions apply the same operation on a vector of data simultaneously instead of dealing with individual scalars. At the architectural level, large registers are used to store a vector of data

2 We present the throughput for a single floating-point unit, some architectures have multiple units that execute the same operation

Algorithm 2.8 Pipelined summation algorithm

Input: p: a vector of n floating-point numbers
Output: $\widehat{S}$: the evaluated sum of p_i

1: $\widehat{S}_1 = 0$
2: $\widehat{S}_2 = 0$
3: $\widehat{S}_3 = 0$
4: $\widehat{S}_4 = 0$
5: $n4 = n - (n \bmod 4)$ {*The largest 4 multiple $\leqslant n$*}
6: **for** $(i = 1 : 4 : n4)$ **do**
7: $\quad \widehat{S}_1 = \widehat{S}_1 \oplus p_i$
8: $\quad \widehat{S}_2 = \widehat{S}_2 \oplus p_{i+1}$
9: $\quad \widehat{S}_3 = \widehat{S}_3 \oplus p_{i+2}$
10: $\quad \widehat{S}_4 = \widehat{S}_4 \oplus p_{i+3}$
11: **end for**
12: {*Remaining elements*}
13: **for** $(j = n4 + 1 : n)$ **do**
14: $\quad \widehat{S}_1 = \widehat{S}_1 \oplus p_j$
15: **end for**
16: $\widehat{S} = \widehat{S}_1 \oplus \widehat{S}_2 \oplus \widehat{S}_3 \oplus \widehat{S}_4$

and apply a SIMD instruction to this vector. At the time we are writing this document, the maximum size of registers on CPUs is 256 bits, and the size of registers for Intel Xeon Phi accelerator is 512 bits.

Wide registers can be used as vectors of standard binary64 floating-point numbers. Using SIMD instructions or vectorization provides a significant boost to the throughput of floating-point units. However, some algorithms can not be vectorized easily and/or efficiently due to various factors.

1. Data dependency: we can not vectorize operations that depend on each other. For example, operations in Algorithm 2.8 can be vectorized, while dependency prevent vectorization in Algorithm 2.1.

2. Non contiguous data: the cost of gathering data from different memory regions in a single register is very high and is not always compensated by vector operations. these latter

are useful only if a lot of vector operations are performed using the gathered data.

3. Restrictions on data alignment: for example if the size of the vector to load into register is 32 bytes, the data to be read should be at a memory address that is multiple of 32. Otherwise, multiple instructions should be used to read different chunks of data and gather them in a single register which takes more time than a single instruction.

4. Vectorization is not always an automatic process and sometimes requires a huge and time consuming development effort.

2.11 CONCLUSION

We have addressed in this chapter some of the most important notions and properties of floating-point arithmetic. The IEEE-754 standard specification and some hardware implementation details are presented. Floating-point operations and rounding errors are addressed for elementary operations and for vector summation. We have tackled also some advanced concepts as error-free transformations and extra precision.
The notions presented in this chapter will be used throughout this document to design and analyze our algorithms for reproducible and accurate BLAS and to demonstrate their numerical properties.

3

ACCURACY AND NUMERICAL REPRODUCIBILITY

3.1 INTRODUCTION

Clock speed on recent CPUs is hitting an energy and heat wall and can no more be increased efficiently. Therefore, performing parallel computations is unavoidable to increase compute power. The computational power of today's computers allows us to solve more complex problems that manipulate huge amounts of data in parallel. The floating-point performance increases exponentially, which also increases the amount of round-errors generated by these operations. Ensuring accurate and reproducible results is becoming more and more challenging. We define numerical reproducibility problem as getting identical numerical results from multiple runs of the same program on the same data independently from the number of used threads and the machine architecture. Reproducibility raises due to the non-associativity of floating-point addition and to the way how parallel systems and architectural differences change the order and the semantic of floating-point operations. In this chapter we focus on accuracy and reproducibility problems for summation and BLAS. Several algorithms that attempt to solve those issues are presented here.

3.2 NON REPRODUCIBILITY OF THE SUMMATION

Due to rounding errors, the floating-point addition is not associative. Performing the floating-point operations in different orders could generate different rounding errors. Therefore, for three given floating-point numbers a, b and c, $a \oplus (b \oplus c)$ might be different from $(a \oplus b) \oplus c$. As an example, for machine precision u we have $u = (-1 \oplus 1) \oplus u \neq -1 \oplus (1 \oplus u) = 0$, since $(1 \oplus u) = 1$.

Because of the non-associativity of floating-point addition, the result of a summation does not only depend on input data but

also on the operation order. For instance Algorithms 2.1 and 2.8 could provide different results despite being mathematically equivalent. In practice, accumulation order can change due to change in architectural details as the register size for example. In parallel environments the operation order can change due to dynamic scheduling and non-deterministic reductions as well as a changing number of threads.

The summation of floating-point numbers has already been introduced in Section 2.7. We recall that for a given vector $p = (p_1, p_2, ..., p_n)$ we define the sum S as:

$$S = \sum_{i=1}^{n} p_i = p_1 + p_2 + ... + p_n,$$

while a floating-point summation $\widehat{S}$ is computed (without taking into account the accumulation order for the moment) as

$$\widehat{S} = p_1 \oplus p_2 \oplus ... \oplus p_n.$$

As it can be noted in Algorithm 2.1, we accumulate the elements of p into one global result, therefore those operations are not independent. This summation can be parallelized by using a different accumulator for each thread. A partial sum is computed for every thread, then a reduction operation is performed. Since the floating-point addition is not associative, it is not surprising to obtain different numerical results when we parallelize the summation in this way. Non-associativity of addition is the main reason of non-reproducibility either in summation or in other operations as in BLAS. Figures 9 and 11 in [5] show that the numerical reproducibility of summation in parallel environments depends mainly on the number of used threads and on the condition number of the problem.

3.3 REPRODUCIBILITY VS ACCURACY

Reproducibility is different from accuracy. Accuracy is measured by the relative error of the numerical result, or in other terms by how many correct bits (or digits) appear in the numerical result

compared to the (usually unknown) exact result. On the other hand, reproducibility is defined as obtaining identical numerical results from one run to another if the input data does not change which means that the relative difference between those results should be 0. Nevertheless, this reproducible result can be either accurate or not, depending on the used algorithm. We can go up to imagine a reproducible result with no correct digit compared to the exact result if the problem to solve is too ill-conditioned.

One way to benefit from reproducible and accurate result is to ensure that this latter is correctly rounded. Note that the correctly rounded result as we define it in Section 2.3 is unique, and in the rounding to nearest mode it also ensures the minimal possible relative error. However, as computing a rounded to nearest result require more computations, it might introduce an important extra-cost to the running time. We consider this issue along analysis proposed in Chapters 4, 5 and 6.

3.4 ACCURATE SUMMATION ALGORITHMS

Many algorithms have been introduced to improve the accuracy of floating-point summation (see references mentioned in [40, 24]). Some of them ensure smaller relative errors compared to the classic algorithm, while others go up to ensure that the result is faithfully or even correctly rounded. The most recent algorithms of each category are presented in this section.

3.4.1 *Distillation Algorithms*

Distillation is an algorithm that transforms an input vector p^0 iteratively to a vector p^k such that $\sum_{i=1}^{n} p_i^0 = \sum_{i=1}^{n} p_i^k$, and for each k, $p_n^k = \sum_{i=1}^{n} p_i^{k-1}$. This process relies on performing summation while replacing the floating-point addition by the algorithm TwoSum. Distillation is usually used to improve summation accuracy or to perform an error-free transformation on a vector of floating-point numbers. A wide range of summation algorithms relies on distillation. We present here SumK and

iFastSum. Other varieties of distillation algorithms have been presented in [71].

3.4.2 *SumK*

This algorithm has been introduced in [44]. It relies mainly on distillation. Ogita and al. identified Algorithm *VecSum* which simply consists in one distillation step. Note that Algorithm 3.1 (*VecSum*) introduces a different vector for the output. However in practice the input vector can be overwritten to save memory.

Algorithm 3.1 Algorithm *VecSum*

Input: p: a $n-$vector of floating-point numbers;
Output: q: an error-free transformation of vector p;

1: $q_1 = p_1$
2: **for** $(i = 2; i \leqslant n; i = i + 1)$ **do**
3: $\quad (q_i, q_{i-1}) = TwoSum(p_i, q_{i-1})$;
4: **end for**

Algorithm 3.1 guarantees that

$$\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i,$$
$$q_n = ((...(p_1 \oplus p_2) \oplus ...) \oplus p_n).$$

If the sum of vector p is ill-conditioned there should be a lot of cancellations during this process. Therefore $\sum_{i=1}^{n} |q_i| \ll \sum_{i=1}^{n} |p_i|$ while $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i$. The authors prove in [44] that if $cond(\sum_{i=1}^{n} p_i) > u^{-1}$, we have:

$$\sum_{i=1}^{n} |q_i| \approx u \cdot \sum_{i=1}^{n} |p_i| .$$

Therefore, computing the sum of the vector q is a significantly less ill-conditioned problem than computing the sum of the vector p, since:

$$cond\left(\sum_{i=1}^{n} q_i\right) = \frac{\sum_{i=1}^{n} |q_i|}{|\sum_{i=1}^{n} q_i|} \approx \frac{u \cdot \sum_{i=1}^{n} |p_i|}{|\sum_{i=1}^{n} p_i|} \approx u \cdot cond\left(\sum_{i=1}^{n} p_i\right).$$

Algorithm SumK uses repeatedly algorithm VecSum to enhance the accuracy of the summation result. Algorithm 3.2 details this process.

Algorithm 3.2 Algorithm SumK

Input: p: a $n-$vector of floating-point numbers
K: the number of iterations
Output: S_k: sum of p_i which is almost as accurate as if computed in K times the working precision

```
1: for (k = 1; k ⩽ K − 1; k = k + 1) do
2:    p = vecSum(p)
3: end for
4: S_k = 0
5: for (i = 1; i ⩽ n; i = i + 1) do
6:    S_k = S_k ⊕ p_i
7: end for
```

The authors proved that for $K = 2$ the relative error is bounded as:

$$\frac{|S_k - S|}{|S|} \leqslant u + \gamma_{n-1}^2 \cdot cond\left(\sum_{i=1}^{n} p_i\right), \tag{3.1}$$

and for $K \geqslant 3$ the relative error is proven to be improved as:

$$\frac{|S_k - S|}{|S|} \leqslant u + 3 \cdot \gamma_{n-1}^2 + \gamma_{2n-2}^K \cdot cond\left(\sum_{i=1}^{n} p_i\right). \tag{3.2}$$

The condition number effect in the last term of Relation (3.2) can be vanished by choosing a large enough K. However the condition number should be known, yet, it is usually not the case in practice.

3.4.3 *iFastSum*

Algorithm iFastSum has been introduced by Zhu and Hayes in [69]. Its principle is not different from SumK. iFastSum adapts the number K of iterations to the condition number of the problem without computing it, and repeats the distillation process

until the final result is correctly rounded.

Relation (3.2) shows that regardless of the value of K we can not ensure that the relative error is smaller than $u + 3 \cdot \gamma_{n-1}^2$. Therefore increasing the value of K is not enough to ensure a correctly rounded result. Algorithm 3.3 presents a simplified version of iFastSum. This latter defines a dynamic error control during the distillation process to estimate an error bound and decide if the result is correctly rounded. Properties of iFastSum are demonstrated in [69]. Although, we give next a brief description of this algorithm due to its significance for this work.

Lines 6-12 define the main distillation loop. Line 9 test ensures that only non-zero errors are kept for the next iterations, and the variable count counts those errors. The input vector p is overwritten by the generated errors. Considering that the exact sum is S, we know that at the end of this loop we have $S = \widehat{S} + S_t + \sum_{i=0}^{count-1} p_i$.

Since for each intermediate evaluation of S_t the generated error is smaller than $ulp(S_t)/2$ and S_{max} is the maximum intermediate value of the sum of generated errors, $\sum_{i=1}^{count-1} p_i$ is bounded by $(count - 1) \times ulp(S_{max})/2$. If the condition in line 17 is verified, $\widehat{S}$ is guaranteed to be a faithfully rounded value of S. The condition in line 18 tests if recursive calls are allowed. Note that if it is not the case (*allowRec* is set to *false*), iFastSum returns a faithfully rounded result. Otherwise, lines 21-26 ensure that the final result is correctly rounded. At this level we know that $\widehat{S} + S_t - maxError \leqslant S \leqslant \widehat{S} + S_t + maxError$. After lines 22 and 23 we have $\widehat{S} + S_- + e_- \leqslant S \leqslant \widehat{S} + S^+ + e^+$. Afterwards, the function round3 (presented in Figure 2.3 of [69]) is used to round $\widehat{S} + S^+ + e^+$ and $\widehat{S} + S_- + e_-$. If both values round to $\widehat{S}$, this latter is ensured to be the correctly rounded result. Otherwise, S_1 and S_2 are computed such that S_1 is a faithfully rounded result of $S - \widehat{S}$, and S_2 is a faithfully rounded result of $S - \widehat{S} - S_1$. Finally, the function round3 is used to compute a correctly rounded result for S. Since $\widehat{S}$ is a faithfully rounded value of S we know that $S_1 < ulp(\widehat{S})$. In the same way we deduce that $S_2 < ulp(S_1)$. We distinguish two possibilities here:

Algorithm 3.3 Algorithm iFastSum

Input: p: a n—vector of floating-point numbers (overwritten during the process)
$allowRec$: a boolean that indicates if recursive calls are allowed (default value is $true$)
Output: $\widehat{S}$: $roundToNearest(\sum_{i=1}^{n} p_i)$

1: $\widehat{S} = 0$
2: **while** true **do** {*Return instruction stops the loop*}
3: $count = 1$ {*Counts non-zero errors*}
4: $S_t = 0$ {*The sum of this iteration*}
5: $S_{max} = 0$ {*The max intermediate value of S_t*}
6: **for** $(i = 1; i \leqslant n; i = i + 1)$ **do**
7: $(S_t, p_{count}) = TwoSum(S_t, p_i)$
8: $S_{max} = max(S_{max}, |S_t|)$
9: **if** $p_{count} \neq 0$ **then**
10: $count = count + 1$
11: **end if**
12: **end for**
13: $maxError = (count - 1) \times ulp(S_{max})/2$
14: $(\widehat{S}, S_t) = TwoSum(\widehat{S}, S_t)$
15: $n = count$ {*Work on non-zero errors only*}
16: $p_{count} = S_t$ {*store the error in p*}
17: **if** $maxError < ulp(\widehat{S})/2$ **then**
18: **if** not $allowRec$ **then** {*Only one level of recursion*}
19: **return** $\widehat{S}$
20: **end if**
21: $(S^+, e^+) = TwoSum(S_t, maxError)$
22: $(S_-, e_-) = TwoSum(S_t, -maxError)$
23: **if** $(\widehat{S} \oplus S^+ \neq \widehat{S})$ or $(\widehat{S} \oplus S_- \neq \widehat{S})$ or $(round3(\widehat{S}, S^+, e^+) \neq \widehat{S})$ or $(round3(\widehat{S}, S_-, e_-) \neq \widehat{S})$ **then**
24: $S_1 = iFastSum(p, false)$ {*p is updated by this call*}
25: $S_2 = iFastSum(p, false)$
26: $\widehat{S} = round3(\widehat{S}, S_1, S_2)$
27: **end if**
28: **return** $\widehat{S}$
29: **end if**
30: **end while**

1. $|S_1| = \mathrm{ulp}(\widehat{S})/2$, in this case only the sign of S_2 is needed. If $S_2 > 0$, rounding up $\widehat{S} + S_1$ returns the correctly rounded result of S; if $S_2 < 0$, rounding down $\widehat{S} + S_1$ returns the correctly rounded result of S. If $S_2 = 0$, we deduce that S is the exact midpoint of two floating-point numbers, in this case the tie breaking rule in rounding to nearest guarantees that $\widehat{S} \oplus S_1$ is the correctly rounded value of S.
2. $|S_1| \neq \mathrm{ulp}(\widehat{S})/2$, in this case $\widehat{S} \oplus S_1$ provides the correctly rounded value of S.

Some steps have been simplified in our description. However iFastSum is presented in more details with formal demonstrations in [69].

3.4.4 *AccSum*

This algorithm has been presented by Rump and al. in [54]. AccSum ensures that the result is faithfully rounded. It also relies on error-free transformation, but not the same as SumK and iFastSum. For a given vector p of size n the algorithm extracts only leading parts of the elements of p as shown in Figure 3.2.

Algorithm ExtractScalar is used to extract leading parts of p_i. Given σ a power of 2 and $\sigma > |p_i|$, Algorithm 3.4 (ExtractScalar) splits the input p_i into q and p' such that:

$$\begin{aligned} \mathrm{lnb}(q) &\geqslant \mathrm{ulp}(\sigma), \\ q &\leqslant 2 \cdot \mathrm{ufp}(p_i), \\ \mathrm{ufp}(p') &< \mathrm{ulp}(\sigma), \\ q + p' &= p_i. \end{aligned}$$

As for TwoSum and FastTwoSum, algorithm ExtractScalar ensures that the transformation is error-free only when a rounding to nearest mode is used. Therefore this rounding mode is required for AccSum to work properly. A visual illustration of algorithm ExtractScalar is shown in Figure 3.1.

The value of σ used in AccSum depends on:

Algorithm 3.4 Algorithm ExtractScalar

Input: σ, p_i: two floating-point numbers
Output: q, p': two floating-point numbers
1: $q = (p_i \oplus \sigma) \ominus \sigma$
2: $p' = p_i \ominus q$

1. $max(|p_i|)$, the maximum of absolute values of vector p, and
2. n, the vector size.

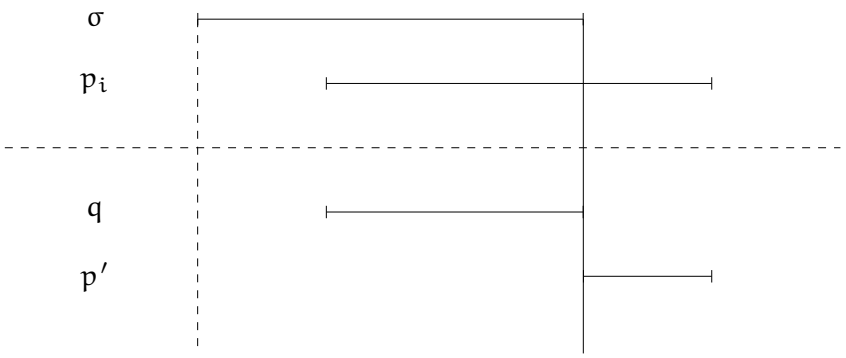


Figure 3.1: Input and output of the ExtractSclalar

Figure 3.2 shows that leading parts of all elements of p are extracted using the same σ_1. In this case the leading parts can be accumulated with no rounding error. In the next iteration σ_1 is scaled into σ_2 to extract the leading parts of the result p'. Using a σ_i to split the elements of p guarantees that all the elements of p' are smaller than $ulp(\sigma_i)$. Therefore it is possible to estimate the maximum value of $\sum p'_i$ and investigate its influence on the calculated sum to define a stopping condition. AccSum ensures that the computed sum is faithfully rounded in a very similar way to iFastSum without introducing advanced tests to ensure correctly rounded result. Algorithm NearSum [55] relies on the same process as AccSum and introduces a similar test to iFastSum to ensure correctly rounded results.

3.4.5 *FastAccSum*

This algorithm is very similar to AccSum. It improves that latter by relying on FastTwoSum instead of ExtractScalar to reduce the number of performed floating-point operations by 25%.

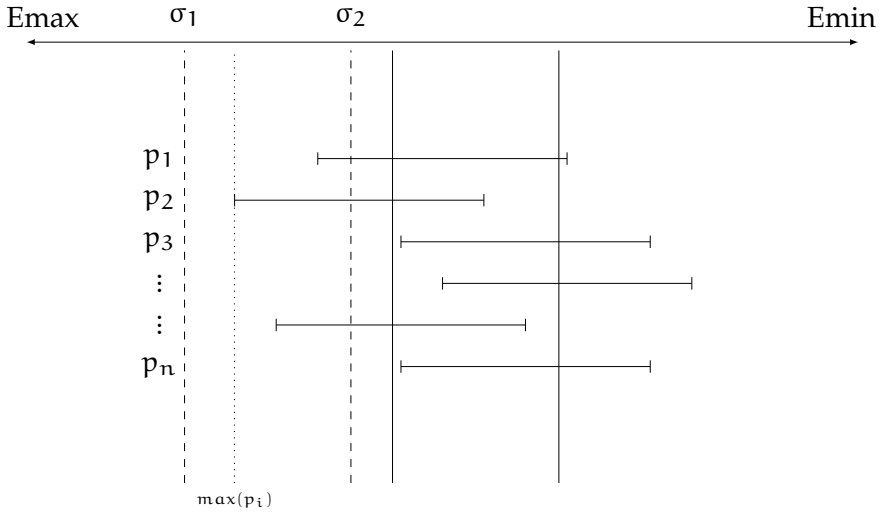


Figure 3.2: Algorithm AccSum

FastAccSum [52] accumulates the leading parts to σ using FastTwoSum instead of extracting them. Note that σ is not computed in the same way as for AccSum because it should be chosen such that all its intermediate values during the accumulation remain larger than p_i. Afterwards, the initial value of σ is subtracted from the final value to yield the sum of accumulated leading parts. As for AccSum this process is repeated to refine the result until it is faithfully rounded.

3.4.6 *HybridSum*

The major drawback of the previously presented algorithms is that they run multiple times through the input data to refine the summation result and provide a correctly or faithfully rounded result. More refinements are required when the problem is ill-conditioned. HybridSum [69] overcomes this problem, i.e., it provides a similar process independently from the condition number. It accumulates with extra precision the entries with the same exponent in a same accumulator.

Higher precision accumulators are not implemented in hardware. HybridSum splits the input mantissa into two parts so the standard floating-point numbers are considered as accumulators. Algorithms 2.5 and 3.5 can be used to split a floating-point number into 2 floating-point numbers with a half length mantissa.

Algorithm 3.5 Split algorithm

Input: $x, mask$: floating-point numbers
Output: x_h, x_l: floating-point numbers
1: $x_h = binaryAnd(x, mask)$ {*put the last 26 bits in the mantissa to 0*}
2: $x_l = x \ominus x_h$

Note that we need a distinct accumulator for every possible exponent: for instance 2048 accumulators for binary64 format. This process reduces the size of input data to a 2048-vector without generating any roundoff error. This is very interesting for large vector entries because it error-free transforms the input to a small vector at a low cost. Afterwards, an iterative summation algorithm can be run on this small vector at small cost. The authors used iFastSum [69] to compute a correctly rounded result. However any summation algorithm can be used. Algorithm 3.6 details how HybridSum transforms the input data into a 2048-vector.

Algorithm 3.6 HybridSum transformation

Input: p: a vector of n floating-point numbers
Output: C: a vector of 2048 floating-point numbers
1: **for** $(i = 1; i \leqslant n; i = i + 1)$ **do**
2: $\quad (p_{i,h}, p_{i,l}) = split(p_i)$
3: $\quad e_h = exponent(p_{i,h})$
4: $\quad e_l = exponent(p_{i,l})$
5: $\quad C_{e_h} = C_{e_h} \oplus p_{i,h}$
6: $\quad C_{e_l} = C_{e_l} \oplus p_{i,l}$
7: **end for**

3.4.7 *OnlineExact*

The principle of OnlineExact [70] is the same as HybridSum. It error-free transforms the input data to a small vector by accumulating the elements with the same exponent in distinct accumulators. Although, OnlineExact uses two standard floating-point numbers (or expansions of size 2) to implement high precision

accumulators. TwoSum is used to add a standard floating-point number to the first part of the accumulator. The error calculated by TwoSum is added to the second part of the accumulator. Therefore, OnlineExact requires two vectors of 2048 elements each to ensure that the transformation is error-free. We exhibit in chapter 4 that algorithm FastTwoSum can be used in this case instead of TwoSum without requiring a particular order for inputs. The process of OnlineExact transformation is detailed in Algorithm 3.7.

Algorithm 3.7 OnlineExact transformation

Input: p: a vector of n floating-point numbers
Output: Ch, Cl: two vectors of 2048 floating-point numbers

1: **for** $(i = 1; i \leqslant n; i = i + 1)$ **do**
2: $\quad e = exponent(p_i)$
3: $\quad (Ch_e, error) = TwoSum(Ch_e, p_i)$
4: $\quad Cl_e = Cl_e + error$
5: **end for**

3.4.8 *How to Choose the Right Summation Algorithm?*

We have presented a wide range of summation algorithms which have different properties. The choice of the used algorithm depends on application needs or the nature of the problem that we deal with. According to numerical properties, we distinguish 3 sets of summation algorithms.

ALGORITHMS THAT IMPROVE ACCURACY like SumK presented in Section 3.4.2 or PrecSum presented in [55]. Algorithms in this category usually provide a result as if computed in higher precision. They aim at improving the accuracy in cases where the machine precision is not enough to ensure application stability. Nevertheless, the accuracy of the result still suffer from condition number effects.

FAITHFULLY ROUNDED SUMMATION are useful for applications that deal with very ill conditioned problems. Algorithms

in this category also rely on improving accuracy. We mention for example AccSum [54] and FastAccSum [52] presented in Sections 3.4.4 and 3.4.5 respectively. The authors of algorithms in this category demonstrate that the error is smaller than the result's ulp. Both algorithms have input size limitations. For a given machine precision u, AccSum works for vectors with size $n < \sqrt{u^{-1}}$, while FastAccSum require $n < u^{-1}$.

CORRECTLY ROUNDED SUMMATION We focus in this document mainly on iFastSum. According to its parameters, it provides either faithfully or correctly rounded result. Algorithm iFastSum has been presented in more details in Section 3.4.3. OnlineExact and HybridSum also ensure correctly rounded result, but both of them rely on iFastSum.

3.5 SOLUTIONS FOR REPRODUCIBLE SUMMATION

Solutions for reproducible summation that do not rely on accuracy improvement are presented in this section. We start with algorithms ReprodSum, FastReprodSum and 1-Reduction . Then we present the solutions provided by the few existing parallel programming libraries.

3.5.1 *ReprodSum*

ReprodSum [11] relies on AccSum presented in the previous section. Algorithm 3.4 (ExtractScalar) is used to extract the high order parts of the input vector. As the sum of the leading parts is exact, it is independent from the accumulation order. Therefore, the parallel and sequential versions of the algorithm provide the same result. While AccSum repeats the process until the result is faithfully rounded, ReprodSum takes the number of iterations K as a parameter. As for SumK, increasing the value of K increases the accuracy of the result. If the condition number of the sum is known, we should select a K that is large enough to overcome huge cancellations or even to ensure a faithfully rounded results. On the other side decreasing the value of K decreases the accuracy, but increases the performance.

3.5.2 *FastReprodSum*

FastReprodSum [11] relies on FastAccSum. It improves the ReprodSum algorithm by also performing 25% less floating-point operations. FastReprodSum uses FastTwoSum to extract the leading parts of the input vector instead of ExtractScalar for ReprodSum. In this case the used σ is selected such that all its intermediate values are larger than p_i. An additional condition is that $ufp(\sigma)$ does not change during the process to ensure that the same number of bits is extracted from p_i independently from the order. Another problem raises in this case. Since the value of σ changes during the process, if rounding to nearest mode is used and the result is a midpoint, i.e., the middle of two floating-point numbers, it would be rounded differently from one run to another depending on $ulp(\sigma)$. Therefore to guarantee that the same leading part is extracted, a directed rounding should be used. However the transformation is no more error-free since FastTwoSum works only in rounding to nearest. Unfortunately here we can not increase the value of K to obtain a faithfully rounded result. Nevertheless, it is not the aim of this algorithm.

Note that both algorithms ReprodSum and FastReprodSum also need to run through the input data at least twice. The first run calculates the maximum absolute value of the input data and defines σ that is used to extract the high leading parts of p_i. The second run extracts the high leading parts and compute the reproducible sum. Also in the parallel case, two communications are required. This property is considered as a huge handicap for both algorithms ReprodSum and FastReprodSum, especially on distributed memory parallel systems where communication cost is critical.

3.5.3 *Indexed Floating-Point Numbers*

Indexed floating-point is a data structure that has been introduced in [12] and presented with more details in Section 5 of [14]. An indexed number consists in K floating-point accumulators. As for floating-point expansions presented in Section 2.9.2, a trade off between accuracy and cost should be found: larger K

ensures more accuracy for indexed numbers while it amplifies their manipulation cost. The accumulation of n floating-point numbers to an indexed number is ensured to be reproducible independently from the operation order.

The principle of indexed numbers is similar to ReprodSum. It breaks up the summation input into slices according to predefined exponent intervals, and accumulates exactly the slices from the same interval in the corresponding accumulator. During the accumulation process, the maximum (the first) accumulator aligns to the largest absolute value in the input data while slices that require smaller exponent ranges than that allowed by the smallest (the last) accumulator are ignored.

3.5.4 *1-Reduction*

The aim of algorithm 1-Reduction [13] is to overcome the handicap of two reductions within ReprodSum and FastReprodSum. It relies on indexed floating-point numbers [12]. Therefore, it does not require the maximum value of the input vector p to define σ. 1-Reduction uses binning to achieve reproducibility. The inputs p_i are split into slices according to predefined exponent ranges. Only the slices of p_i that are in the same exponent range are accumulated together as shown in Figure 3.3.

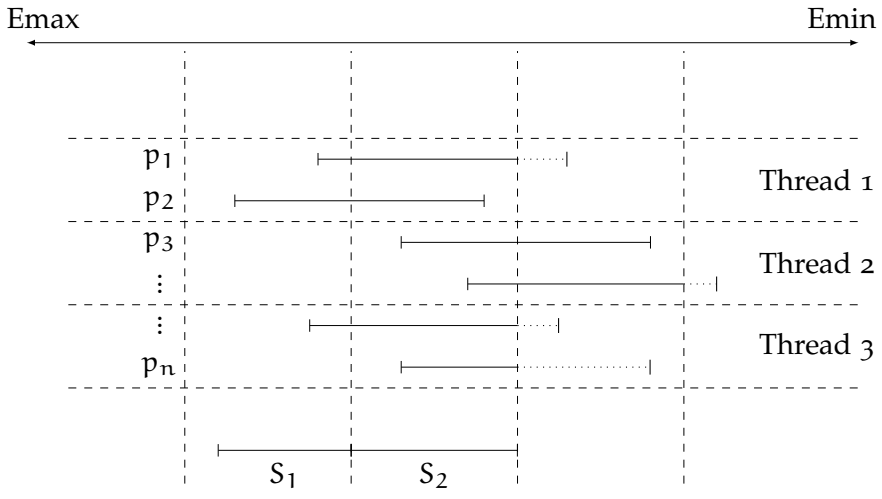


Figure 3.3: Algorithm 1-Reduction (K = 2)

In the parallel case, local data of each thread is accumulated into an indexed floating-point number. Since the exponent range for each bin is predefined, it is shared for all threads. Local indexed numbers are aligned on the right bin according to thread local maximum of absolute values. Afterwards, during the reduction step only the K leading bins are kept.

3.5.5 *Static Scheduling and Deterministic Reduction*

This solution is offered in parallel programming libraries. We focus here on the library OpenMP since we used it for our implementations. Dynamic scheduling is usually used to improve performance by giving more work to the faster thread. Even if threads run on different cores of the same processor, other factors (turbo boost, operating system interruptions, non-uniform memory speed) can make one thread running faster than the others. However, dynamic scheduling changes the accumulation order for summation. Therefore, provided results are not reproducible. The same apply for non-deterministic reductions. For summation when the reduction is not deterministic we accumulate local sums that finished first to the global sum. Therefore even if the data is scheduled in the same way, the reduction tree order can be different from one run to another.

Static scheduling and deterministic reduction are used to ensure that even in parallel the order of accumulation is the same from one run to another. In terms of timing, this solution is less efficient due to the lack of dynamic behavior during runtime. Yet this is not a definitive solution to reproducibility problem. Static scheduling and deterministic reduction ensure reproducibility only if the number of threads do not change from one run to another, and if the program is run on the same processor with the same instruction set (or a different processor that supports the same instruction set).

3.6 REPRODUCIBLE BLAS AND SUMMATION

BLAS functions are ubiquitous in scientific applications. Since most of BLAS routines rely on summation, parallel versions of BLAS are not necessarily reproducible. We present in this section the currently existing solutions to ensure reproducible BLAS.

3.6.1 *Conditional Numerical Reproducibility*

CNR (Conditional Numerical Reproducibility) is a feature that was introduced in release 11.0 of Intel MKL library [33]. CNR solves reproducibility problem when a software that uses MKL is run by two different architectures with different instruction sets. However, it requires the number of threads to be the same from one run to another to ensure reproducible results [65]. This feature is studied in more details in Appendix A.

3.6.2 *ReproBLAS Library*

The library ReproBLAS [42] relies on Indexed floating-point numbers (presented in Section 3.5.3) to provide a reproducible BLAS implementation. Parallel accumulations in BLAS are performed using an index number instead of standard floating-point numbers to be independent from accumulation order. The ReproBLAS library is presented in more details in [14]. According to the library website, and the last downloadable version, only sequential and MPI implementations are provided at this moment. Future versions should include OpenMP implementation.

3.6.3 *ExBLAS library*

ExBLAS [19] is a recent BLAS implementation that aims at ensuring reproducibility by computing correctly rounded outputs. The summation algorithm used in this library relies on superaccumulators and floating-point expansions presented in sections 2.9.3 and 2.9.2 respectively. Note that it costs much less to accumulate a floating-point number into an expansion than to accumulate it into a large superaccumulator. Therefore, expan-

sions are used as first level filtering, and the superaccumulator is accessed only when the accumulation result no more fits in the expansion [9]. The summation process is explained in Figure 3.4.

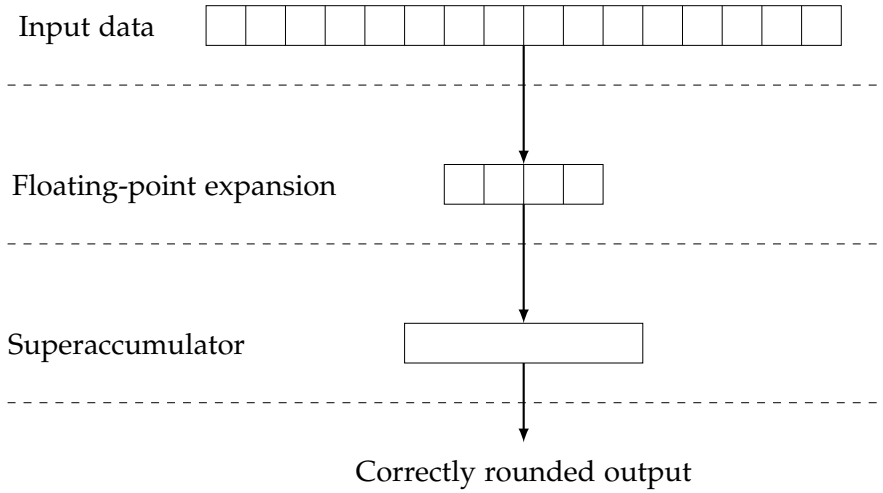


Figure 3.4: The principle of summation used within ExBLAS

Small expansions are efficient when the exponent range of the input data is small. However for large exponent range, the precision offered by small expansions is not enough, the access to the superaccumulator becomes more frequent and the algorithm is less efficient. Large expansions hold their efficiency for large exponent ranges. However, accumulations to larger expansions cost more than for small ones. A trade off on the expansion size should be made according to the input vector exponent range.

3.7 CONCLUSION

In this chapter, we have presented the accuracy limits and the numerical reproducibility problem for summation. Different algorithms have been proposed to overcome those problems. In Section 3.4 we present recent solutions to improve the accuracy of summation or even to compute a correctly or faithfully rounded result, while Section 3.5 studies algorithms and techniques to ensure reproducible parallel summation. Section 3.6 present solutions for reproducibility used in some BLAS implementations. We clearly distinguish accuracy from reproducibility. More accuracy does not always ensure reproducibility while a repro-

ducible result is not necessarily accurate. However, ensuring that the result is correctly rounded always yields reproducibility because this latter is unique. Therefore we will focus next on ensuring correctly rounded results to guarantee both accuracy and reproducibility for summation and BLAS.

Part II

TOWARDS REPRODUCIBLE AND ACCURATELY ROUNDED BLAS

4

PARALLEL REPRODUCIBLE AND CORRECTLY ROUNDED SUMMATION

4.1 INTRODUCTION

In this chapter, we focus only on parallel reproducible summation. We aim at ensuring reproducible summation by computing a correctly rounded result. For the sequential case, we will study a wide selection of available algorithms and choose the most efficient one. For parallel correctly rounded summation we design a new algorithm relying on error-free transformations. Our solution is compared to already available reproducible summation algorithms used in ReproBLAS [42]. We also compare our summation to ExBLAS [19] summation since both algorithms solve the reproducibility problem by computing the correctly rounded sum. The optimized summation provided by Intel MKL library is taken as reference to exhibit the extra-cost of our algorithm. Performance evaluation in this chapter is done for two environments significant of a workstation type and a many-core accelerator. CPU measures are done for environment A (workstation, Table B.1), and accelerator measures are done for environment B (Intel Xeon Phi accelerator, Table B.1). We show that our summation algorithm exhibits almost no extra-cost cost compared to other implementations in the best case, while in the worst one it costs about 6× more than MKL summation which is neither accurate nor reproducible.

4.2 IMPLEMENTATION AND OPTIMIZATION OF SEQUENTIAL SUMMATION

In this section we study algorithms AccSum, FastAccSum, iFastSum, HybridSum and OnlineExact. We start by presenting some implementation details, then we exhibit performance results.

As shown in Table B.1 we prohibit the compiler use of aggressive floating-point optimizations. The option "-fp-model double" is used so all the intermediate values are computed in double precision (80 bits registers are not used). The option "-fp-model strict" disables value-unsafe optimizations. For example, if value-unsafe optimizations are allowed, the compiler could rely on addition associativity and replace for in stance the line 1 instruction in Algorithm 3.4 ($q = (p_i \oplus \sigma) \ominus \sigma$) by "$q = p_i$".

Vectorization and data prefetching have been manually introduced using Intel intrinsic instructions [32]. Loops are manually unrolled for vectorization. However we allow the compiler to introduce more unrolling with the option "-funroll-all-loops".

For the sake of optimization, we have introduced few changes to algorithms HybridSum and OnlineExact. We prove that for HybridSum it is possible to ensure that the transformation is error-free with no need to calculate the exponent of the trailing part of split results. For OnlineExact we demonstrate that FastTwoSum can be used instead of TwoSum to reduce the floating-point operation count. Our implementation of both algorithms is presented hereafter.

4.2.1 *Optimization of HybridSum*

As described in Algorithm 3.6, HybridSum transforms the input vector p to a smaller one by accumulating elements with the same exponent into the same accumulator (lines 5, 6). The input floating-point numbers are split into two parts, each with half the mantissa of standard binary64 numbers (line 2). Both Algorithms 2.5 (Veltkamp) and 3.5 (binary mask) can be used to split the input data. However we prefer Veltkamp's algorithm for two reasons.

1. For binary64 inputs, Algorithm 2.5 guarantees that both outputs (high and low parts) have $\lfloor m/2 \rfloor$ bits mantissa even if m is odd [40]. At the contrary if the Algorithm 3.5 is used, at least one of the two parts could have $\lceil m/2 \rceil$ bits mantissa. Here $m = 53$.

2. Algorithm 2.5 is more efficient in practice[1] since it only relies on floating-point operations. Therefore it neither requires a mix of integer and floating-point registers or units, nor casting operations. At the contrary Algorithm 3.5 uses integer units for mask operation.

Veltkamp's Split algorithm (Algorithm 2.5)

Input: x: floating-point number, s: positive integer
Output: x_h, x_l: floating-point numbers

1: $a = \beta^s + 1$ {*β is the radix*}
2: $b = x \otimes a$
3: $c = x \ominus b$
4: $x_h = a \oplus c$
5: $x_l = x \ominus x_h$

To get the exponent of a floating-point number we perform two operations. We put the sign bit to 0 using a mask operation. Afterwards we shift 52 bits to the right to eliminate the mantissa and keep only the binary presentation of the exponent. Note that we do not have the real value of the exponent because of the `bias` that should be considered as shown in Tables 2.1 and 2.2. However, we only need to distinguish elements with the same exponent, so there is no need to compute the real value here.

We next demonstrate that when we split one input p_i into two parts $p_{i,h}$ and $p_{i,l}$, it is not necessary to know the exponent of $p_{i,l}$ to accumulate it to the right accumulator. The original version of `HybridSum` transformation has been proved to be error-free in Section 3.2 of [69]. However we recall its demonstration in Theorem 4.1 using our Chapter 3 notations before demonstrating the properties of our the modified version.

Theorem 4.1. *Let* p *be a vector of floating-point numbers resulting from Veltkamp's split algorithm applied on* `binary64` *floating-point numbers using* $s = \lceil m/2 \rceil$*, where* m *is the mantissa size for* `binary64`*. Let all elements of* p *have the same exponent* e*,* n *be the size of* p *and* C *be a* `binary64` *floating-point number. We demonstrate that for*

1 According to our experiments on environment A.

$n \leqslant 2^{\lceil m/2 \rceil}$, $C = p_1 \oplus p_2 \oplus ... \oplus p_n$ *is a* binary64 *floating-point number.*

Proof. For a given floating-point number x, Muller and al. have shown in [40, page 135] that for binary formats, $split(x, s)$ yields x_h and x_l such that.

- x_h fits in $m - s$ bits.
- x_l fits in $s - 1$ bits.

Since we use $s = \lceil m/2 \rceil$ and $m = 53$ is odd for binary64 floating-point numbers, $m - s = \lfloor m/2 \rfloor$ and $s - 1 = \lfloor m/2 \rfloor$. Therefore both x_h and x_l fit in $\lfloor m/2 \rfloor$ bits.

Note that all elements of p are generated using Veltkamp's split algorithm, so mantissa of p_i fits in $\lfloor m/2 \rfloor$ bits. For every $p_i \in p$, $exponent(p_i) = e$, so we have:

$$\begin{aligned} |p_i| &< 2^{e+1}, \\ lnb(p_i) &\geqslant 2^{e-\lfloor m/2 \rfloor + 1}. \end{aligned} \tag{4.1}$$

For $C = \sum_{i=1}^{n} p_i$, and according to Relation (4.1),

$$|C| < n \cdot 2^{e+1}.$$

We consider that $n \leqslant 2^{\lceil m/2 \rceil}$, so

$$|C| < 2^{\lceil m/2 \rceil} \cdot 2^{e+1} = 2^{e+\lceil m/2 \rceil + 1}.$$

Therefore, its unit in first place is bounded as:

$$ufp(C) \leqslant 2^{e+\lceil m/2 \rceil}. \tag{4.2}$$

For $C = \sum_{i=1}^{n} p_i$, Relation (2.6) yields

$$lnb(C) \geqslant \min_i(lnb(p_i)). \tag{4.3}$$

So according to Relation (4.1)

$$lnb(C) \geqslant 2^{e-\lfloor m/2 \rfloor + 1}. \tag{4.4}$$

From Relations (4.2) and (4.4), we have:

$$\begin{aligned}
\frac{\mathrm{ufp}(C)}{\mathrm{lnb}(C)} &\leqslant \frac{2^{e+\lceil m/2\rceil}}{2^{e-\lfloor m/2\rfloor+1}},\\
&\leqslant 2^{e+\lceil m/2\rceil-e+\lfloor m/2\rfloor-1},\\
&\leqslant 2^{m-1},\\
&\leqslant \frac{u^{-1}}{2},\\
\text{and} \quad \mathrm{lnb}(C) &\geqslant \mathrm{ufp}(C)\cdot 2u.
\end{aligned}$$

Therefore, according to Relation (2.5), C is a floating-point number.

□

Let $(p_{i,h}, p_{i,l}) = \mathrm{split}(p_i, \lceil m/2\rceil)$. We exhibit that it is sufficient to compute the exponent of the leading part e_h to derive the accumulator corresponding for both leading and trailing parts. Hence, the trailing part $p_{i,l}$ is accumulated to the accumulator $C_{e_h-\lceil m/2\rceil}$ without computing its exponent.

We first prove in Lemma 4.1 an additional property of the Veltkamp split algorithm which will also be useful for our demonstration.

Lemma 4.1. *Let x be a floating-point number in binary format, m be the mantissa size, $0 < s < m$ and $(x_h, x_l) = \mathrm{split}(x, s)$ using Algorithm 2.5 (Veltkamp). We show that $\mathrm{exponent}(x_h) \geqslant \mathrm{exponent}(x)$.*

Proof. As in [40] we only focus on the case where the input $1 \leqslant x < 2$ without loss of generality. Muller and al. [40] has shown that:

$$\begin{aligned}
x &= x_h - x_l,\\
x_l &\leqslant 2^{s-m},\\
x_h &\text{ is a multiple of } 2^{s-m+1}.
\end{aligned} \tag{4.5}$$

Therefore, to ensure that $\mathrm{exponent}(x_h) \geqslant \mathrm{exponent}(x)$ we need in this case to prove that $x_h \geqslant 1$.

We prove by contradiction that $x_h \geqslant 1$. For $x_h < 1$, we show that the properties of the algorithm presented in Relation (4.5)

can not be true.
Let $1 \leqslant x < 2$ be written as $x = 1 + w$, such that $0 \leqslant w < 1$. Since x_h is a multiple of 2^{s-m+1},

$$x_h < 1 \implies x_h \leqslant 1 - 2^{s-m+1}. \tag{4.6}$$

From Relation (4.5), we have:

$$\begin{aligned} x_l &= x - x_h, \\ x_l &\geqslant 1 + w - 1 + 2^{s-m+1}, \\ x_l &\geqslant w + 2^{s-m+1}, \\ x_l &\geqslant 2^{s-m+1}, \end{aligned} \tag{4.7}$$

yet it is proven in [40] as recalled in Relation (4.5) that $x_l \leqslant 2^{s-m}$. Therefore $x_h < 1$ can not be true. We deduce that $x_h \geqslant 1$, which implies that $\mathtt{exponent}(x_h) \geqslant \mathtt{exponent}(x)$.

□

Theorem 4.2. *Let* p *be a vector of* n *floating-point numbers built with Veltkamp's split algorithm for* $s = \lceil m/2 \rceil$*, such that:*

- *either* p_i *is a leading part with* $\mathtt{exponent}(p_i) = e$*;*
- *or* p_i *is the trailing part of a leading part* x_h *such that* $\mathtt{exponent}(x_h) = e + \lceil m/2 \rceil$.

For $n < 2^{\lfloor m/2 \rfloor}$*, the accumulation* $C = p_1 \oplus p_2 \oplus ... \oplus p_n$ *is a floating-point number.*

Proof. Let x be a floating-point number, and e_x its exponent. Muller and al. [40] have shown that $(x_h, x_l) = \mathtt{split}(x, s)$ are such that:

- $|x_h| \leqslant 2^{e_x+1}$ and $\mathtt{lnb}(x_h) \geqslant 2^{e_x+s-m+1}$;
- $|x_l| \leqslant 2^{e_x+s-m}$ and $\mathtt{lnb}(x_l) \geqslant 2^{e_x-m+1}$.

For `binary64` format and using $s = \lceil m/2 \rceil$, those relations are written as:

$$\begin{aligned} &|x_h| \leqslant 2^{e_x+1} \text{ and } \mathtt{lnb}(x_h) \geqslant 2^{e_x-\lfloor m/2 \rfloor+1}, \\ &|x_l| \leqslant 2^{e_x-\lfloor m/2 \rfloor} \text{ and } \mathtt{lnb}(x_l) \geqslant 2^{e_x-m+1}. \end{aligned} \tag{4.8}$$

Here, a leading part, $x_h \in p$ is such that $exponent(x_h) = e$. So Relation (4.8) yields

$$\begin{aligned} |x_h| &< 2^{e+1}, \\ lnb(x_h) &\geqslant 2^{e-\lfloor m/2 \rfloor + 1}. \end{aligned} \tag{4.9}$$

On the other side, a trailing value $x_l \in p$ is such that $exponent(x_h) = e + \lceil m/2 \rceil$. Muller and al. show in [40] that $exponent(x_h) \leqslant e_x + 1$, and Lemma 4.1 shows that $exponent(x_h) \geqslant e_x$. Therefore we distinguish two cases here. i) If $exponent(x_h) = e_x$, we have $e_x = e + \lceil m/2 \rceil$ and

$$|x_l| \leqslant 2^{e_x - \lfloor m/2 \rfloor} = 2^{e + \lceil m/2 \rceil - \lfloor m/2 \rfloor} = 2^{e+1}, \tag{4.10}$$

and

$$lnb(x_l) \geqslant 2^{e_x - m + 1} = 2^{e + \lceil m/2 \rceil - m + 1} = 2^{e - \lfloor m/2 \rfloor + 1}. \tag{4.11}$$

ii) In the second case, $e_x = e + \lceil m/2 \rceil - 1$, that yields:

$$|x_l| \leqslant 2^{e_x - \lfloor m/2 \rfloor} = 2^{e + \lceil m/2 \rceil - 1 - \lfloor m/2 \rfloor} = 2^{e}, \tag{4.12}$$

and

$$lnb(x_l) \geqslant 2^{e_x - m + 1} = 2^{e + \lceil m/2 \rceil - 1 - m + 1} = 2^{e - \lfloor m/2 \rfloor}. \tag{4.13}$$

We deduce from Relations (4.9), (4.10), (4.11), (4.12) and (4.13) that all $\forall p_i \in p$ are such that:

$$\begin{aligned} |p_i| &\leqslant 2^{e+1}, \\ lnb(p_i) &\geqslant 2^{e - \lfloor m/2 \rfloor}. \end{aligned} \tag{4.14}$$

For $C = \sum_{i=1}^{n} p_i$, we have

$$|C| \leqslant n \cdot 2^{e+1},$$

so for $n < 2^{\lfloor m/2 \rfloor}$,

$$\begin{aligned} |C| &< 2^{e + \lfloor m/2 \rfloor + 1}, \\ ufp(C) &\leqslant 2^{e + \lfloor m/2 \rfloor}. \end{aligned} \tag{4.15}$$

According to Relations (4.14) and (4.3)

$$\mathrm{lnb}(C) \geqslant 2^{e-\lfloor m/2 \rfloor}. \tag{4.16}$$

From Relations (4.15) and (4.16), and following the same process as in Theorem 4.1, we find that

$$\mathrm{lnb}(C) \geqslant \mathrm{ufp}(C) \cdot 2u.$$

Therefore, C is a floating-point number.

□

Theorem 4.2 proves that it is possible to avoid to compute the exponent of the trailing part of split results. In practice, this optimization is very efficient since it replaces with a single integer subtraction two operations (mask and shift) that requires also to move a floating-point number to an integer register.

Algorithm 4.1 HybridSum transformation with single exponent computation

Input: p: a vector of n floating-point numbers
Output: C: a vector of 2048 floating-point numbers

1: **for** $(i = 1; i \leqslant n; i = i + 1)$ **do**
2: $\quad (p_{i,h}, p_{i,l}) = split(p_i)$
3: $\quad e_h = exponent(p_{i,h})$
4: $\quad C_{e_h} = C_{e_h} \oplus p_{i,h}$
5: $\quad C_{e_h - \lceil m/2 \rceil} = C_{e_h - \lceil m/2 \rceil} \oplus p_{i,l}$
6: **end for**

For our implementation of HybridSum, each accumulator sustains $n < 2^{\lfloor m/2 \rfloor}$ accumulations. However, counting the number of floating-point numbers accumulated to each C_e is costly both in terms of time and space, Therefore, the authors suggest in [69] to only keep track of the total number of summands. Each time the latter reaches the maximum size supported by the accumulators, the accumulators themselves are split and accumulated to a new accumulator vector according to their current exponent. Then the process continues on the rest of the input vector [69].

4.2.2 *Optimization of OnlineExact*

Like HybridSum, OnlineExact transforms the input vector to a smaller one by accumulating inputs according to their exponent. As described in Algorithm 3.7 OnlineExact uses a size 2 expansion to simulate a higher precision accumulator. Each accumulator C_e consists of two standard floating-point numbers $C_{e,h}$ and $C_{e,l}$ (lines 3, 4). The input does not need to be split, it is instead accumulated to $C_{e,h}$. The error of this first operation can be computed using algorithm TwoSum and accumulated to $C_{e,l}$, the second part of accumulator. The authors have shown in [70] that up to $2^{\lfloor m/2 \rfloor}$ floating-point numbers with the same exponent can be exactly accumulated to a size 2 expansion. When the input vector is larger than $2^{\lfloor m/2 \rfloor}$, the accumulators themselves are considered as separate floating-point numbers and accumulated according to there exponents to a new fresh vector, then the process continues.

4.2.2.1 *Optimization Using* FastTwoSum

Let C be an accumulator that consists of two floating-point numbers C_h and C_l (lines 3, 4). And let p be the vector of floating-point numbers that are accumulated into C (all elements of p have the same exponent e).

Each element of p is accumulated to C_h using algorithm TwoSum, then the computed error is accumulated into C_l. We prove that TwoSum in this case can be replaced by FastTwoSum without having to compare its input.

We can ensure that $(s, e) = \mathrm{FastTwoSum}(a, b)$ is an error-free transformation only if we have preliminary knowledge about the larger element between a and b: $|a| \geqslant |b|$ or $\mathrm{exponent}(a) \geqslant \mathrm{exponent}(b)$ (see Algorithm 2.2). Comparing a and b and permuting inputs if necessary is possible, yet the additional cost of branch related to the comparison operation might be higher than the cost of the three floating-point operations that we gain compared to the algorithm TwoSum – this depends on the branch

predictor performance and misprediction cost.

Rump and al. have weakened the requirement for algorithm FastTwoSum to guarantee that the transformation is error-free. They have shown in [54] that FastTwoSum works if $\mathrm{lnb}(a) \geqslant \mathrm{ulp}(b)$ independently from the order of the inputs a and b. We prove that this condition is satisfied in our case.

Note that for each element p_i of exponent e in p we have:

$$\begin{aligned} &\mathrm{ufp}(p_i) = 2^e, \\ &\mathrm{ulp}(p_i) = \mathrm{ufp}(p_i) \cdot 2u = 2^e \cdot 2u, \\ &\mathrm{lnb}(p_i) \geqslant 2u \cdot 2^e. \end{aligned}$$

Therefore, for each intermediate value of C_h and each $p_i \in p$, and according to Relation (2.6)

$$\mathrm{lnb}(C_h) \geqslant \min_i(\mathrm{lnb}(p_i)) \geqslant 2^e \cdot 2u = \mathrm{ulp}(p_i).$$

Elements of p might have different signs. Therefore, it is possible for $|C_h|$ to be smaller than $|p_i|$ due to cancellations. However, $\mathrm{FastTwoSum}(C_h, p_i)$ is still guaranteed to be an error-free transformation of $C_h + p_i$ because $\mathrm{lnb}(C_h) \geqslant \mathrm{ulp}(p_i)$.

Note also that all demonstrations from the original paper [70] and those in this section remain valid for

$$\begin{aligned} \mathrm{ufp}(p_i) &\leqslant 2^e, \\ \mathrm{lnb}(p_i) &\geqslant 2u \cdot 2^e. \end{aligned} \tag{4.17}$$

Algorithm 4.2 OnlineExact transformation using FastTwoSum

Input: p: a vector of n floating-point numbers
Output: Ch, Cl: two vectors of 2048 floating-point numbers

1: **for** $(i = 1; i \leqslant n; i = i + 1)$ **do**
2: $\quad e = exponent(p_i)$
3: $\quad (Ch_e, error) = \mathrm{FastTwoSum}(Ch_e, p_i)$
4: $\quad Cl_e = Cl_e + error$
5: **end for**

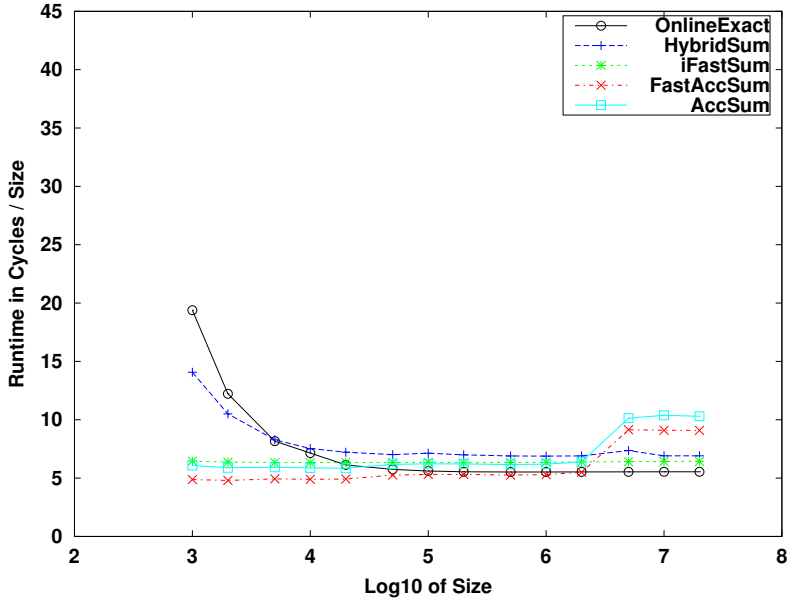
4.2.3 *Runtime Performance for Summation Algorithms*

We implemented and tested the presented algorithms that guarantee either correctly or faithfully rounded summation result. We focus on runtime performance only in this section since all the tested algorithms have the same accuracy. Accuracy is addressed in more details in Section 4.5.

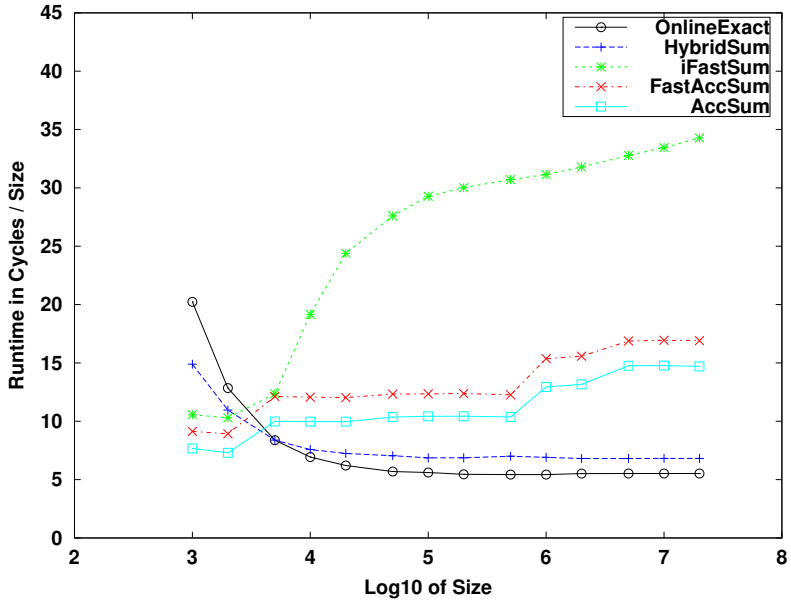
Our experimental approach is explained in Appendix B. The generator introduced in Section B.4.1 is used to generate two sets of vectors with condition numbers 10^8 and 10^{32} and different vector sizes. We perform our tests on environment A (workstation, Table B.1), and we measure runtime in cycles using the assembly instruction `rdtsc`.

Figures 4.1a and 4.1b respectively exhibit tested algorithm runtimes for condition numbers 10^8 and 10^{32}. We show on the X-axis the `log10` of the input vector size, and on the Y-axis the number of cycles divided by the size of the vector. Every instance is run 16 times, and only the minimum execution time is presented to minimize measurement errors. Insignificant running time measures are excluded.

For summations with small condition number (10^8), Figure 4.1a shows that all the five presented algorithms have almost similar performance. However, for small vectors both algorithms `HybridSum` and `OnlineExact` are not very efficient due to their nature of the algorithms that transforms large vectors to smaller ones and then apply an iterative summation algorithm. For larger input vectors, algorithm `OnlineExact` is a little more efficient than the other algorithms, yet the difference is not very significant. When data no more fits in the L3 cache, both the algorithms `AccSum` and `FastAccSum` run much slower than the other algorithms. Figure 4.1b shows the running time of the same algorithms for larger condition numbers (10^{32}). In this case, iterative algorithms are less efficient since they require more iterations and therefore multiple loads of the input vector and more floating-point operations. At the contrary, `HybridSum` and `OnlineExact` that do not depend on condition number behave



(a) Cond = 10^8



(b) Cond = 10^{32}

Figure 4.1: Performance results for summation algorithms

as efficient as before.

Figure 4.2 shows the running time for previous algorithms with respect to the condition number. On the X-axis we present the log10 of the condition number. On the Y-axis we show the number of cycles. We already mentioned that algorithms HybridSum and OnlineExact do not depend on the condition number. There-

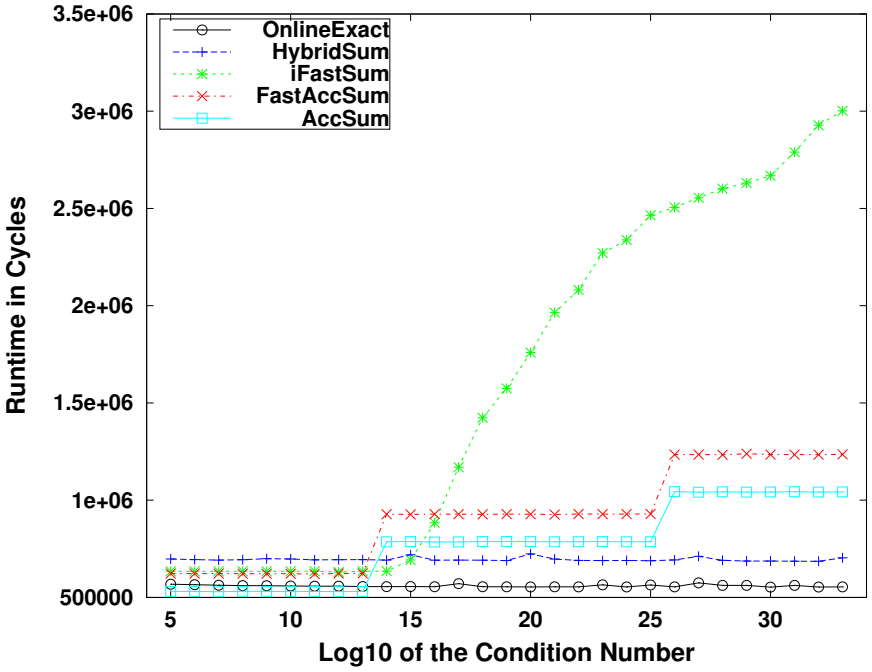


Figure 4.2: Performance according to condition number (vector size = 50000)

fore both algorithms are more efficient for ill conditioned problems. The only drawback of HybridSum and OnlineExact is that they are less efficient for small input data.

4.2.4 *Our Hybrid Summation Algorithm* Rsum

We have shown in the previous section that the efficiency of tested summation algorithms mainly depends on the size and the condition number of the problem. We introduce one hybrid solution that consists in selecting the used algorithm according to the input size. OnlineExact is used for large vectors (> 5000 according to our experiments). For smaller vectors FastAccSum is the most efficient algorithm. However we prefer using iFastSum since its results are correctly rounded while FastAccSum only provides faithfully rounded results. We call this hybrid algorithm Rsum. We next present its parallel version and how it compares to other solutions.

4.3 PARALLEL CORRECTLY ROUNDED SUM

Our algorithm for a parallel correctly rounded summation is presented in Figure 4.3. We distinguish 3 different steps.

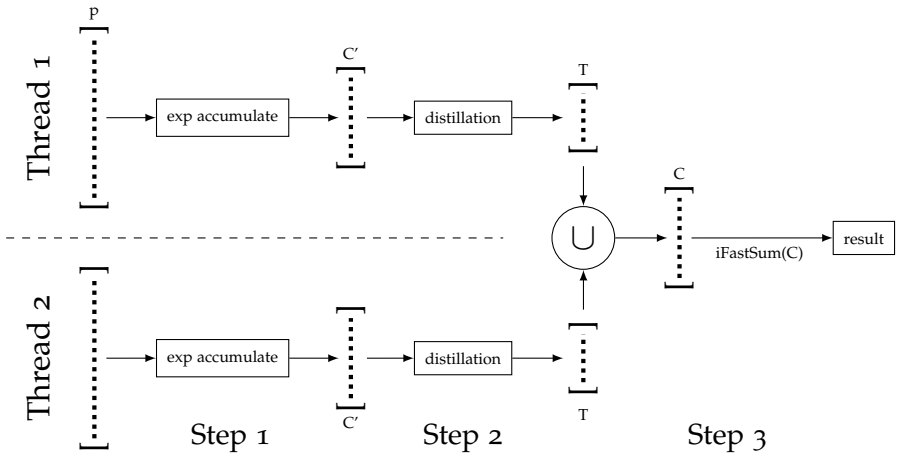


Figure 4.3: Parallel algorithm for correctly rounded sum (entry vector p is distributed to every thread, here 2)

STEP 1 The input vector p has been split between all available threads. Then each thread uses Algorithm 3.7 (OnlineExact) to accumulate elements with the same exponent. Algorithm 3.6 (HybridSum) also can be used, yet it is slightly less efficient for our test environment. This step consists in reducing the size of the local data to a smaller vector C' that fits in L1 or L2 caches. The size of C' is 4096 binary64 numbers when using Algorithm 3.7 and 2048 for Algorithm 3.6. For smaller vectors, this step does not need to be executed, and we start with the second step instead. Note that a single run through the input data is performed at this level.

STEP 2 In this second step, the local vector C' is distilled. An iterative distillation algorithm is efficient on a vector that fits in a high level cache (L1 or L2). The result of this distillation is a vector T such that elements of T are non-overlapping and ordered by magnitude. In other words for each index i:

$$\mathrm{ulp}(T_i) > \mathrm{ufp}(T_{i+1}).$$

This property guarantees that vector T can not contain less elements and so requires the smallest possible memory space. The size of T is bounded by the size of the exponent range divided by the size of the mantissa: it is smaller than $2048/53 \approx 39$ for binary64. Note that this step reduces the size of C' so the sequential computation in the next step takes minimal time.

STEP 3 In this last step we gather the distillation result of all threads into a single vector C. For t threads, a binary tree union with $O(\log 2(t))$ complexity can be implemented. Since all the operations that we have performed are error-free transformations, it is guaranteed that:

$$\sum_{i=1}^{n} p_i = \sum_{j=1}^{m} C_j.$$

Finally, we use the summation algorithm iFastSum to compute the correctly rounded sum $S = RN(\sum_{i=1}^{n} p_i)$ of the vector C.

4.4 RUNTIME PERFORMANCE FOR OUR PARALLEL SUMMATION

Before comparing our parallel summation algorithm Rsum to other available solutions, we show its scalability in Figure 4.4. Our implementation is run in sequential and parallel with different thread configurations. X-axis shows log10 of the input data size, while Y-axis shows the running time in cycles divided by the input size. Our summation algorithm scales well up to 16 threads for large vectors (the number of available cores in our environment A in Table B.1). The same algorithm is tested with 240 threads on the environment B presented in Table B.1 (Intel Xeon Phi accelerator). We recall that for the sequential version, iFastSum is used for small vectors and OnlineExact for large vectors.

Our correctly rounded Rsum is also compared to other available solutions. To illustrate the performance of our algorithm we compare its running time to the BLAS routine cblas_dasum (sum of absolute values) implemented in Intel MKL library. No parallel version for cblas_dasum is provided in MKL. We use

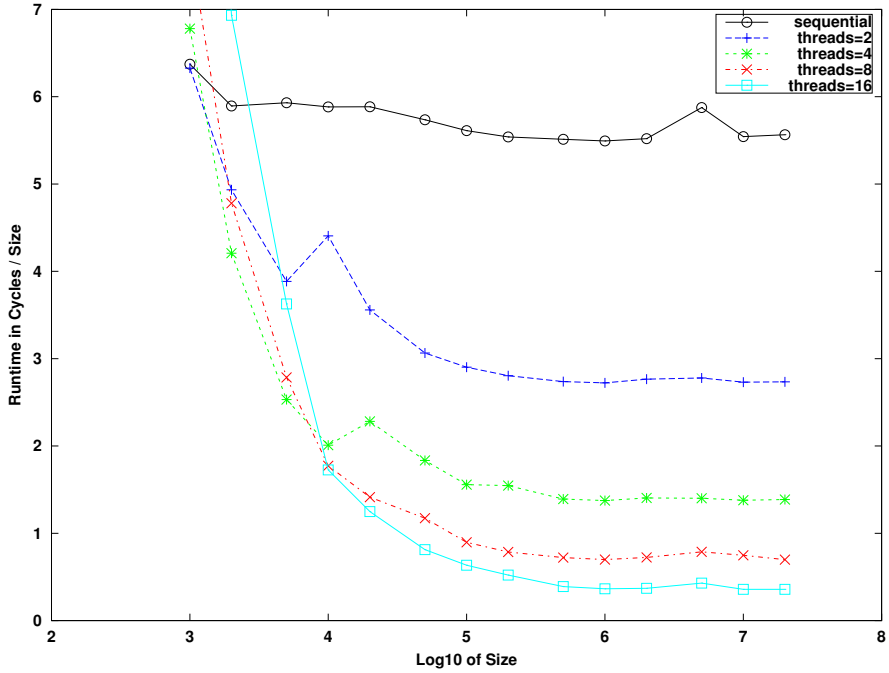


Figure 4.4: Rsum: scaling (environment A, cond = 10^8)

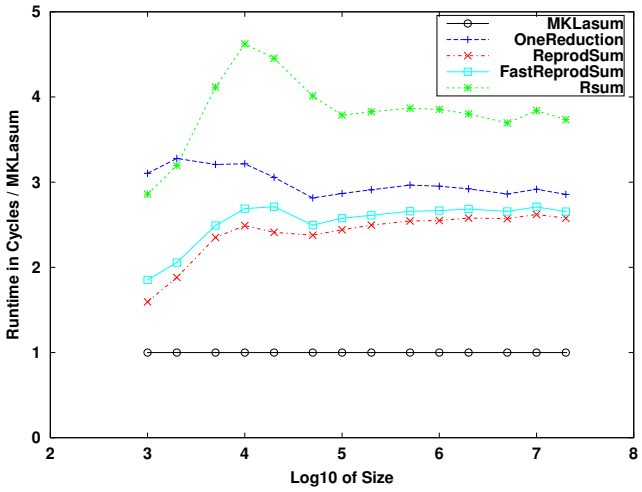
the optimized sequential MKL version to implement a parallel one. The input vector is split according to the number of threads, the sequential version is used to accumulate the local data for each thread, then one OpenMP reduction yields the final result.

Our solution is also compared to the algorithms ReprodSum [11], FastReprodSum [11] and 1-Reduction [12] presented in Sections 3.5.1, 3.5.2 and 3.5.4 respectively. We had to implement all those algorithms instead of using the ReproBLAS library itself since it does not offer a shared memory parallel version[2]. In Section 4.6 our solution is also compared to the ExBLAS library.

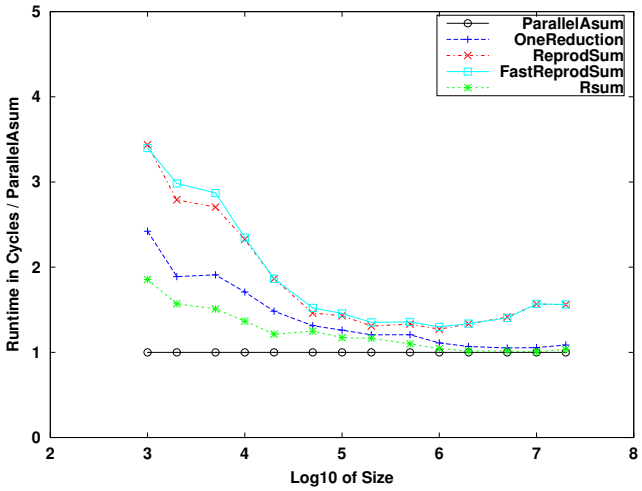
Figure 4.5 exhibits the performance of our algorithm on different architectures. We present on the X-axis the log10 of the input vector size, and we show the execution time normalized by the MKL based summation on the Y-axis.

In Figure 4.5a we show that in the optimized sequential case our algorithm runs about 4× slower than MKL implementation

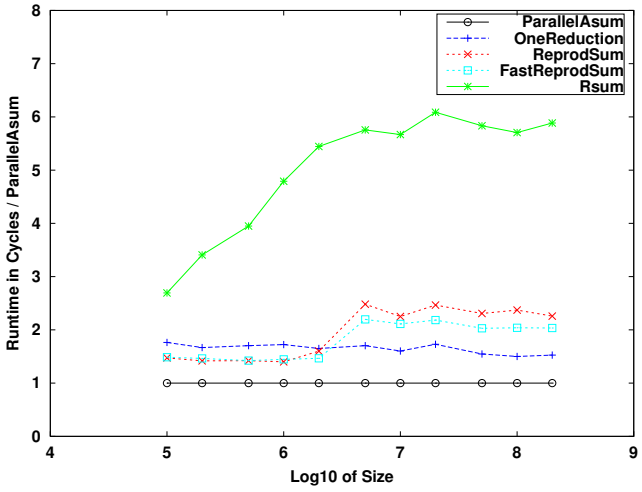
2 At the time we are writing this document only sequential and MPI parallel versions are available



(a) Sequential version (environment A)



(b) CPU parallel (16 threads, environment A)



(c) Xeon Phi version (240 threads)

Figure 4.5: Rsum: Performances (Cond = 10^8)

that is neither reproducible nor accurate. Reproducible solutions ReprodSum (K = 2), FastReprodSum (K = 2) and 1-Reduction (K = 3) are about 2× to 3× slower than MKL's implementation. However, they do not ensure that the result is correctly rounded.

Figure 4.5b shows that for the CPU parallel case our algorithm and $1 - \text{Reduction}$ are as efficient as the parallel summation based on MKL. Note that at this level all those three algorithms are reaching the scaling limit imposed by the memory bandwidth. ReprodSum and FastReprodSum algorithms are about 2× slower than the others because both run through the input data twice. Therefore, they suffer twice from the bandwidth limitation.

On Xeon Phi accelerator, we show in Figure 4.5c that our summation algorithm is about 6× slower than the optimized summation, and about 3× slower than the other reproducible summation algorithms. Our algorithm suffers from operations that can not be vectorized: when we accumulate the elements of the input vector in the vector C according to their exponent we access to non-contiguous data. Therefore, there is no way to efficiently vectorize those operations. Unlike the other algorithms, our summation do not benefit from the AVX-512 registers available on Xeon Phi.

4.5 ACCURACY RESULTS

In this section we present accuracy results for the different summation algorithms. We consider classic summation algorithm (Algorithm 2.1), reproducible algorithms ReprodSum, FastReprodSum and 1-Reduction, and we experimentally confirm that our summation algorithm ensures correctly rounded results.

Accuracy results are presented in Figure 4.6. On the X-axis we show the log10 of the condition number, the Y-axis shows the log10 of the relative error of the computed result compared to the correctly rounded one calculated using the MPFR library [62].

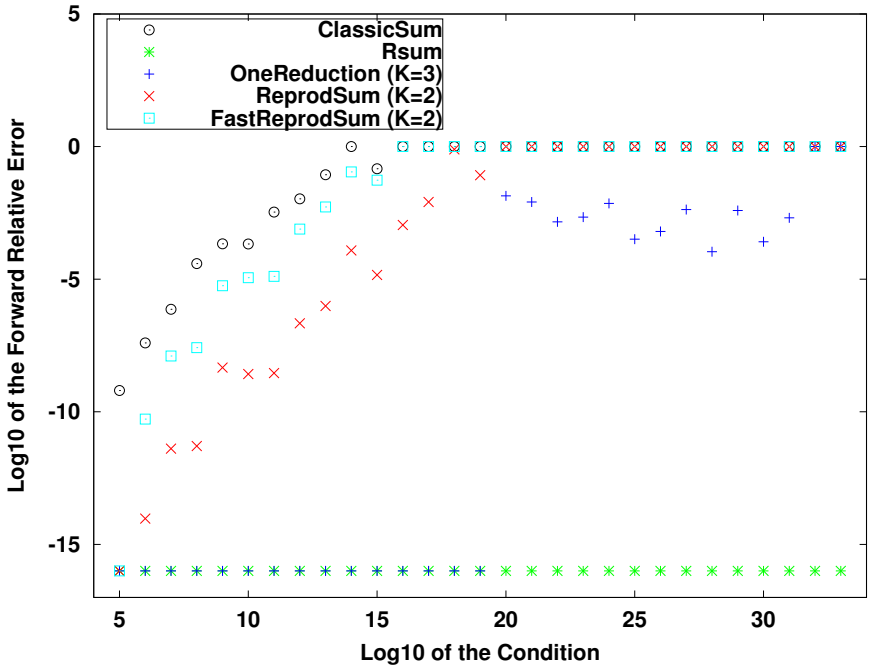


Figure 4.6: Accuracy results for different summation algorithms (vector size = 100000)

The accuracy of the classic summation algorithm decreases for large condition numbers. For the tested vector size, provided result does not contain any correct significant digit when the condition number is larger than 10^{15}. Reproducible solutions are more accurate than the classic summation algorithms. However, despite being reproducible, the result is also of a very poor significance for problems with large condition number. Finally, our algorithm guarantees that the summation result is always correctly rounded and that it does not depend anymore on the condition number of the input data.

Note that both algorithms ReprodSum and 1-Reduction will be more accurate for larger K. However they will require more computations and will certainly run slower than for the current value of K To obtain results similarly accurate to Rsum, other reproducible solutions need to identify K according to the input condition number.

4.6 COMPARE TO EXBLAS

We consider the summation algorithm provided by the ExBLAS library. This latter is available online at [19]. Principle of the algorithm used in ExBLAS are presented in Section 3.6.3 and the full algorithm is presented in [9]. The generator provided by the ExBLAS library is used to generate input vectors with different exponent ranges. This latter is defined by the difference between the maximum and the minimum exponents of input floating-point numbers (decimal exponent). Two ranges of 8 and 32 have been considered.

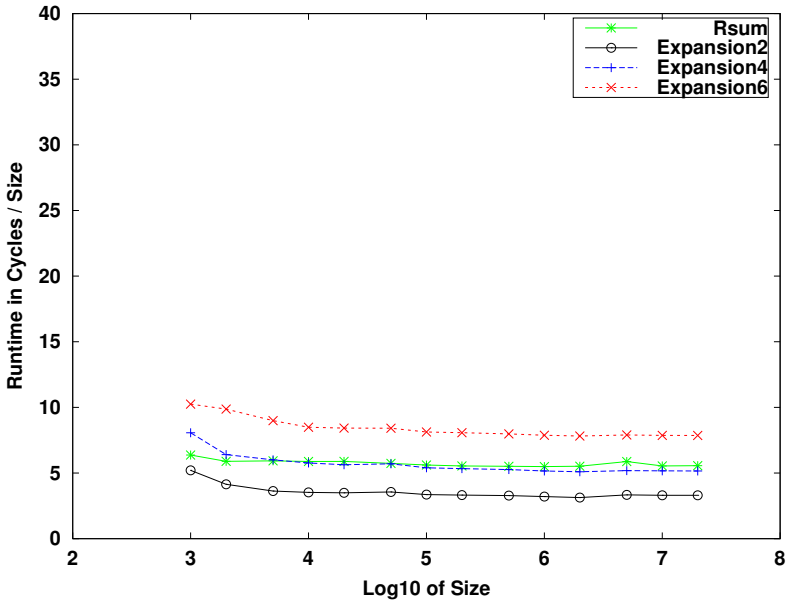
We recall that ExBLAS uses floating-point expansions and superaccumulators to add the inputs without any error. The expensive process where the input moves to the superaccumulator of is performed only when the result does no more fit in the expansion. Note that large expansions are more efficient for problems with large exponent range, while small expansions are suited for problems with smaller exponent range. We test 3 sizes for the expansion, 2, 4 and 6. To study the performance in different scenarios we compare our solution to ExBLAS summation with different expansion size and for different exponent ranges.

Figure 4.7 presents our Rsum algorithm compared to ExBLAS summation in the sequential case for two different exponent ranges, and with expansion sizes: $2, 4$ and 6.
In Figure 4.7a we show that small expansions (of size 2) are very efficient, while larger ones increases the running time ExBLAS summation up to the cost cost of our Rsum for size 4 expansion, and more than Rsum for size 6 expansion.
For larger exponent range, Figure 4.7b shows that size 2 expansions are no more efficient because this expansion size is not large enough to hold the accumulation result. Therefore, the costly access to superaccumulator is more regular. However, larger expansions of sizes 4 and 6 exhibit the same performance as for small exponent range.

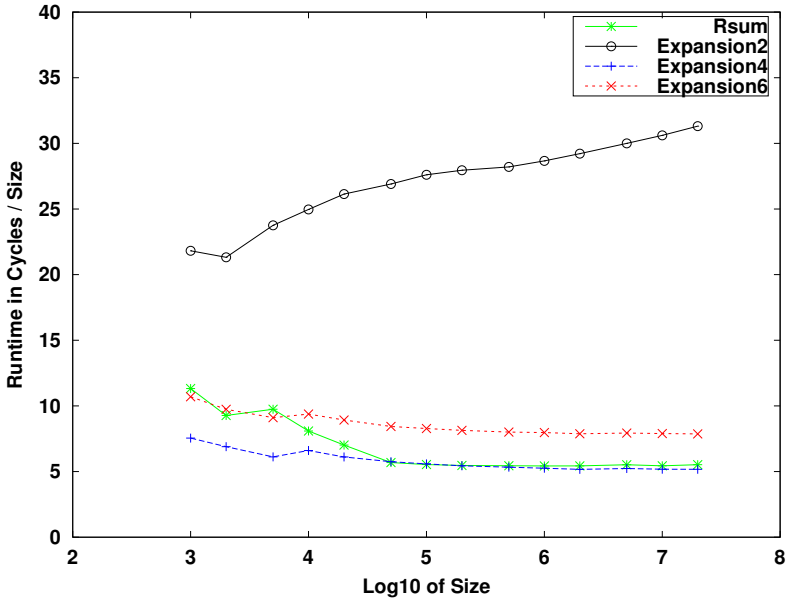
Parallel results are shown in Figure 4.8. For the sake of readability, running times are normalized by our summation algo-

rithm (see Y-axis legend in Figures 4.8a and 4.8b).

Figure 4.8a shows the result for small exponent range. We have already explained that our summation algorithm reaches the memory bandwidth limit. Both configurations with expansion of size 2 or 4 reach the same performance. Size 6 expansion

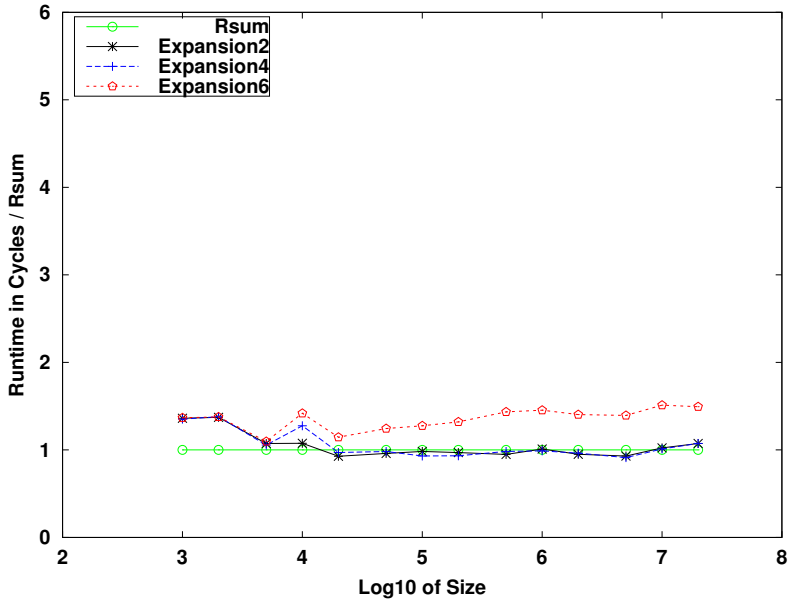


(a) Exponent range = 8

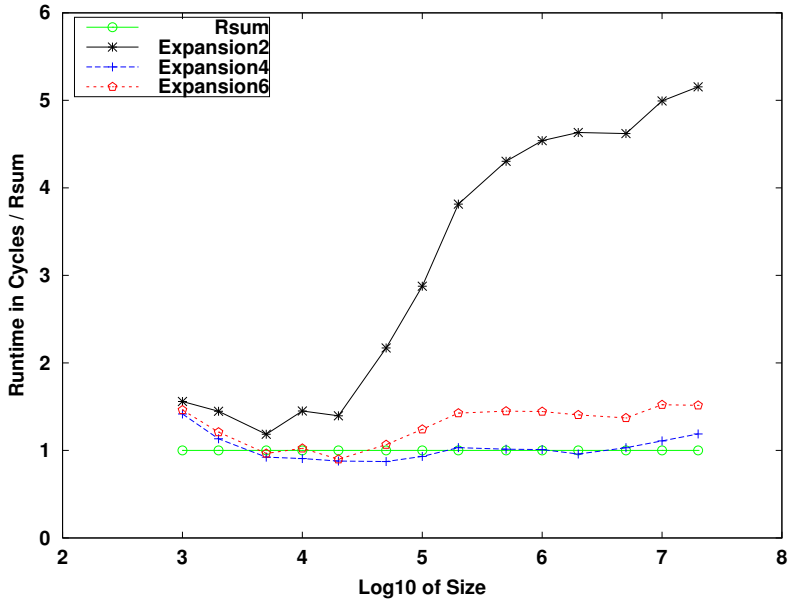


(b) Exponent range = 32

Figure 4.7: Rsum vs ExBLAS: relative performance (ratio/size), summation, sequential case



(a) Exponent range = 8



(b) Exponent range = 32

Figure 4.8: Rsum vs ExBLAS: relative performance (ratio/size), summation, parallel case (threads = 16)

is resource intensive and is not able to reach the same efficiency even when all available cores are used.

For larger exponent range, we observe with Figure 4.7b the same behavior as for the sequential case, when size 2 expansion introduces a heavy loss of efficiency. Expansions with size 4 and

6, despite being less efficient than size 2 expansion for small exponent range, keep the same efficiency for large exponent range.

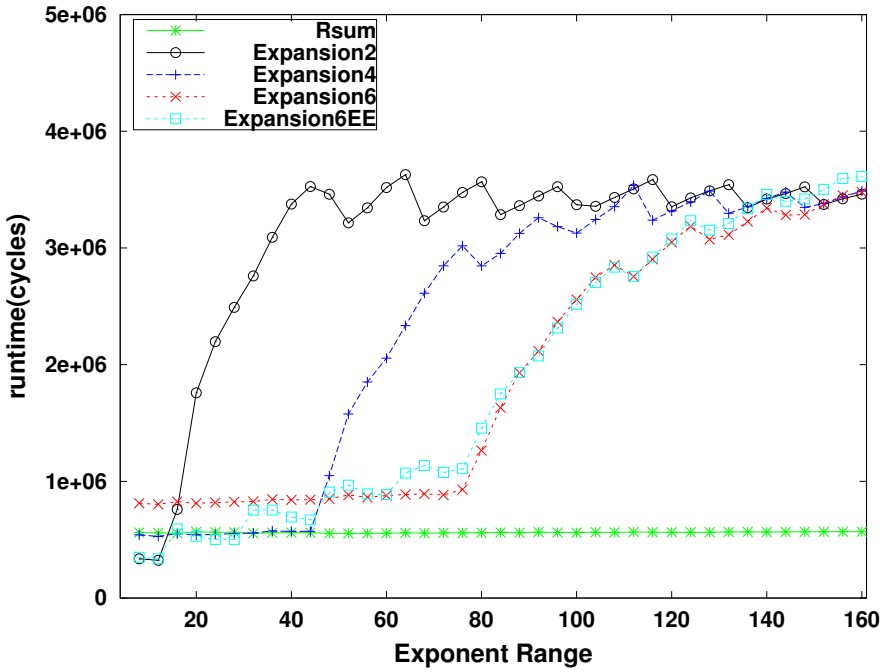


Figure 4.9: Impact of the exponent range on summation algorithm efficiency (vector size = 10^5)

To show the impact of the exponent range on summation algorithms, we exhibit in Figure 4.9 the runnings time of our solution and ExBLAS summation for different exponent ranges and a given vector size. Exponent ranges between 4 and 160 are tested. We recall that the decimal exponent range of binary64 format is about 620. Exponent range is shown on the X-axis, and running time in cycles is shown on the Y-axis. We also included the early-exit version of the ExBLAS algorithm (denoted as Expansion6EE). It consists in avoiding to perform useless operations when a floating-point number is accumulated into an expansion. The algorithm used in ExBLAS consists in accumulating the input into the first element of expansion array using the algorithm TwoSum. Then the committed error is accumulated to the next expansion element until reaching the expansion end. Early-exit consists in testing if the error is zero before accumulating it to the next expansion element. This technique is particularly efficient when a large expansion is used for inputs with small expo-

nent range.

Figure 4.9 shows that the only configuration that is more efficient than our summation algorithm is when using size 2 expansion or when early-exit is used for small exponent ranges (< 16). Using larger expansions is more efficient for large exponent ranges. However, larger expansions requires much more floating-point operations for the accumulation to be performed. Therefore, our algorithm compares favorably to ExBLAS when the input is not restricted to a small exponent range or when large expansions are used.

4.7 CONCLUSION

In this chapter we presented Rsum, a new parallel algorithm for reproducible correctly rounded summation. It relies on error-free transformations and appears to be very efficient in terms of running time and accuracy compared to other existing parallel summation algorithms. Rsum relies on different summation algorithms depending on the input size. It uses OnlineExact is used to error-free transform the input data at a constant cost. Some algorithmic changes have been introduced to this latter to improve its running time performance. We have shown in section 4.4 that our algorithm scales almost perfectly for large vectors up to 16 cores. Compared to MKL based implementation, the parallel version of our solution has almost no extra-cost when it runs on CPU, and up to $5\times$ extra-cost when it runs on Xeon Phi accelerator. However, it always ensures that the result is correctly rounded and its timing performance and accuracy does not depend neither on the condition number nor on the exponent range of the input. Section 4.6 shows that our algorithm compares very favorably to ExBLAS summation that also ensures a correctly rounded result.

5

REPRODUCIBLE LEVEL 1 BLAS

5.1 INTRODUCTION

Chapter 4 introduces our algorithm for reproducible and correctly rounded floating-point summation. In this chapter, we focus on level 1 BLAS subroutines. Subroutines at this level performs scalar-vector and vector-vector operations with a linear complexity, i.e. O(n) where n is the entry dimension. We consider the subroutines `dot` (dot product), `asum` (sum of absolute values) and `nrm2` (euclidean norm) and introduce a reproducible solution for their computation. Algorithms are designed for these subroutines. Performance tests show that our parallel implementation for CPU has about $2\times$ extra-cost compared to implementations based on Intel MKL library in the worst case scenario. For Xeon Phi implementations the extra-cost is up to about $6\times$ and this overhead is explained as we did for summation in chapter 4.

5.2 DOT PRODUCT

Dot product takes as input two vectors x and y of equal length and computes the scalar corresponding to the sum of the products $x_i \times y_i$ as:

$$S = \sum_{i=1}^{n} x_i \cdot y_i.$$

Dot product is ubiquitous in scientific computing and is also one important building block for higher level BLAS subroutines. In the next section we explain how the dot product of two n-vectors can be transformed with no error to a sum of a $2n$-vector. Therefore we use `Rsum` to implement a correctly rounded dot product.

5.2.1 *Sequential Dot Product*

The dot product of two n-vectors is transformed into a sum of a $2n$-vector using algorithms TwoProd or 2MultFMA (Algorithms 2.6 and 2.7). Algorithm 2MultFMA is more efficient, yet it requires an FMA floating-point unit to be available. Let us compute (p_i, r_i) such that $p_i + r_i = x_i \times y_i$ for every (x_i, y_i) thanks to TwoProd or 2MultFMA according to Relation (2.12). This error-free transformation ensures that

$$\sum_{i=1}^{n} x_i \cdot y_i = \sum_{i=1}^{n} p_i + \sum_{i=1}^{n} r_i = \sum_{i=1}^{2n} q_i.$$

with $q_i = p_i$ and $q_{n+i} = r_i$. Therefore, we have the following floating-point equality:

$$\mathrm{round}\left(\sum_{i=1}^{n} x_i \cdot y_i\right) = \mathrm{round}\left(\sum_{i=1}^{n} p_i + \sum_{i=1}^{n} r_i\right). \tag{5.1}$$

The sum of the $2n$-vector (the right part of Relation (5.1)) is performed using algorithm iFastSum for small length data. For larger input data, algorithm OnlineExact is used. Note that the $2n$-vector does not need to be created when using algorithm OnlineExact. The multiplication result p_i and its error r_i can be accumulated directly and exactly according to their exponents to vector C.

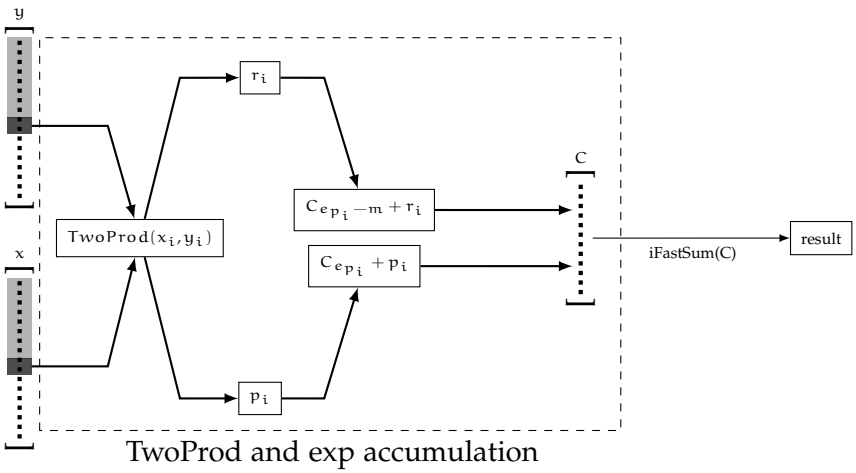


Figure 5.1: Algorithm for a sequential correctly rounded dot product

Let e_{p_i} be the exponent of p_i. We prove that even if the exponent of the error r_i is not computed and is accumulated to the accumulator $C_{e_{p_i}-m}$ as shown in Figure 5.1, Relation (4.17) remains valid.

Theorem 5.1. *Let x, y be two floating-point numbers of precision u and mantissa length m. The error-free transformation:*

$$(p, r) = \mathsf{TwoProd}(x, y)$$

verifies:

$$\begin{aligned} \mathrm{ufp}(r) &\leqslant 2^e, \\ \mathrm{lnb}(r) &\geqslant 2u \cdot 2^e, \end{aligned}$$

where $e = e_p - m$ for $e_p = \mathrm{exponent}(p)$.

Proof. Let e_x and e_y be the exponents of x and y. We assume that no overflow nor underflow occurs and that rounding to nearest mode is used since it is required for algorithm TwoProd to work properly. Without loss of generality, we also assume that $x > 0$ and $y > 0$. Therefore we have:

$$\begin{aligned} x &\leqslant 2^{e_x+1} - \mathrm{ulp}(x), \\ y &\leqslant 2^{e_y+1} - \mathrm{ulp}(y). \end{aligned}$$

The real number $z = x \times y$ is such that:

$$\begin{aligned} z &\leqslant (2^{e_x+1} - \mathrm{ulp}(x)) \times (2^{e_y+1} - \mathrm{ulp}(y)), \\ z &\leqslant 2^{e_x+1} \times (1-u) \times 2^{e_y+1} \times (1-u), \\ z &\leqslant 2^{e_x+e_y+2} \times (1-u)^2, \\ z &< 2^{e_x+e_y+2}, \\ \mathrm{ufp}(z) &\leqslant 2^{e_x+e_y+1}. \end{aligned} \tag{5.2}$$

Starting from Relation (2.6):

$$\mathrm{lnb}(z) = \mathrm{lnb}(x) \times \mathrm{lnb}(y),$$

and since for each floating-point number x, $\mathrm{lnb}(x) \geqslant \mathrm{ulp}(x)$ we deduce that:

$$\mathrm{lnb}(z) \geqslant \mathrm{ulp}(x) \times \mathrm{ulp}(y).$$

According to Relation (2.4), we write:

$$\begin{aligned}
\mathrm{lnb}(z) &\geqslant 2u \times \mathrm{ufp}(x) \times 2u \times \mathrm{ufp}(y),\\
\mathrm{lnb}(z) &\geqslant 2u \times 2^{e_x} \times 2u \times 2^{e_y},\\
\mathrm{lnb}(z) &\geqslant (2u)^2 \times 2^{e_x+e_y},\\
\mathrm{lnb}(z) &\geqslant 2^{e_x+e_y+2-2m}.
\end{aligned}$$

Note that TwoProd ensures that $p = x \otimes y$, so $p = \mathrm{round}(z)$. According to Relation (5.2) we have:

$$\begin{aligned}
z &\leqslant 2^{e_x+e_y+2} \times (1-u)^2,\\
\mathrm{round}(z) &\leqslant \mathrm{round}(2^{e_x+e_y+2} \times (1-u)^2),\\
p &\leqslant 2^{e_x+e_y+2} \times \mathrm{round}((1-u)^2).
\end{aligned}$$

Since $(1-u)^2 < (1-u)$, we have:

$$\begin{aligned}
p &\leqslant 2^{e_x+e_y+2} \times \mathrm{round}(1-u),\\
\mathrm{ufp}(p) &\leqslant 2^{e_x+e_y+2} \times \mathrm{ufp}(1-u),\\
\mathrm{ufp}(p) &\leqslant 2^{e_x+e_y+2} \times 2^{-1},\\
\mathrm{ufp}(p) &\leqslant 2^{e_x+e_y+1}.
\end{aligned}$$

We recall that r is the multiplication error, so

$$\begin{aligned}
r &\leqslant \frac{\mathrm{ulp}(p)}{2},\\
r &\leqslant \frac{2u \cdot \mathrm{ufp}(p)}{2},\\
r &\leqslant u \cdot \mathrm{ufp}(p).
\end{aligned} \tag{5.3}$$

And since $r = z - p$,

$$\mathrm{lnb}(r) \geqslant \min(\mathrm{lnb}(z), \mathrm{lnb}(p)). \tag{5.4}$$

Note that $p = \mathrm{round}(z)$, so $\mathrm{lnb}(p) \geqslant \mathrm{lnb}(z)$. Therefore

$$\begin{aligned}
\mathrm{lnb}(r) &\geqslant \mathrm{lnb}(z),\\
\mathrm{lnb}(r) &\geqslant 2^{e_x+e_y+2-2m},\\
\mathrm{lnb}(r) &\geqslant 2^{e_x+e_y+1} \cdot 2^{1-2m},\\
\mathrm{lnb}(r) &\geqslant 2u^2 \cdot \mathrm{ufp}(p),
\end{aligned} \tag{5.5}$$

According to our hypothesis, $e_p = e + m$, so $ufp(p) = 2^{e+m}$ yields in Relations (5.3) and (5.5):

$$r \leqslant u \cdot 2^{e+m} = 2^{e+m-m} = 2^e.$$

So we derive

$$ufp(r) \leqslant 2^e;$$

and

$$\begin{aligned} lnb(r) &\geqslant 2u^2 \cdot 2^{e+m}, \\ lnb(r) &\geqslant 2u \cdot u \cdot 2^{e+m}, \\ lnb(r) &\geqslant 2u \cdot 2^{-m} \cdot 2^{e+m}, \\ lnb(r) &\geqslant 2u \cdot 2^e. \end{aligned}$$

□

Hence, the exact accumulation of the dot product only require one exponent extraction for every partial product $x_i \times y_i$.

5.2.2 *Parallel Dot Product*

Our parallel algorithm for correctly rounded dot product is very similar to the summation one presented in Figure 4.3. The only difference is that it performs a TwoProd operation before accumulating the result and the error according to the result exponent as presented in Figure 5.1 for the sequential version. As for summation, the parallel correctly rounded dot product algorithm consists in the 3 steps presented in Figure 5.2.

STEP 1 In the first step, both input vectors x and y are split equally between threads. Each thread performs an error free transformation for its local dot product: Algorithm TwoProd replaces the floating-point multiplication. The multiplication result and its error are accumulated into C' according to the result exponent. Note that for small input vectors ($\leqslant$ 10000 according to our experiments) this accumulation according to exponent can be skipped and the multiplication result and error are stored without accumulation to C'. This avoids performing additional

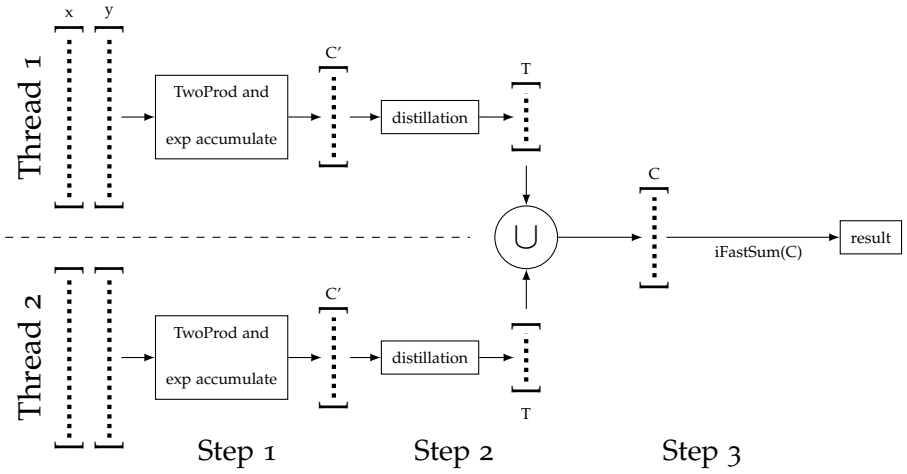


Figure 5.2: Parallel algorithm for correctly rounded dot product (input vectors x and y are distributed to every thread, here 2)

operations that do not reduce the size of the input data. Note that the size of C' is 4096 `binary64` number. Therefore it fits at least in L2 cache on recent processors (Intel processors based on Haswell or more recent architectures have at least 256 KB cache while 4096 `binary64` occupies 32 KB). A similar transformation relying on `HybridSum` could be used. However according to the running time for our test environment `OnlineExact` transformation is more efficient.

STEP 2 In the second step we perform a distillation of the result of Step 1 to transform the vector C' to the vector T that requires the smallest possible memory space to store the local dot product without error.

STEP 3 Finally, the local vectors T are gathered into a single vector C. The sequential iterative correctly rounded summation algorithm `iFastSum` is used to compute the sum of C.
Since all the performed transformations are error free, applying a correctly rounded summation to C provides the correctly rounded result of the dot product.

5.2.3 *Runtime Performance Results*

We consider again the environments presented in Table B.1. As for summation, we test our sequential implementation and CPU parallel version on environment A (workstation). A Xeon Phi accelerator version is tested on environment B. Additionally, we consider environment C which corresponds to a supercomputer with distributed memory[1]. Those test environments are significant of today's practice of floating-point computing. OpenMP library is used to implement the shared memory parallel dot product for both CPU and Xeon Phi. A hybrid MPI/OpenMP version is implemented for environment C. The OpenMP library is used for multithreading on the same socket and MPI is used to exchange data between different sockets.

We compare the performance results of our dot product to the highly optimized Intel MKL library and to implementations relying on 1-Reduction algorithm used by the ReproBLAS library [42]. We had to implement our version of the 1-Reduction dot product since ReproBLAS library does not offer neither OpenMP, nor hybrid OpenMP/MPI versions. We take care that the same libraries are used for all algorithms for the comparison to be fair. We also use the generator from [44] and explained in Section B.4.1 to generate vectors of different sizes and a same condition number to ensure that the presented performance results only depend on the input data size. However, according to our experiments, the condition number would have no impact on the running time of the considered algorithms.

Algorithms ReprodSum and FastReprodSum are not included in those tests because we have exhibited in the previous chapter that 1-Reduction is more efficient. The CNR feature [65] is also not used here since it does not aim at ensuring the reproducibility between sequential and parallel runs even on the same processor. We study this feature in details in Appendix A.

In Figure 5.3 we present the running time of different dot product implementations for CPU based sequential and shared mem-

1 https://www.cines.fr/calcul/materiels/occigen/

ory parallel systems, and for Xeon Phi accelerator. We show on the X-axis the log10 of the input vector size. The Y-axis shows the running time normalized by the Intel MKL dot product one.

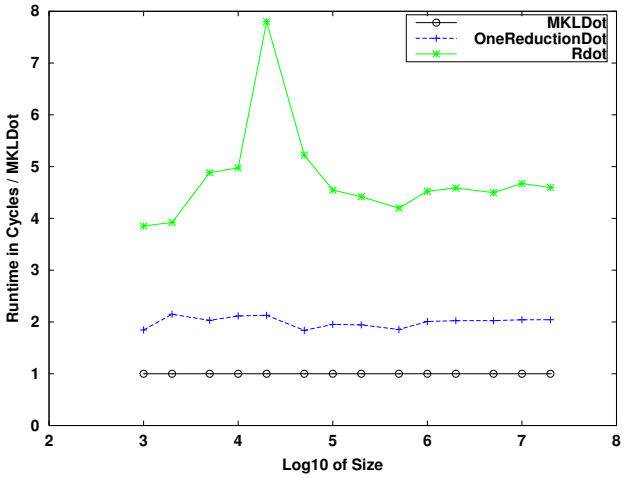
Figure 5.3a shows that our sequential correctly rounded dot product runs about 5× slower than Intel MKL dot product. On the other side, 1-Reduction dot product is only twice slower compared to MKL one.

When we run the CPU parallel version using all the available cores, Figure 5.3b shows that both our solution and the 1-Reduction dot product reach the same performance as Intel MKL, the latter being restricted by memory bandwidth.

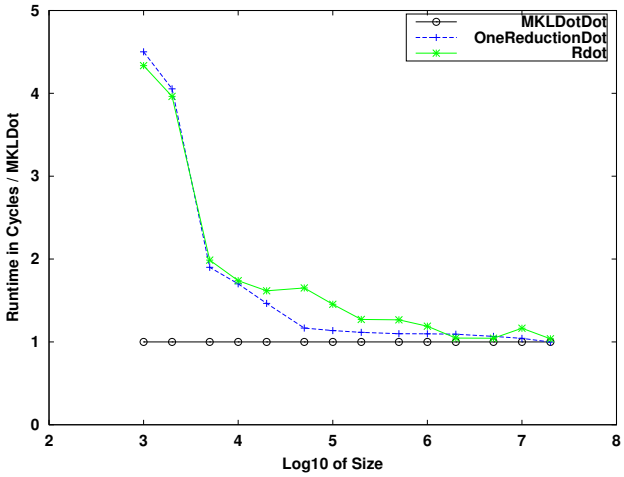
We show in Figure 5.3c that for Intel Xeon Phi accelerator, our correctly rounded dot product runs about 4× slower than both 1-Reduction and MKL solutions. As we explained for summation, our algorithm suffer from exponent driven operations that can not be vectorized efficiently. Therefore AVX-512 floating-point units available on Xeon Phi are mostly used to perform scalar operations. This limits considerably the performance of our dot product on Xeon Phi accelerator.

Results for large distributed memory system performed on Occigen supercomputer (environment C) are presented in Figure 5.4. Note that only dot product was tested on this architecture due to access time limitation to this supercomputer. We run the three previously presented dot products on the same data set (fixed size and condition number) with different hardware configurations.

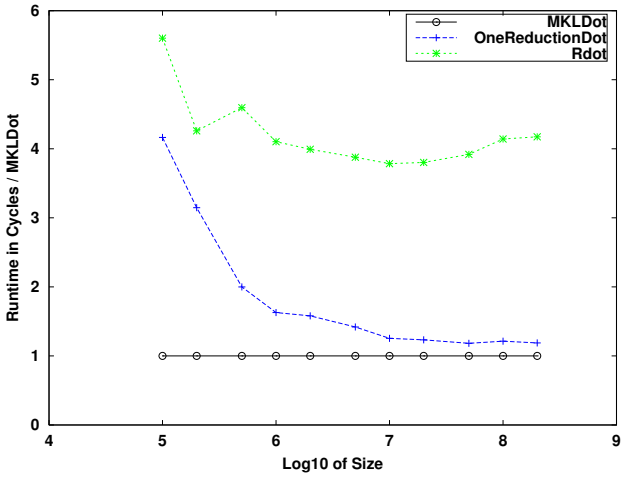
Figure 5.4a shows the scalability for the single socket configuration. On the X-axis we show the number of used threads, and the running time in cycles on the Y-axis we show. It is not a surprise that MKL dot product does not scale so far since it is quickly limited by the memory bandwidth. 1-Reduction and our correctly rounded dot product scale well up to exhibit no extra-cost ratio compared to MKL. Again such scaling occurs



(a) Sequential version

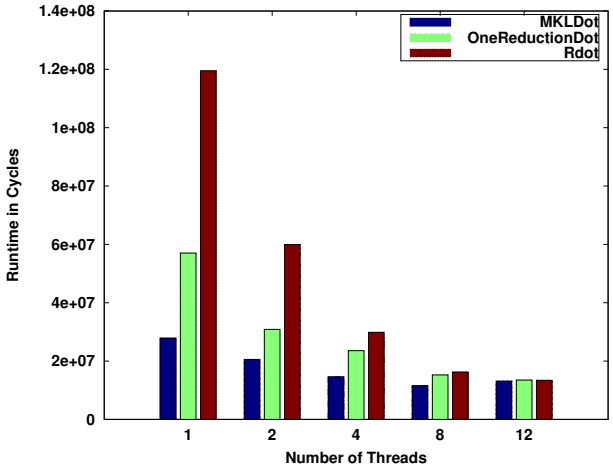


(b) CPU version (16 threads)

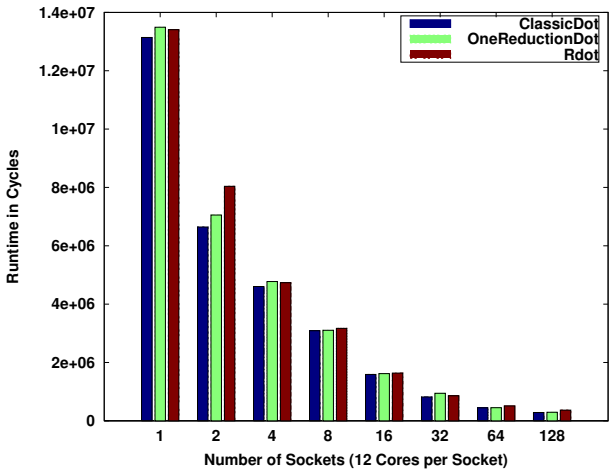


(c) Xeon Phi version (240 threads)

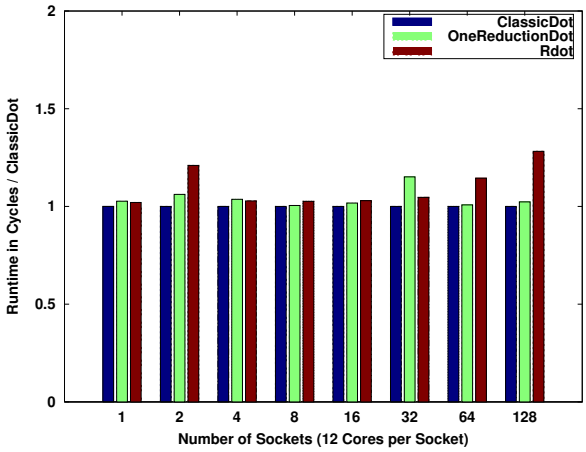
Figure 5.3: Performance results for dot product on shared memory systems (Cond = 10^8)



(a) Single socket scaling



(b) Multi-socket scaling



(c) Multi-socket scaling (Normalized to ClassicDot)

Figure 5.4: Performance results for dot product on distributed memory system (size = 10^7, Cond = 10^{32})

until being limited by the memory bandwidth.

Figures 5.4b and 5.4c show the running time of different dot products for multi socket configuration. For the classic optimized dot product we use the OpenMP implementation provided by MKL to compute the (socket) local dot, then a MPI reduction is used to compute the final result. On the X-axis we show the number of used sockets (all cores are used on each socket). On the Y-axis, we show in Figure 5.4b the running time in cycles and in Figure 5.4c the extra-cost compared to the classic dot product. Note that the same data is presented in different ways on these two figures. Both correctly rounded and 1-Reduction dots stay almost as efficient as classic dot. All algorithms have the same performance when performing local computations on a single socket as shown in Figure 5.4a. Since all algorithms perform a single additional communication in multi socket configuration, they all exhibit similar performances.

5.2.4 *Accuracy Results*

Accuracy results for dot product are shown in Figure 5.5. Previous versions are tested for different condition numbers. We show on the X-axis the log10 of the condition number, and on the Y-axis the log10 of the relative error compared to correctly rounded dot product computed with MPFR. In almost all cases, 1-Reduction dot besides being reproducible is more accurate than Intel MKL dot. However, for ill-conditioned problems both MKL and 1-Reduction derived implementation provide worthless results. As expected our dot product always exhibits correctly rounded results independently from the condition number of the problem.

5.3 EUCLIDEAN NORM

The euclidean norm of a vector p is defined as $(\sum_{i=1}^{n} p_i^2)^{1/2}$. The sum $\sum_{i=1}^{n} p_i^2$ can be correctly rounded either in sequential or in parallel by using the previous dot product to multiply the input vector p by itself. Afterwards, we apply a square root on the dot

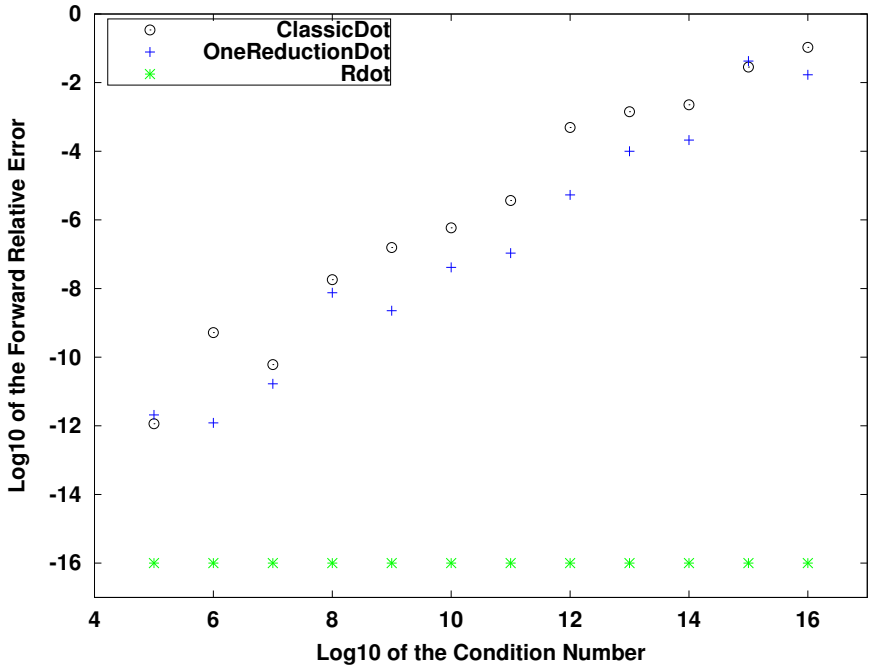
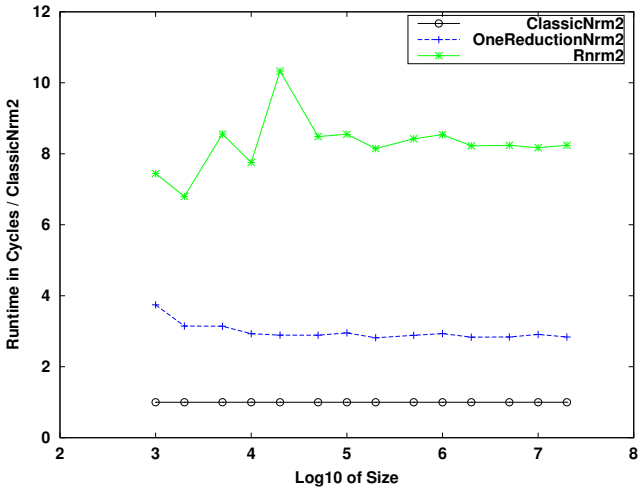


Figure 5.5: Accuracy results for dot product (size = 10^5)

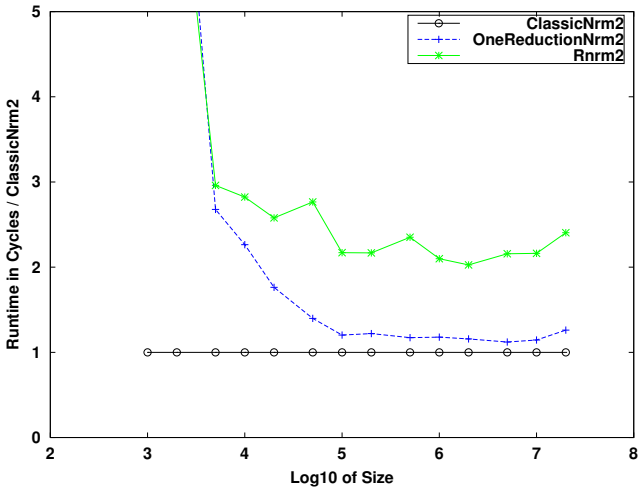
product result to yield a faithfully rounded euclidean norm as proven in [21]. This approach does not ensure that the final result of the euclidean norm is correctly rounded. However this faithfully rounded result is reproducible because it depends on a reproducible correctly rounded summation and on the reproducible IEEE-754 square root operator.

Note that we do not use directly the MKL implementation of nrm2 because it is implemented using a more complicated and slower algorithm: it evaluates the euclidean norm as $\max_i(p_i) \times (\sum_{i=1}^{n}(p_i/\max_i(p_i))^2)^{1/2}$ to avoid underflows and overflows and for a fair performance comparison. Since our solution does not manage to avoid underflows nor overflows, we use instead the dot product of MKL to compute the sum $\sum_{i=1}^{n} p_i^2$, and then a square root to return the expected result. We name it as the classic euclidean norm.
Performance results are presented in Figure 5.6.

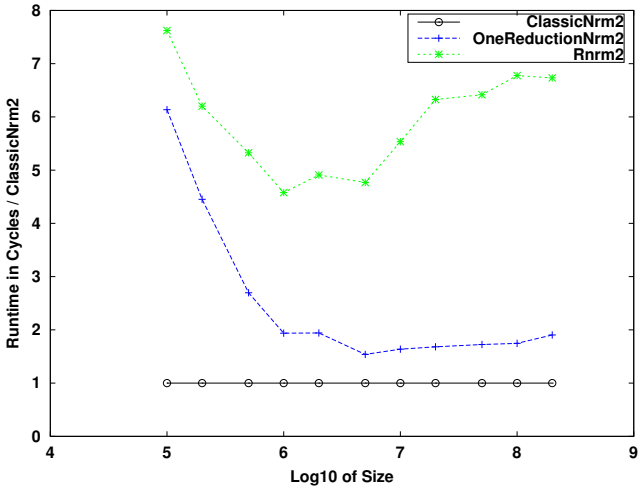
Except the final square root, algorithms that compute euclidean norm perform the same number of floating-point operations compared to their dot product counterparts for a given input



(a) Sequential version



(b) CPU version (16 threads)



(c) Xeon Phi version (240 threads)

Figure 5.6: Performance results for euclidean norm

vector size. Therefore the same running time is observed as the running time remains CPU bound. However, for classic euclidean norm which is memory bound[2] a significant performance boost is observed. It runs about twice as fast as its dot product counterparts since it reads twice less data from the memory.

Figure 5.6a shows that in the sequential case our reproducible faithfully rounded euclidean norm runs about 8× slower than the classic euclidean norm. The extra-cost is shown in Figure 5.6b to be about twice in parallel on CPU when all 16 cores are used. And Figure 5.6c shows that on Xeon phi accelerator our euclidean norm costs about 6× more compared to the classic one.

5.4 SUM OF ABSOLUTE VALUES

The sum of absolute values ($asum$) of a vector p is defined as $S = \sum_{i=1}^{n} |p_i|$. Our summation algorithm $Rsum$ presented in Chapter 4 can also be used to compute a correctly rounded sum of absolute values. However, condition number of $asum$ is known to equal 1. This justifies the use of the more runtime efficient compensated algorithm $SumK$ [44]. Since the relative error bound of $SumK$ is always greater than u (as shown in Relations (3.1) and (3.2)), its result can not be guaranteed to be correctly rounded. Nevertheless, this error bound is very pessimistic and occurs only in the worst case scenario. We propose an algorithm based on $Sum2$ ($SumK$ with $K = 2$) that evaluates the error bound during the computations to verify if the $Sum2$ result is correctly rounded or not. If the result is correctly rounded it is returned by our algorithm. Otherwise, the correctly rounded summation algorithm presented in Chapter 4 is used.

Algorithm 5.1 details our process to compute the correctly rounded sum of absolute values.

Lines 1-6 are nothing more than $Sum2$ (Section 3.4.2) with an additional variable $maxError$ that stores the maximum of the generated errors when accumulating into variable $\widehat{S}$. This allows

2 bottlenecked by the memory bandwidth

Algorithm 5.1 Correctly rounded sum of absolute values

Input: p: a vector of n floating-point numbers
Output: $\widehat{S}$: the correctly rounded sum of p.

1: $q = vecSum(|p|)$ {*$q_n = p_1 \oplus p_2 \oplus ... \oplus p_n$ and $\sum_{i=1}^{n} q_i = \sum_{i=1}^{n} |p_i|$*}
2: $maxError = 0$
3: **for** $(i = 1 : n-1)$ **do**
4: $\quad \widehat{S} = \widehat{S} \oplus q_i$ {*the sum of errors for the compensation*}
5: $\quad maxError = max(ulp(|\widehat{S}|)/2, maxError)$
6: **end for**
7: $errorBound = maxError \times (n-1)$
8: $(\widehat{S}, q_n) = TwoSum(\widehat{S}, q_n)$ {* compensation *}
9: $E^+ = roundUp(q_n + errorBound)$
10: $E_- = roundDown(q_n - errorBound)$
11: **if** $\widehat{S} \oplus E^+ = \widehat{S} \oplus E_-$ **then**
12: $\quad \widehat{S} = \widehat{S} \oplus q_n$
13: **else**
14: $\quad \widehat{S} = Rsum(|p|)$ {*algorithm presented in Chapter 4*}
15: **end if**

us to have a dynamic error bound for $\widehat{S} + q_n$, this error bound is computed by multiplying $maxError$ by the number of possible errors $(n-1)$. Note that this multiplication is exact for $n \leqslant u^{-1}$ since $maxError$ is a power of 2. Therefore we know that the exact result S is such that:

$$\widehat{S} + q_n - errorBound \leqslant S \leqslant \widehat{S} + q_n + errorBound. \tag{5.6}$$

We also know that $roundDown(q_n - errorBound) \leqslant q_n - errorBound$ and $q_n + errorBound \leqslant roundUp(q_n + errorBound)$, so after running instructions 9 and 10, this leads to:

$$\widehat{S} + E_- \leqslant S \leqslant \widehat{S} + E^+. \tag{5.7}$$

Since rounding is monotonic, we have

$$\begin{gathered} round(\widehat{S} + E_-) \leqslant round(S) \leqslant round(\widehat{S} + E^+), \\ \widehat{S} \oplus E_- \leqslant round(S) \leqslant \widehat{S} \oplus E^+. \end{gathered} \tag{5.8}$$

We distinguish two cases here

- If $\widehat{S} \oplus E_-$ and $\widehat{S} \oplus E^+$ round to the same floating-point value, any value in the interval $[\widehat{S} \oplus E_-, \widehat{S} \oplus E^+]$ including S have the same rounding. Hence the rounding of any point of the interval is a correctly rounded result of S. We return $\widehat{S} \oplus q_n$ that is the the correctly rounding of S in this case.
- If $\widehat{S} \oplus E_-$ and $\widehat{S} \oplus E^+$ do not round to the same floating-point value, we use algorithm Rsum to return a correctly rounded sum of absolute values.

Note that in practice, it is uncommon for $\widehat{S} \oplus E_-$ and $\widehat{S} \oplus E^+$ to have different roundings for this very well conditioned problem. This situation occurs only when the exact result is very close to the midpoint of two floating-point numbers. For more than 400 randomly generated vectors with size between 1000 and 10^8, the condition in line 8 of Algorithm 5.1 was always verified and there was no need to use the more complicated algorithm.

5.4.1 *Parallel Version*

A parallel version of algorithm Sum2 is introduced in [68]. It consists in two steps. (1) First we apply the sequential algorithm Sum2 on local data without performing the final error compensation. So we end with 2 floating-point numbers per thread. (2) Then we gather all these numbers in a single vector. Afterwards the master thread applies a sequential Sum2 on this vector (this same idea is used for vectorization).

We use this algorithm for our sum of absolute values in parallel. Furthermore, as in for the sequential algorithm we compute the error bound during the accumulation to test if the final result is correctly rounded or not. As before, when $\widehat{S} \oplus E_-$ and $\widehat{S} \oplus E^+$ round to different values, parallel Rsum is used to provide a correctly rounded asum.

Note that in parallel, $\widehat{S}$, q_n and maxError are not necessarily reproducible [53]. Therefore the values that depend on those latter, i.e. $\widehat{S}$, E_- and E^+ are also not necessarily reproducible. Therefore $\widehat{S} \oplus E_-$ and $\widehat{S} \oplus E^+$ could round to the same value in one run,

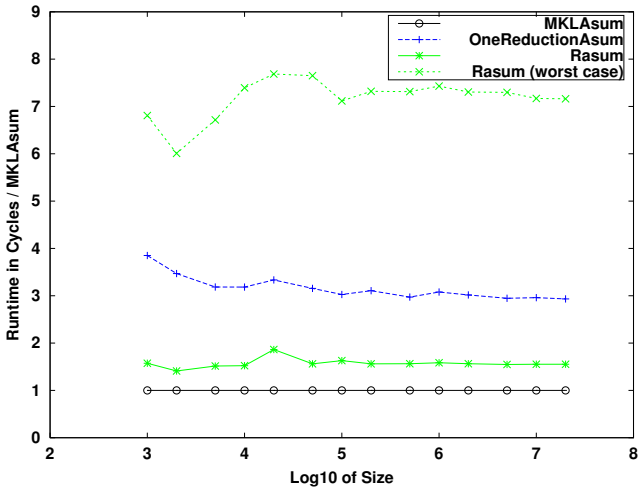
and to different values in another. However, in both cases a correctly rounded result is provided, either directly or using our Rsum algorithm. A running time difference could be observed even if reproducible results are provided. We recall again that this situation occurs very rarely in practice (only if the exact result is very close to the midpoint of two floating-point numbers). This reproducible asum is named Rasum in the following.

5.4.2 *Performance Results*

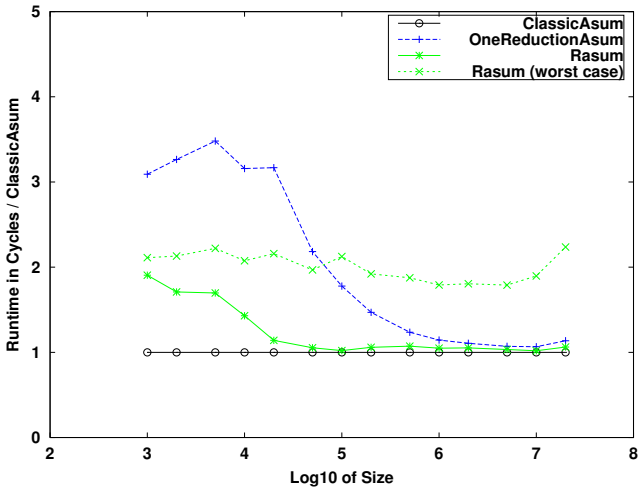
In Figure 5.7 we show running time results for the sum of absolute values. Our algorithm Rasum is compared to ClassicAsum which relies on MKL cblas_dasum, and to algorithm OneReductionAsum that relies on the OneReduction reproducible algorithm. In all subfigures, the X-axis shows the log10 of the input size, and the Y-axis the running time normalized by ClassicAsum. The performance of our solution depends on whether algorithm Sum2 is accurate enough to provide a correctly rounded result or not. Therefore we show performance for both cases.

Performance of the sequential implementation for CPU is shown in Figure 5.7a. If the correctly rounded result is provided by algorithm Sum2 (if the condition in line 8 of Algorithm 5.1 is satisfied), the extra-cost of our algorithm is less than twice compared to MKL implementation, and it runs even faster than OneReductionAsum. However, for the (rare) cases where Sum2 does not ensure that the result is correctly rounded, our more costly algorithm is used and the extra-cost grows up to 7× compared to MKL implementation.

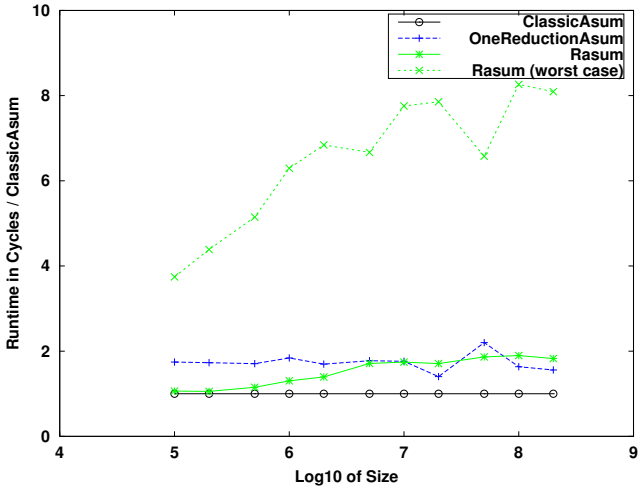
For the parallel case, Figure 5.7b shows that ClassicAsum exhibits the same behavior as the other memory bound routines and does not scale very well. For large vectors, both correctly rounded summation algorithm Rasum, and reproducible summation algorithm OneReductionAsum reach the performance of ClassicAsum. In the worst case scenario, our Rasum costs about twice compared to ClassicAsum based on MKL implementation.



(a) Sequential version



(b) CPU version (16 threads)



(c) Xeon Phi version (240 threads)

Figure 5.7: Performance results for sum of absolute values

Finally, for Xeon Phi accelerator, we show in Figure 5.7c that reproducible solutions run about two times slower compared to ClassicAsum. Our Rasum also runs about twice slower than ClassicAsum. However, the cost can go up to 8× in the worst case that relies on Rsum which is not efficient on this environment.

5.5 CONCLUSION

We have presented in this chapter algorithms for a correctly rounded dot product and sum of absolute values, and also a faithfully rounded one for the euclidean norm. Only level 1 BLAS has been addressed in this chapter and results have been presented on three different platforms. Our proposed algorithms ensure the reproducibility and the accuracy of results independently from the condition number of the input. The extra-cost of our solutions on CPU is up to 9× in the sequential case. However our algorithms scale better than MKL solutions and exhibit almost no extra-cost in the parallel case. Nevertheless on Xeon Phi accelerator, we observe that MKL optimized implementations are up to about 5× faster than ours. However, these algorithms are still useful for applications that could require reproducibility or accurate results.

6

REPRODUCIBLE LEVEL 2 BLAS

6.1 INTRODUCTION

This chapter focuses on level 2 BLAS. We address matrix vector multiplication ($gemv$) and triangular solver ($trsv$). Sequential and parallel correctly rounded algorithms for matrix-vector multiplication are introduced. For triangular solver unfortunately we do not ensure that the final result is correctly rounded. However we guarantee that it is reproducible independently from machine architecture and thread configuration, and that it provides more accurate results than the $trsv$ provided by MKL. As in previous chapters, running time and accuracy will be tested on the same computing environments.

6.2 MATRIX VECTOR MULTIPLICATION

Matrix-vector multiplication ($gemv$) is defined in the BLAS as $y = \alpha A \cdot x + \beta y$, where A is a matrix of size $m \times n$, y is a m-vector, x is a n-vector and α and β are two floating-point numbers. For a correctly rounded matrix-vector multiplication we need to ensure that $y_i = \alpha a^{(i)} \cdot x + \beta y_i$ is correctly rounded where $a^{(i)}$ is the i^{th} row of matrix A, and y_i is the i^{th} element of vector y.

6.2.1 *Sequential Correctly Rounded Algorithm*

Algorithm 6.1 details our proposed correctly rounded $gemv$. For every y_i and $a^{(i)}$ we distinguish three steps. (1) In the first step, we transform the dot product $a^{(i)} \cdot x$ into a sum of non-overlapping floating-point numbers stored in a vector T (line 1). This error-free transformation uses a minimum extra storage (< 39 numbers for $binary64$), and it is performed in different ways depending on the vector size. The transformation process presented in

Section 5.2.2 is used (steps 1 and 2 of parallel algorithm for dot product). (2) In the second step, we multiply all elements of T by α and y_i by β both using TwoProd. The results generated by TwoProd are stored in the vector T2 (lines 4-8). Note that the data is transformed with no error, and that

$$\sum_{j=1}^{size(T2)} T2_j = \alpha a^{(i)} \cdot x + \beta y_i.$$

with $size(T2) \leqslant 2 \times 39 + 2 = 80$. (3) Therefore, the last step consists in applying iFastSum to vector T2 to evaluate the final correctly rounded value of y_i.

Algorithm 6.1 Correctly rounded matrix-vector multiplication

Input: A: a matrix of size $m \times n$,
y: m-vector, x: n-vector, α, β: two scalars.
Output: $y = \alpha A \cdot x + \beta y$ (all y_i are correctly rounded).

1: **for** $(i = 1 : m)$ **do**
2: $a^{(i)} = A[i, 1 : n]$
3: $T = dotTransform(a^{(i)}, x)$ {* Step 1 *}
4: $K = size(T)$
5: **for** $k = 1 : K$ **do**
6: $(T2[k], T2[k + K]) = TwoProd(T[k], \alpha)$
7: **end for**
8: $(T2[2 \times K + 1], T2[2 \times K + 2]) = TwoProd(y_i, \beta)$
9: {* End of step 2 *}
10: $y_i = iFastSum(T2)$ {* Step 3 *}
11: **end for**

6.2.2 *Parallel Correctly Rounded Algorithm*

For matrix-vector multiplication, it is possible to design several algorithms according to the matrix decomposition. Three possible ones are: row layout, column layout and block decomposition. We opt for row layout decomposition because the error free transformation that we use for dot product is more efficient when working on large vectors. This choice also avoids the addi-

tional cost of reduction.

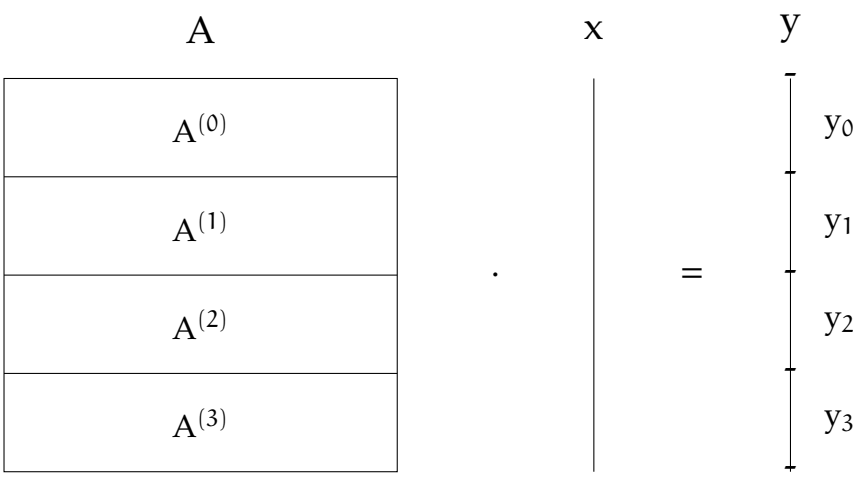


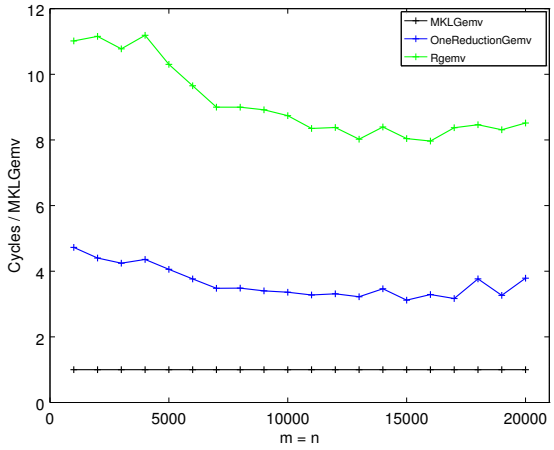
Figure 6.1: Parallel algorithm for correctly rounded matrix-vector multiplication (4 threads case)

Figure 6.1 shows how our parallel matrix-vector multiplication is performed. Vector x must be attainable for all threads. On the other side matrix A and vector y are split into t parts where t is the number of threads. Each thread handles the panel $A^{(i)}$ of A and the sub-vector $y^{(i)}$ of y. This latter is updated with $y^{(i)} = \alpha A^{(i)} \cdot x + \beta y^{(i)}$ as described in the sequential case.

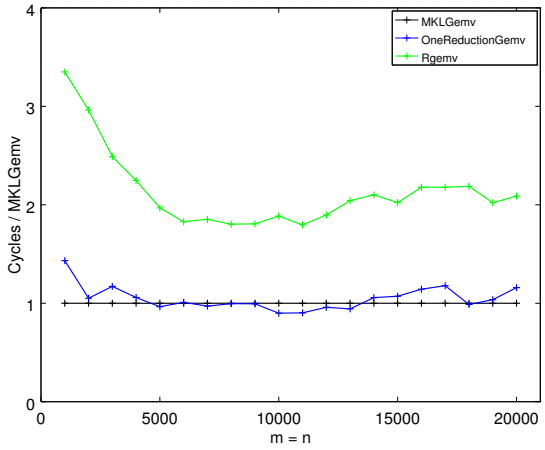
6.2.3 *Running Time Results*

As presented in Chapter 4 and 5 for sum and dot product, we generate data with different sizes and condition numbers. The used generator is presented in Section B.4.2. Data with fixed condition number $cond = 10^8$ and different sizes between 1000 and 20000 are generated for this test.

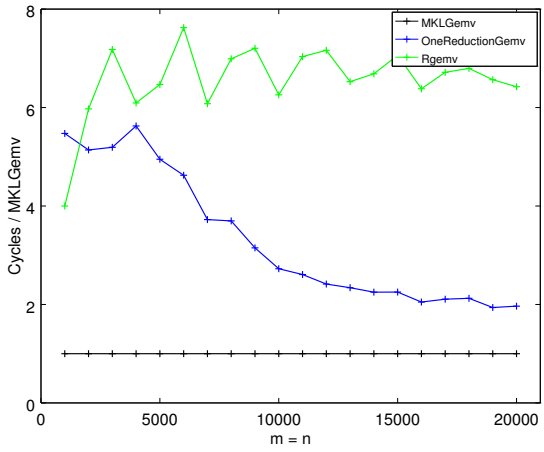
We perform our running time evaluations on environments A (workstation) and B (accelerator) presented in Table B.1. Running time of our correctly rounded matrix vector multiplication (*Rgemv*) is compared to the reference optimized MKL implementation (*MKLGemv*) and to one implementation we derived from 1-*Reduction* algorithm (*OneReductionGemv*) by performing all accumulations of the dot products $a^{(i)} \cdot x$ in an indexed



(a) Sequential version



(b) CPU version (16 threads)



(c) Xeon Phi version (240 threads)

Figure 6.2: Performance results for matrix vector multiplication (Cond $= 10^8$)

number (presented in Section 3.5.3). We parallelize OneReductionGemv in the same way as Rgemv (See Section 6.2.2).
Performance results are shown in Figure 6.2. We show on the X-axis the size of the input, while the Y-axis shows the running times normalized by MKL implementation.

Figure 6.2a shows that for large data sets, our sequential Rgemv is about 8× slower than MKL, while OneReductionGemv is only 3 to 4× slower.

Note that even at this BLAS level, the time spent loading data from the memory significantly dominates the one spent performing floating-point operations. Therefore, as for dot product (see Figure 5.4a) MKLGemv does not scale so far and the extra-cost of Rgemv is down to 2× in parallel case on CPU (environment A) and OneReductionGemv shows almost no extra-cost.

Since Rgemv relies on the same transformation used for Rdot, it suffers from similar performance issues on Xeon Phi accelerator. Figure 6.2c shows that it costs about 6× more than MKL implementation, while OneReductionGemv costs only 2 to 4× more compared to the same reference.

6.2.4 *Accuracy Results*

We focus here on the accuracy of each algorithm depending on the condition number of the input data. We generate data with condition number between 10^5 and 10^{16}, and with $m = n = 12000$ using the algorithm presented in Section B.4.2. Figure 6.3 shows on X-axis the log10 of the condition number, and on Y-axis the log10 of the relative error defined as $\| \widehat{y} - y \|_\infty / \| y \|_\infty$, where $\widehat{y}$ is the result provided by the tested algorithm and y is our reference result computed using MPFR library.

The accuracy of gemv results mainly depends on the accuracy of the dot product $a^{(i)} \cdot x$. Therefore we observe the same behavior as for dot product in Figure 5.5. Our Rgemv always ensures that the provided result is correctly rounded of course. MKLGemv accuracy depends on the condition number. Results

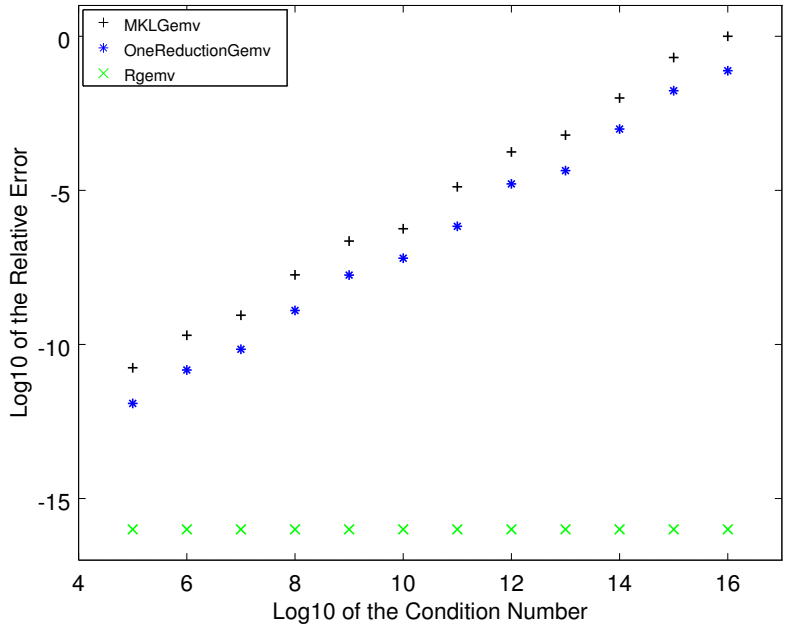


Figure 6.3: Accuracy results for matrix vector multiplication ($m = n = 12000$)

are acceptable for small ones, but have no significance for large ones. The same behavior is observed for $OneReductionGemv$ besides being reproducible and slightly more accurate than MKL.

We recall also that even in parallel, if row oriented decomposition is used for classic algorithms (as $MKLGemv$) the results are reproducible when the number of threads changes. However, this reproducibility is not guaranteed between different architectures. The CNR feature available on MKL can solve this problem in some cases, and it is studied in details in Appendix A.

6.3 TRIANGULAR SOLVER

Let T be a square triangular matrix of size n, and let b be a n-vector of floating-point numbers. Solving this triangular system is finding a n-vector x such that $Tx = b$. For the sake of simplicity we consider that T is a lower triangular matrix. The system $Tx = b$ can be solved using the following formula

$$x_i = (b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j)/t_{ii}, i = 1...n, \tag{6.1}$$

Note that to compute x_i, all x_j with $j < i$ should have been calculated. This creates a dependency between each element of x and the previous ones. Algorithm 6.2 presents the well known forward substitution process that solves a lower triangular system.

Algorithm 6.2 Forward substitution algorithm

Input: T: a lower triangular $n \times n$ matrix,
b: a n-vector of floating-point numbers.
Output: x: a n-vector of floating-point numbers.
Ensure: $Tx = b$

```
1: x_1 = x_1/t_11
2: for (i = 2 : n) do
3:    S = 0
4:    for (j = 1 : i − 1) do
5:       S = S + t_ij · x_j
6:    end for
7:    x_i = (b_i − S)/t_ii
8: end for
```

A parallel version of this algorithm has been presented in [25]. Their approach is to split T into blocks as shown in Figure 6.4. Each $trsv$ block depends on $gemv$ blocks in the same row, while each $gemv$ block depends on the $trsv$ block in the same column. $trsv$ block computations are performed using a serial algorithm. While $gemv$ every block is executed in parallel.

Another parallel algorithm is presented in [56]. It relies on sparse matrix multiplication to compute T^{-1}, then it multiplies this latter by the input b to compute the solution x. However we do not consider this algorithm because of the cost of matrix inversion. Note that this algorithm is more efficient for the level 3 BLAS function $trsm$ (solving multiple triangular systems using the same coefficient matrix).

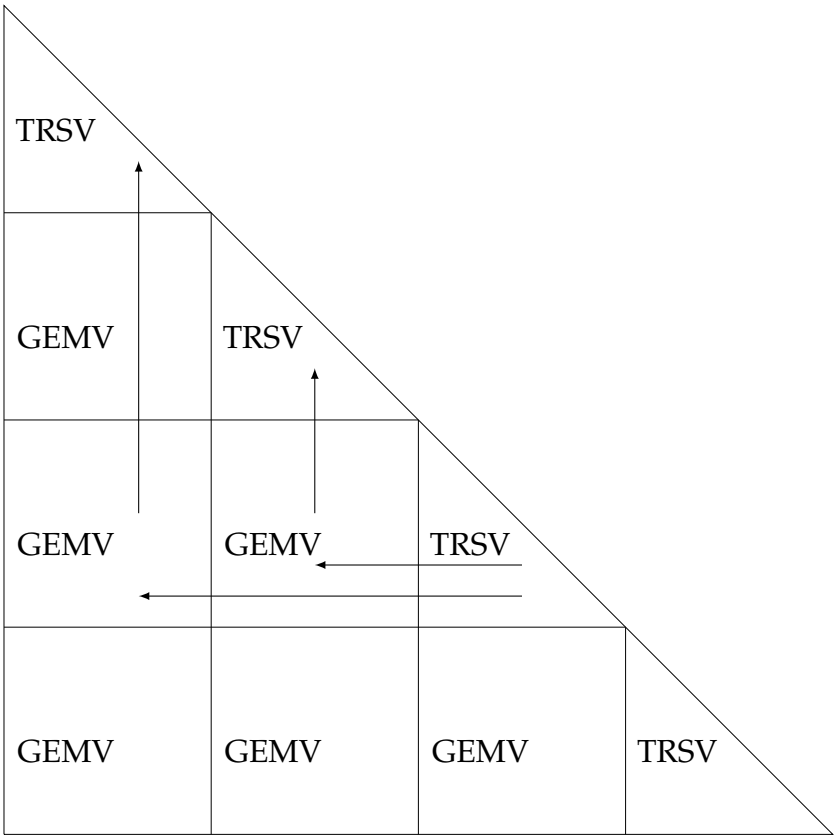


Figure 6.4: Parallel triangular solver and example of dependencies between *trsv* and *gemv*

Higham has shown in [24] that for a machine precision u, the backward substitution algorithm provides a solution $\widehat{x}$ such that

$$\frac{\| x - \widehat{x} \|_\infty}{\| x \|_\infty} \leqslant \frac{\gamma_n \cdot \mathrm{cond}(T, x)}{1 - \gamma_n \cdot \mathrm{cond}(T)}. \tag{6.2}$$

with

$$\gamma_n = \frac{nu}{1 - nu}.$$

Here n is the input system size, and the Skeel condition number [58] is defined as:

$$\begin{aligned} \mathrm{cond}(T, x) &= \frac{\| \, |T^{-1}| \cdot |T| \cdot |x| \, \|_\infty}{\| \, |x| \, \|_\infty}, \\ \mathrm{cond}(T) &= \| \, |T^{-1}| \cdot |T| \, \|_\infty \, . \end{aligned} \tag{6.3}$$

Higham also demonstrated that the error bound in Relation (6.2) does not depend on the order of the loop at the line 3 of Algo-

rithm 6.2 [24]. Therefore, either the sequential or the parallel version of this algorithm have the same error bound.

6.3.1 *Reproducible and Accurate trsv*

The main source of non-reproducibility in the forward substitution algorithm is the parallel evaluation of $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ when computing every x_i (Relation (6.1)). We compare three different algorithms here.

1. the first algorithm computes the correctly rounded result of $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ for each x_i;
2. the second computes $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ using a reproducible algorithm;
3. the last one also computes $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ using a reproducible algorithm, and applies an iterative refinement process to increase the result accuracy.

We present next those three algorithms in more details.

THE FIRST ALGORITHM consists in computing the correctly rounded result of $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$. In practice, either TwoProd or 2MultFMA (depending on FMA availability) is used for the multiplication $t_{ij} \cdot x_j$. Afterwards, the sum of all results and generated errors are accumulated according to their exponent as shown in Figure 5.2. We can rely either on HybridSum or on OnlineExact for this transformation.

In parallel, $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ is rounded to nearest in the same way. However, the process is split into two tasks as shown in Figure 6.5. For a block size r, a first part error-free transforms r dot products in parallel. The second part is a sequential trsv call builds on the previous error-free transformation to compute the correctly rounded value of $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$.

The block size r should be chosen very carefully such that all the accumulator vectors (one accumulator vector for each block row) fit at least in L3 cache. HybridSum is preferred here because it requires smaller accumulator vector for each row and

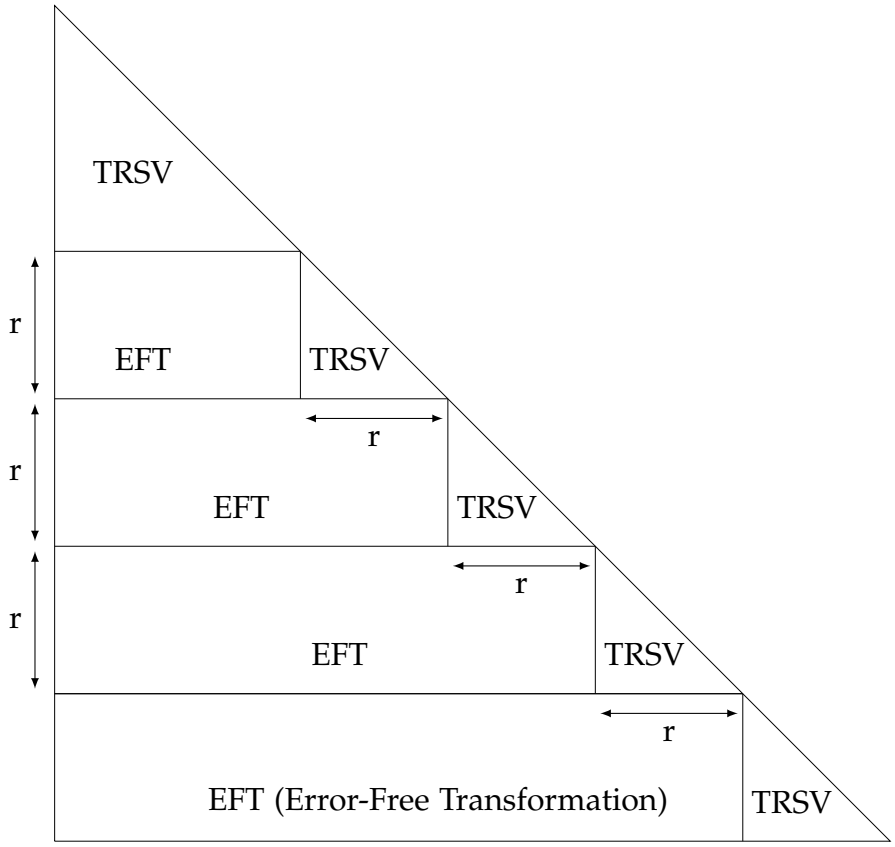


Figure 6.5: Parallel triangular solver relying on correctly rounded dot product

allows using larger blocks to better benefit from parallelism.

Iakymchuk and al. have used a similar process to implement a parallel reproducible `trsv` in [28] relying on floating-point expansions and superaccumulators instead of algorithm `HybridSum`.

THE SECOND ALGORITHM uses `1-Reduction` to evaluate a reproducible value of the sum $b_i - \sum_{j=1}^{i-1} t_{ij} \cdot x_j$ independently from the operation order. Algorithms `ReprodSum` and `FastReprodSum` are not used because they both require to compute the max of absolute values before evaluating the sum. Indeed this is impossible to implement efficiently in parallel since we have only a part of vector x in each call. On the other side, `1-Reduction` works properly even when it does not have all the input data as explained in Section 3.5.4.

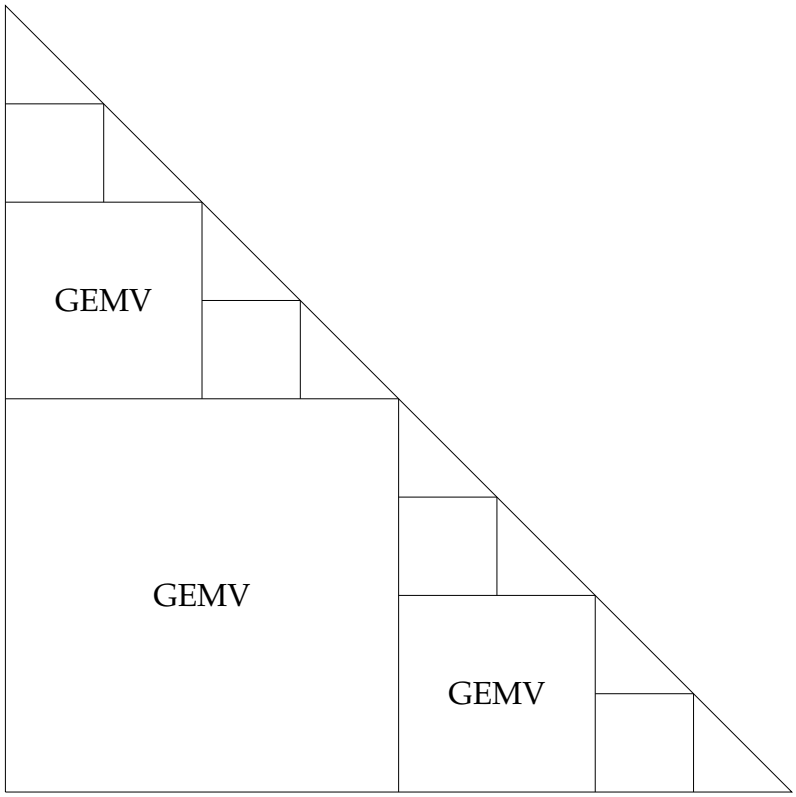


Figure 6.6: Recursive parallel triangular solver

Note that *trsv* is not yet provided in the current ReproBLAS [42]. So we had to implement and use our own *OneReductionTrsv*. We split recursively the matrix T as shown in Figure 6.6 to have larger parallel *gemv* blocks.

THE THIRD ALGORITHM relies on the previous one to solve the triangular system in a reproducible way, then it uses iterative refinement to improve the result accuracy. Iterative refinement consists in computing the following steps [24].

1. $\widehat{x} = solve(T, b)$.
2. Compute the residual $r = b - T\widehat{x}$.
3. $d = solve(T, r)$.
4. Update $\widehat{x} = \widehat{x} + d$.
5. Repeat from step 1 if $\widehat{x}$ is not accurate enough (depending on stop condition).

We evaluate the residual r with a higher precision to ensure more accurate results and faster convergence. Reproducibility of the matrix-vector multiplication in Step 2 should also be ensured to be reproducible, Otherwise this iterative process would in general converge to different solutions even if the solver is reproducible.

Algorithm 6.3 Reproducible iterative refinement

Input: T: a lower triangular matrix $n \times n$ floating-point numbers,
b: a n-vector of floating-point numbers.
Output: $\widehat{x}$: a n-vector of floating-point numbers.
Ensure: $T\widehat{x} \approx b$

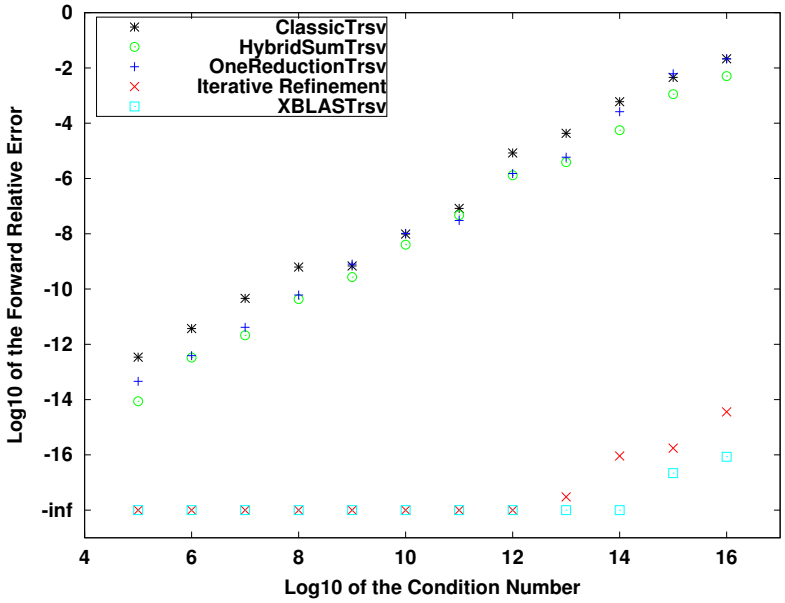
1: $\widehat{x} = OneReductionTrsv(T, b, K = 2)$
2: $i = 0$
3: $dx^0 = \infty$
4: **repeat**
5: $\quad i = i + 1$
6: $\quad r = OneReductionGemv(A, -\widehat{x}, b, K = 3)$
7: $\quad dx^i = OneReductionTrsv(T, r, K = 2)$
8: $\quad \widehat{x} = \widehat{x} \oplus dx^i$
9: **until** $2 \parallel dx^i \parallel_\infty > \parallel dx^{i-1} \parallel_\infty$ OR $\parallel dx^i \parallel_\infty < 2u \parallel \widehat{x} \parallel_\infty$

We use a 1-Reduction based dot product for both the solver and the residual evaluation. For the solver 1-Reduction is used with $K = 2$, while for the residual $K = 3$ is chosen. TwoProd is used to perform multiplications to avoid multiplication error and improve accuracy of the residual r. Algorithm 6.3 details our reproducible iterative refinement. with a stop condition inspired from [15].

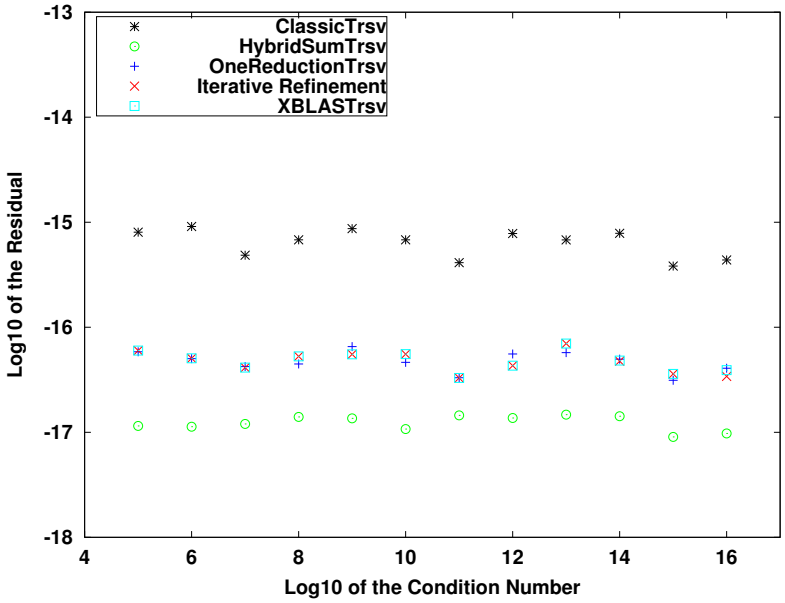
6.3.2 *Accuracy Results*

The accuracy of numerical results produced by previously presented algorithms is studied in this section. We use the triangular system generator presented in Section B.4.3 to generate data with fixed size and different condition numbers. We show here the impact of the system condition number on the solution accu-

racy and on the residual.



(a) Direct relative error



(b) Residual relative to b

Figure 6.7: Accuracy of triangular solvers (size = 1000)

Figure 6.7 compares the different proposed solutions with the double-double implementation of trsv provided by XBLAS [1] (routine BLAS_trsv_x with extended precision). We also included the solver from XBLAS library. In Figure 6.7a we show on the X-axis the log10 of the Skeel condition number defined in Re-

lation (6.3). The Y-axis shows the log10 of the relative error for the calculated solution $\widehat{x}$ compared to the solution $\tilde{x}$ obtained by a MPFR relying solver. The relative error is computed as $\| \widehat{x} - \tilde{x} \|_\infty / \| \tilde{x} \|_\infty$.

Note that HybridSumTrsv, OneReductionTrsv and our iterative refinement algorithm all ensure reproducible results independently from the running environment. We focus more on their accuracy in this section. As expected the relative error for MKL implementation grows linearly with the condition number. HybridSumTrsv computes the correctly rounded result of the intermediate dot products, and it manages to improve slightly the accuracy of the result, yet exhibiting the same behavior than MKLTrsv. Its relative errors grows linearly with respect to the condition number. Since the dot product that relies on OneReduction is more accurate than the classic dot product we observe that OneReductionTrsv is also more accurate than MKLTrsv and less accurate than HybridSumTrsv. For condition numbers smaller than 10^{12}, both XBLASTrsv and our iterative refinement algorithm compute the same result as the MPFR solver. For larger condition numbers, our iterative refinement solver is slightly less accurate than XBLAS, yet it provides very accurate results compared to other algorithms.

Figure 6.7b shows on the X-axis the residual normalized by b. The normalized residual is computed as $\| r \|_\infty / \| b \|_\infty$. The condition number of the system seems to have no effect on the residual for all algorithms. MKLTrsv has the larger residual. Surprisingly, the residual of HybridSumTrsv is smaller than the one of XBLASTrsv whereas these latter provide more accurate results. and iterative refinement algorithm that provide more accurate results. It is well known that smaller residual does not imply higher accuracy [20]. For a computed solution $\widehat{x}$ a small residual means that $A\widehat{x}$ predicts accurately the right hand side b, while a higher accuracy means that the difference between the computed solution $\widehat{x}$ and the exact solution x is smaller. Note that for two computed solutions $\widehat{x_1}$ and $\widehat{x_2}$ it is possible that $\| \widehat{x_1} - x \|_\infty > \| \widehat{x_2} - x \|_\infty$ and $\| A\widehat{x_1} - b \|_\infty < \| A\widehat{x_2} - b \|_\infty$.

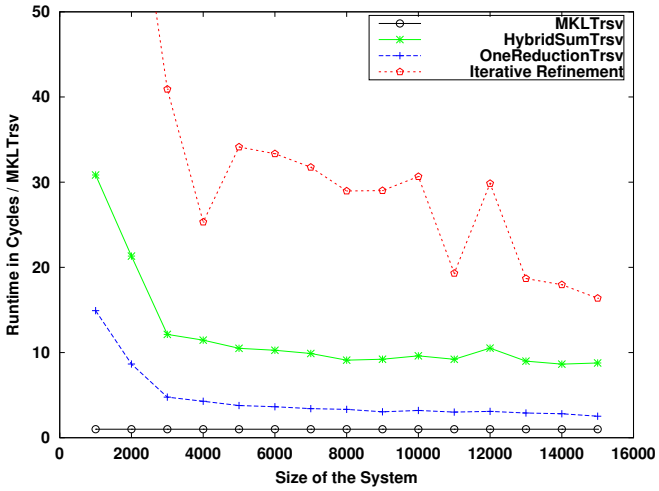
6.3.3 *Performance Results*

Running time for the different solvers is presented in Figure 6.8. Tests are performed with environments A and B (workstation and accelerator in Table B.1). In all subfigures we show on X-axis the size of the triangular system and on the Y-axis the running time normalized by those of the optimized MKLTrsv solver.

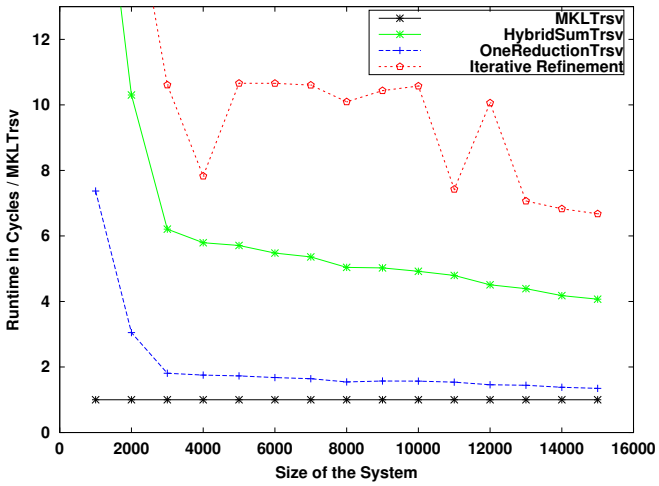
Figure 6.8a shows the running time for the sequential versions of tested solvers. OneReductionTrsv is about twice slower than MKLTrsv while ensuring reproducible and slightly more accurate results, and also smaller residual. Algorithm HybridSumTrsv is about 9× slower. However it ensures much smaller residuals and slightly more accurate solutions. And finally the iterative refinement algorithm is about 20-30× slower in the sequential case (depending on the number of performed iterations) while it provides the same results as the MPFR relying solver for condition numbers smaller than 10^{12} and ensures a decent accuracy for larger condition numbers.

We show in Figure 6.8b that the extra-cost compared to MKLTrsv is lower for all algorithms. HybridSumTrsv costs about 4-6× more depending on input size, OneReductionTrsv has almost no extra-cost for input sizes larger than 3000, while the iterative refinement algorithm costs about 7-10× more than MKLTrsv.

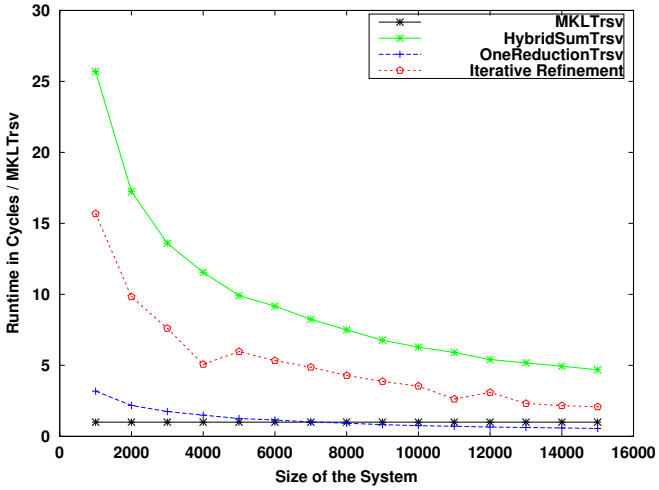
Performance results on Xeon Phi accelerator are also shown. We admit first that the trsv problem is not suitable for accelerators (including Intel Xeon Phi) due to dependencies and sequential computations so introduced. Note that Xeon Phi accelerator is not designed to perform sequential computations due to its very slow clock speed and its architectural design: even Xeon Phi one single core can not execute instructions for the same thread in adjacent cycles [49, 50]. Therefore at least two threads per core should be used to maximize performance. In short, the sequential part in the triangular solver will limit extensively the performance with Xeon Phi which it is mainly designed to run multi-threaded wide SIMD vector applications.



(a) Sequential version



(b) CPU version (16 threads)



(c) Xeon Phi version (240 threads)

Figure 6.8: Performance results for triangular solver (Cond = 10^8)

In practice, the parallel triangular solver is faster on CPU (environment A) than on Xeon Phi. However some applications that are suitable for accelerators use triangular solver as an intermediate step. For example, a linear solver might rely on BLAS 3 operation that are suitable for accelerators to perform a LU or QR factorization, then switch through triangular solver to compute the solution. Therefore, we thought that it is important to provide a reproducible triangular solver for Xeon Phi.

Figure 6.8c exhibits the running time for different solvers. All solvers are very close to each other in terms of running time. HybridSumTrsv is the slowest algorithm here being about 5× slower than MKLTrsv in the best case. OneReductionTrsv compares differently to MKL implementation depending on the input size. For small to medium size inputs ($\leqslant$ 8000) OneReductionTrsv has about 1-2× extra-cost, while for larger inputs it is up to 30% faster than MKL implementation. Unfortunately, MKL is not an open source library and the used algorithm is not public. Therefore we can not explain this performance drop of MKL triangular solver when running on Xeon Phi. The extra-cost of our iterative refinement solver is about 3-5× which is acceptable in most cases, especially when we consider the accuracy gain.

6.4 CONCLUSION

In this chapter we presented algorithms for reproducible and accurate level 2 BLAS functions. For matrix-vector multiplication we introduced Rgemv which is an algorithm that relies on error-free transformations to ensure correctly rounded results. For triangular solver, several solutions are compared, One can either use a solver based on error-free transformations that ensures small residual or an algorithm relying on OneReduction algorithm with iterative refinements that is almost as accurate as XBLAS solver. In terms of running time, we have shown that the cost of correctly rounded *gemv* is about twice compared to MKL implementation in the parallel case on CPU. In the sequential case and on Xeon Phi accelerator the extra-cost is higher and can be justified only if we deal with ill conditioned problems or in applications where the numerical reproducibility is an impor-

tant requirement. For trsv, the two algorithms that we proposed are compared to MKLTrsv. HybridSumTrsv costs about $5-10\times$ more than MKL implementation either on CPU or Xeon Phi accelerator and does not considerably improve accuracy. However, relying on the reproducible OneReduction summation and enhancing the resolutions with iterative refinement ensures accurate results and shows very promising running times on test environments. Its extra-cost in parallel varies from 3 to $10\times$ depending on the environment.

7

CONCLUSION AND FUTURE WORK

Obtaining reproducible result on a massively parallel environment is not an easy task, and it has been listed as one of top ten challenges for exascale [38]. We have presented some solutions to improve numerical reproducibility of HPC software. We suggest to implement a reproducible version of BLAS. These routines are widely used in scientific applications since optimized implementations for different platforms are available. Non-reproducibility of BLAS numerical results rises mainly because of the non-associativity of floating point addition. The order of floating-point operations change in parallel systems because of dynamic scheduling, non-deterministic reductions, varying number of threads depending on resource availability and heterogeneous architectures. Therefore, numerical results might change from one run to another.

We recall that our goal was to implement a reproducible BLAS library that ensures the following properties:

- Bitwise Numerical Reproducibility: result should be reproducible up to the last bit.
- Maximum Accuracy: result is ensured to have less error-bound than classical BLAS.
- Running Time Performance: extra-cost should be acceptable in practice in respect to optimized but non reproducible BLAS.

Numerical results provided by our BLAS implementation should not depend on thread configuration, nor on processor architecture. Implementing reproducible BLAS library would help developers to easily ensure that their applications behave in the same way independently from the running environment.

We exploit error-free transformations and efficient recent summation algorithms to provide very accurate results. For level 1 BLAS, the dot product and the sum of absolute values are correctly rounded. Our euclidean norm is ensured to be reproducible while being only faithfully rounded besides being . For level 2 BLAS we proposed a correctly rounded matrix-vector multiplication and a parallel reproducible triangular solver which is almost as accurate as the one provided by XBLAS which use extended precision (double-double format).

We use different approaches depending on the addressed routine. Error-free transformations and correctly rounded summation algorithms are used for dot product, euclidean norm and matrix vector multiplication. Our reproducible and correctly rounded sum of absolute values relies on the compensated summation algorithm *Sum2*. For triangular solver, we implement a reproducible iterative refinement process to ensure both numerical reproducibility and accuracy. All used algorithms are carefully optimized for each of the presented test environments. Since we introduce additional floating-point operations to ensure numerical reproducibility and accuracy, it is expected for our library to run slower than optimized BLAS implementations.

	Sequential	Shared memory	Distributed memory	Xeon Phi
sum	4	1	N.D.	6
dot	4.5	1	1	4
nrm2	8	2	N.D.	7
asum (worst)	1.5 (7)	1 (2)	N.D.	2 (8)
gemv	8	2	N.D.	7
trsv	25-30	8-10	N.D.	3-4

Table 7.1: Extra-cost ratio of our reproducible and accurate solutions

In table 7.1 we synthesize the extra-cost of our accurate and reproducible summation and BLAS implementations compared to classic optimized ones[1]. It varies between no extra-cost up

1 Either MKL implementation or ones that rely on MKL subroutines

to 30× slower depending on the running environment and the subroutine. The main factor that influences the running time of our BLAS is the summation algorithm. The recent summation algorithm proposed by Neal [41] is not studied in details here. However, it relies on the same principle as HybridSum and OnlineExact (accumulating the inputs with the same exponent), and is expected to exhibit the same running time properties (slightly more or less efficient depending on the environment).

The practical usability of our library depends on applications. In cases where the additional cost is low enough to be acceptable for the user our implementation can easily replace non-reproducible ones. At the contrary, in cases where the extra-cost is very high, it will be harder to replace optimized BLAS even if they are not reproducible. However, our solutions can be used for debugging or validation steps. Note also that reproducibility might be an important requirement for some applications even if the extra-cost is high, especially if the running time of the part that raises non-reproducibility issue is negligible compared to the full running time of the application.

7.0.1 *Future Work*

Comparing and studying summation algorithms and optimizing them carefully at a very low level for each architecture is a very hard and time consuming process. Designing an auto tuning process that compares several implementations for summation algorithms and select the most efficient one during the installation step would be helpful to identify what is the best efficient implementation for each architecture.

The current version of our library is limited to level 1 and 2 BLAS. It runs on Intel CPUs and Intel Xeon Phi accelerator only. Ensuring numerical reproducibility on each platform that supports IEEE-754 standard requires providing an implementation for other architectures, and also extending our solutions to level 3 BLAS. Unfortunately our approach is not suitable for this level. The need for extra memory to compute a correctly rounded dot

product prevents us from using tiling efficiently. Therefore, a different approach should be developed for level 3 BLAS and even higher level LAPACK routines as matrix decompositions and linear solver.

Part III

APPENDICES

A

ABOUT INTEL MKL CONDITIONAL NUMERICAL REPRODUCIBILITY

A.1 INTRODUCTION

We present and analyze the CNR feature (Conditional Numerical Reproducibility) available for Intel MKL library since release 11.0 [33]. The CNR feature does not aim to solve the problem of reproducibility between sequential and parallel versions of the same program. According to Intel's technical report [65] the number of threads must remain the same from run to run for the results to be consistent. Although, it solves the reproducibility problem when a software that uses MKL runs on two different architectures. We study this feature and how it impacts performance.

A.1.1 *How to Use CNR Feature*

By default Intel MKL library uses versioning to identify the code based on instruction set supported by the host processor. CNR feature allows the user to manually select the code to run for used MKL functions. Instruction sets supported by MKL are SSE2, SSE3, SSSE3, SSE4_1, SSE4_2, AVX, AVX2. These instruction sets are ordered according to their release date and are also inclusive. One Intel processor that supports a recent instruction set also supports the older ones.

CNR specifies the instruction set to be used in two ways. Either via the environment variable "MKL_CBWR" (export MKL_CBWR = "instruction set"), or in the source code by calling function "mkl_cbwr_set" before calling other MKL functions.

When CNR is turned off MKL functions might use dynamic scheduling and non-deterministic reductions. In this case results can be non-reproducible even if the thread configuration and the

instruction set is not changed. When MKL_CBWR value is set to "AUTO", the code corresponding to the most recent instruction set is selected. Deterministic reductions and static data scheduling are also used [59]. Therefore, the order of floating-point operations is guaranteed to be the same when the same number of threads is used on the same processor or other processors that support the same instruction set.

To ensure reproducibility between different Intel processors that have different instruction sets the value of MKL_CBWR should be set to a value that corresponds to one instruction set supported by every processor. To ensure reproducibility on all Intel architecture compatible processors (even non Intel processors), the value of MKL_CBWR should be set to "COMPATIBLE" [65].

A.2 REPRODUCIBILITY AND PERFORMANCE

We present in this section numerical and performance results for BLAS implemented by MKL. Tests have been performed on two different processors with two different instruction sets (AVX and AVX2). Three subroutines from different BLAS levels have been addressed: dot product (level 1 BLAS), triangular solver (level 2 BLAS) and matrix-matrix multiplication (level 3 BLAS).

A.2.1 *Test Methodology*

We perform our tests on both environments A and D presented in Table B.1. The CPUs from those environments are based on two different architectures and support two different instruction sets. On each processor we test all possible values for "MKL_CBWR" except "AUTO" because it is equivalent to AVX on environment A, and to AVX2 on environment D. And we exclude also the deprecated "SSE3".

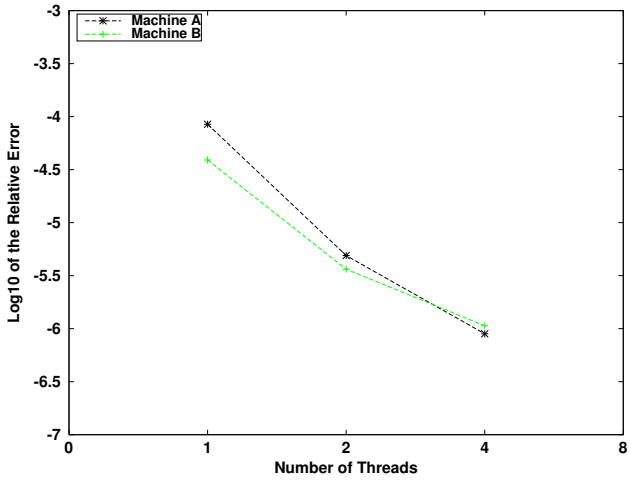
A.2.2 *Numerical Results*

We present in Figures A.1 and A.2 the numerical results of three functions: `cblas_ddot`, `cblas_dtrsv` and `cblas_dgemm`. Generators presented in appendix B are used to yield data. However, condition number has no effect on efficiency of MKL implementation (condition number only affect the accuracy of MKL results).

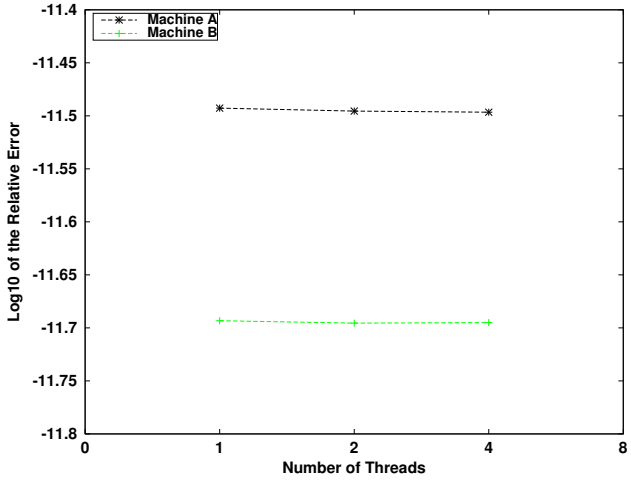
In Figure A.1 we show the test of BLAS functions with CNR turned off, while Figure A.2 shows the numerical results for the same functions when MKL_CBWR is set to *AVX*. We opt for *AVX* because it is the most recent instruction set supported by both processors. For the subfigures, we show on the X-axis the number of used threads. On the Y-axis we show the `log10` of the relative error compared to the reference result computed using MPFR library [62]. On Figures A.1a, A.1b and A.1c we see that all three functions provide different results on different processors when CNR is turned off. This is due to the dispatcher that selects a code based on the instruction set *AVX* for environment A, and a code based on *AVX2* for environment D. However, when MKL_CBWR is set to *AVX*, both test environments give the same numerical results when the same number of threads is used as shown in Figures A.2a, A.2b and A.2c. Any other value except *AVX2* (because it is not supported on processor A) should ensure that both processors get the same results. Although, the choice impacts performance as we will show in section A.2.3. It is not surprising that the most recent supported instruction set ensures the best performance.

In Figure A.3 we show that reproducibility is not always guaranteed when we change number of threads. Therefore, the CNR feature can not be used to ensure reproducibility between the sequential and the parallel versions of a program.

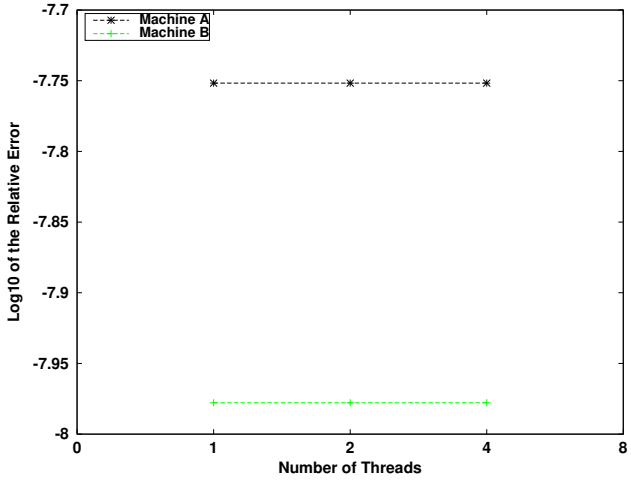
Note that for triangular solver and matrix multiplication it is possible for the parallel version to run dependant operations in the same order as on the sequential version. Figure A.3c shows that results are reproducible when the code based on *AVX* or



(a) Dot product. Cond = 10^8, Size = 10^6

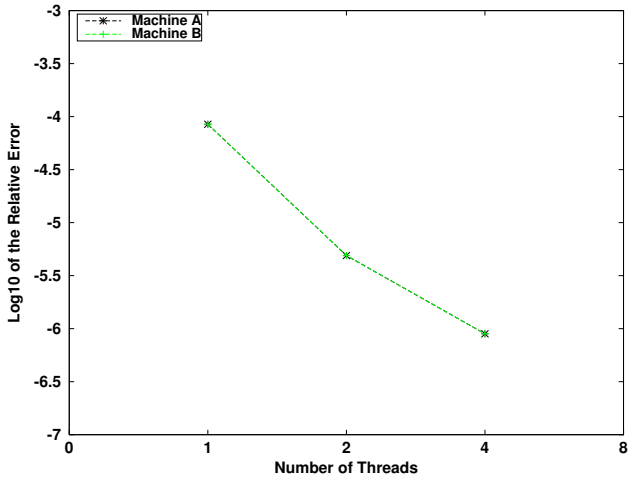


(b) Trsv. Cond = 10^8, Size = 12000

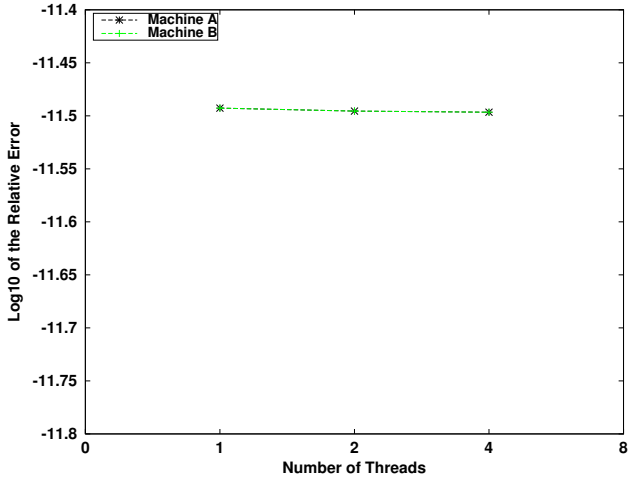


(c) Gemm. Cond = 10^8, Size = 1500

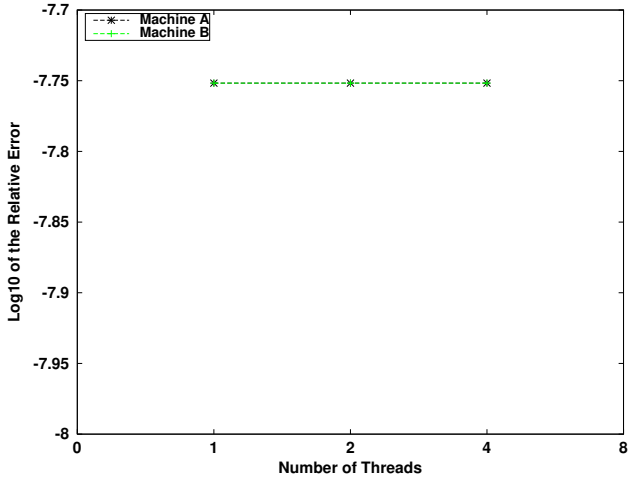
Figure A.1: Numerical reproducibility for two different environments with CNR turned off



(a) Dot product. Cond = 10^8, Size = 10^6

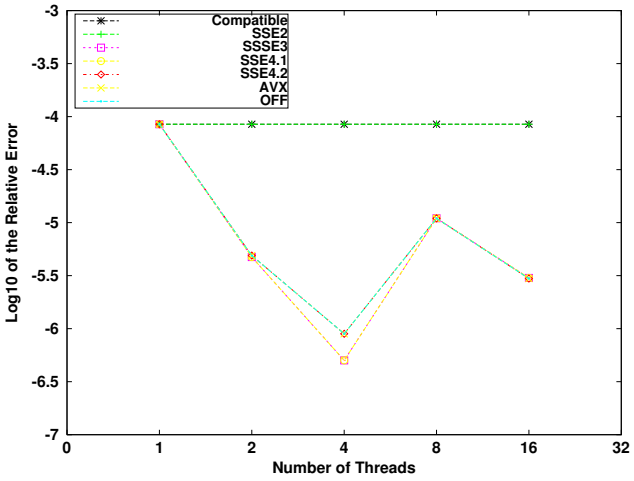


(b) Trsv. Cond = 10^8, Size = 12000

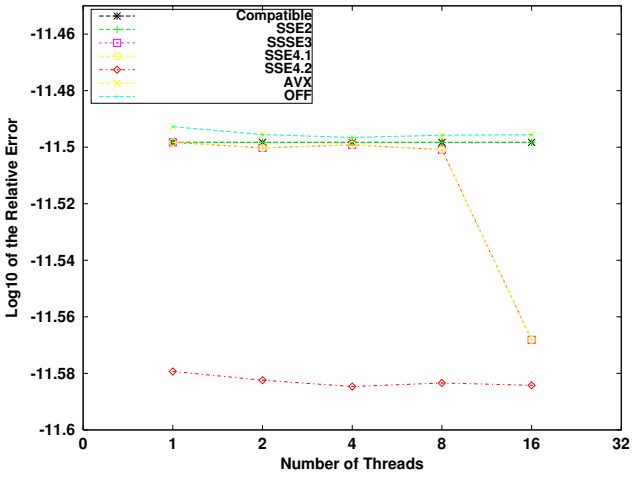


(c) Gemm. Cond = 10^8, Size = 1500

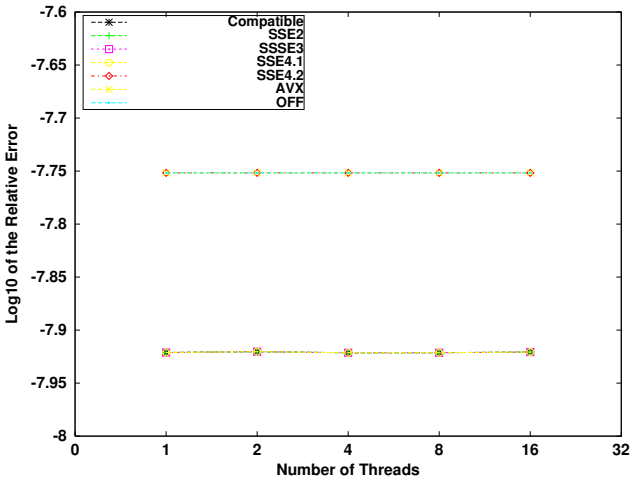
Figure A.2: Numerical reproducibility for two different environments with MKL_CBWR set to *AVX*



(a) Dot, Cond = 10^8, Size = 10^6



(b) Trsv, Cond = 10^8, Size = 12000



(c) Gemm, Cond = 10^8, Size = 1500

Figure A.3: Numerical reproducibility for different thread configurations (environment A)

SS4.2 is used. However the results are not reproducible for SS4.1, SSSE3 or SSE2 (result variations are very small and hard to see in Figure A.3c. So see Figure A.4 for smaller range of Y-axis). The same behavior is observed for trsv in Figure A.3b.

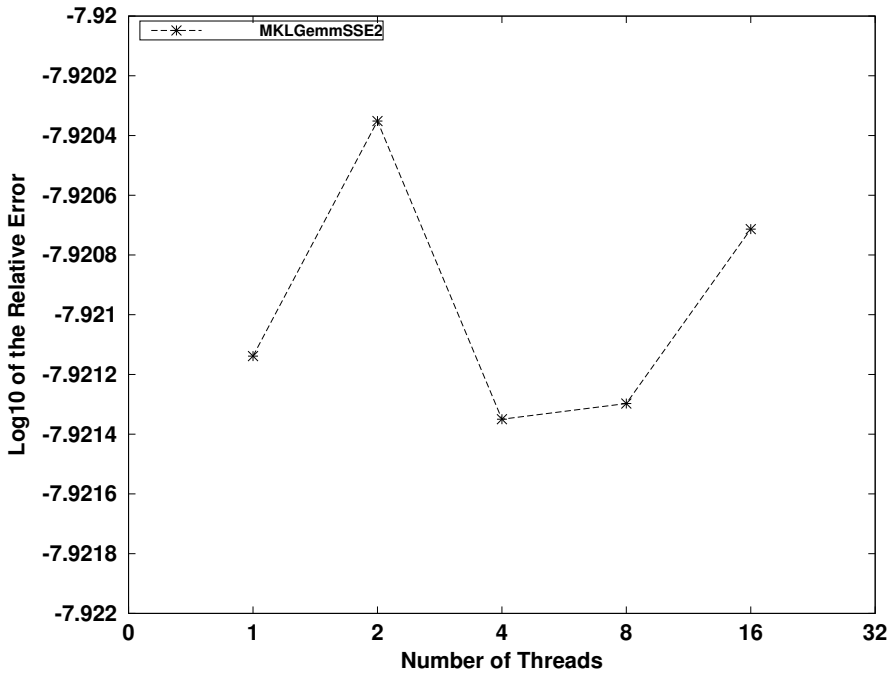
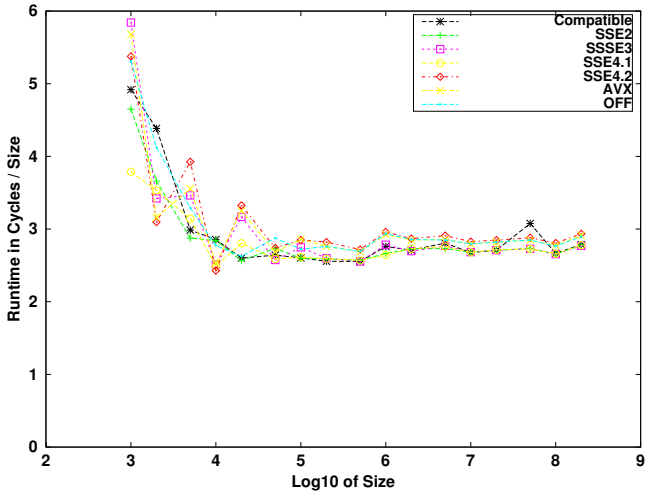


Figure A.4: Non reproducibility of gemm based on SSE2 for different number of threads, Size = 1500 (Focus of Figure A.3c)

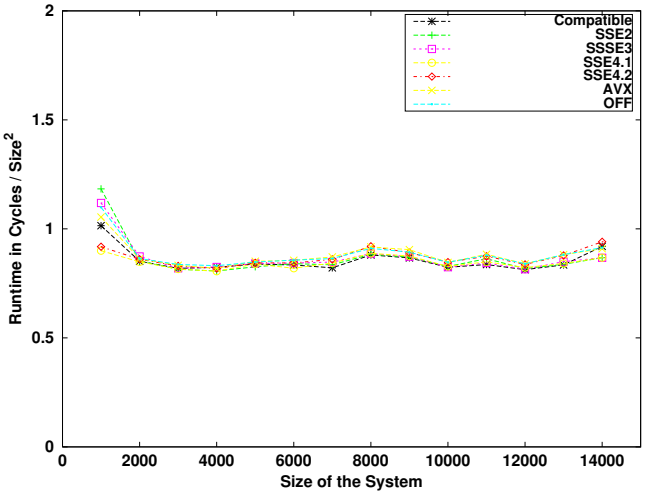
A.2.3 *Performance Results*

Impact of CNR on performance varies depending on the addressed function. Timing results are shown in Figures A.5 and A.7.

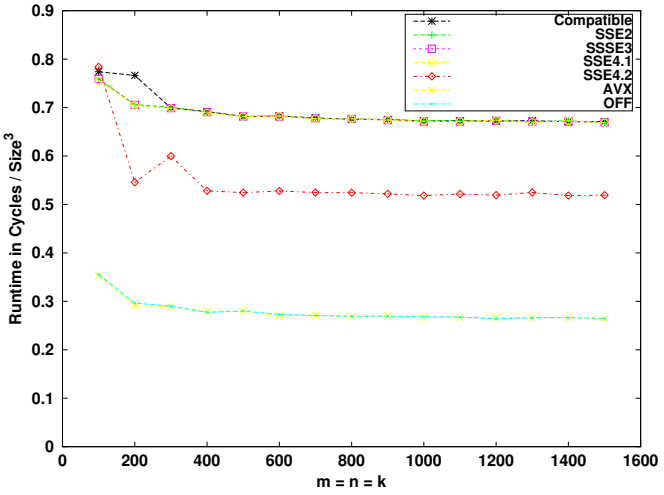
For memory bound level 1 and 2 BLAS functions we see with Figures A.5a and A.5b that there is no significant performance difference in the sequential case. At this BLAS level, the performance bottleneck lies in the available memory bandwidth. Since the cost of loading a floating-point number from the memory is much higher than the cost of a floating-point operation this memory bottleneck can not be removed even when the most efficient available instruction set is used. In the parallel case for dot and trsv (Figures A.6a and A.6b) all configurations perform similarly except Compatible and SSE2 that scale down at the sequential version level. We have investigated the execution of



(a) Sequential dot

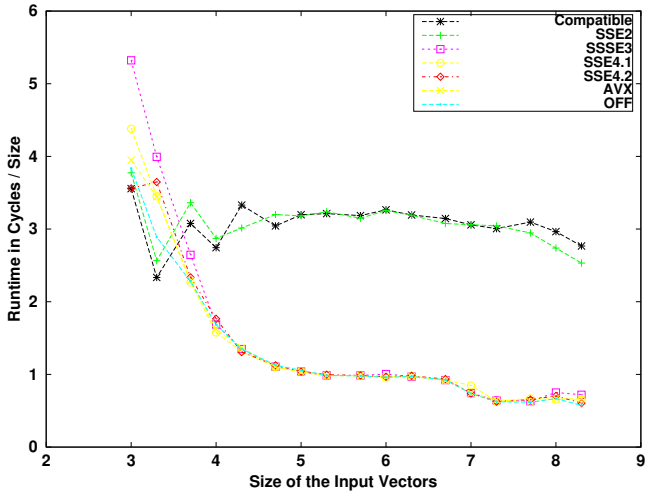


(b) Sequential trsv

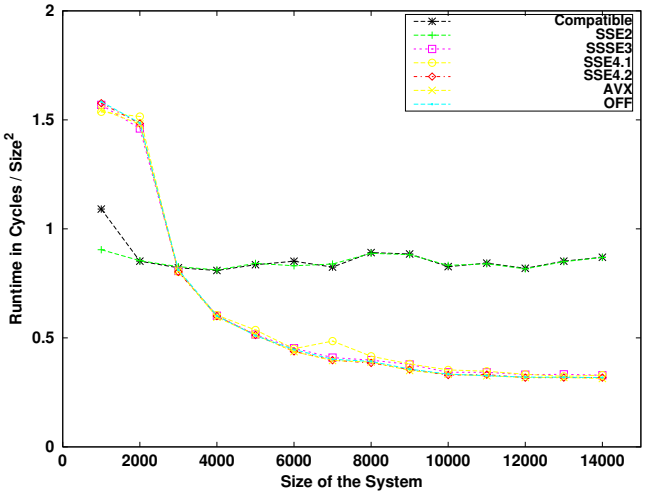


(c) Sequential gemm

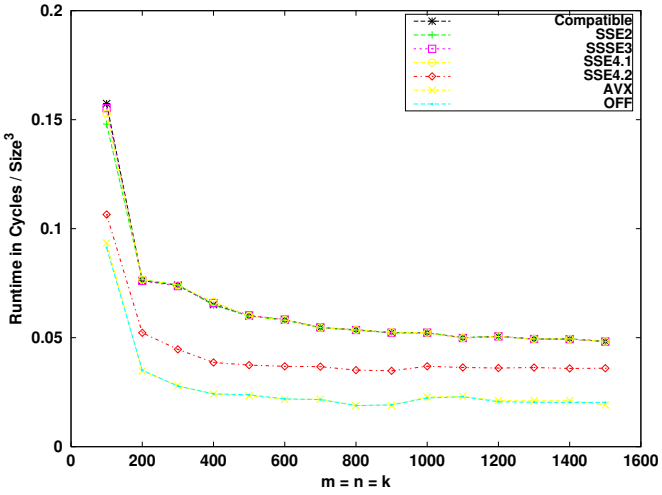
Figure A.5: Performance of sequential MKL on environment A



(a) Parallel dot

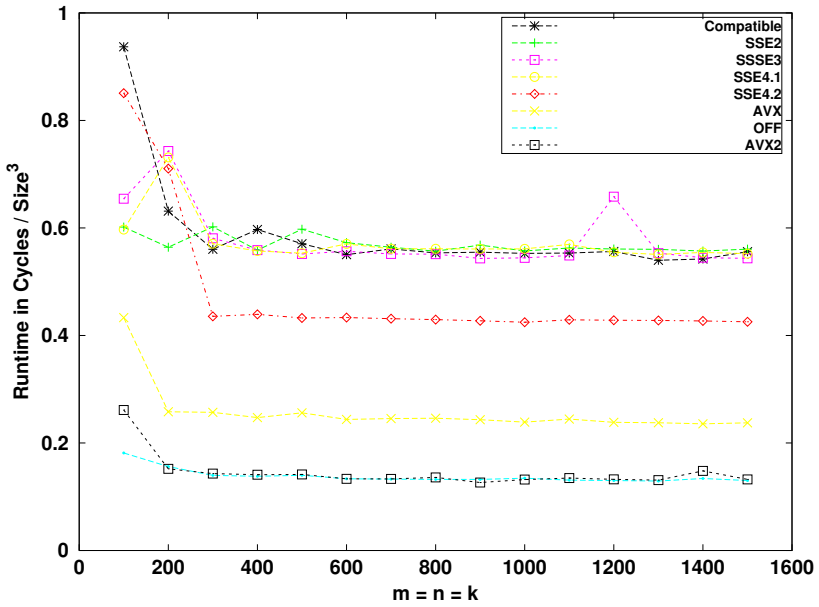


(b) Parallel trsv

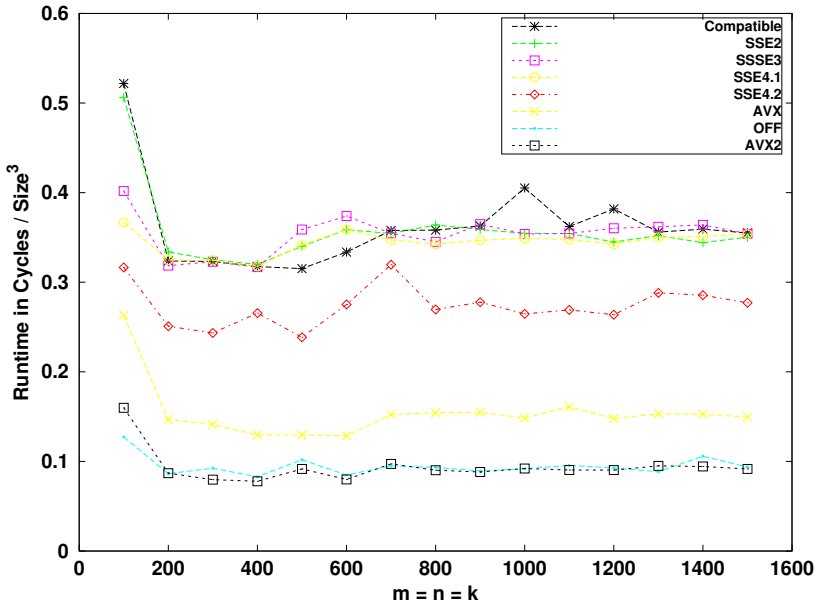


(c) Parallel gemm, Cond = 10^8

Figure A.6: Performance of parallel MKL on environment A



(a) Sequential gemm



(b) Parallel gemm

Figure A.7: Performance results for gemm (environment D)

those two versions by setting to verbose the environment variable "KMP_AFFINITY" [22] and see detailed informations about the thread mapping. Both Compatible and SSE2 versions run a single threaded function for dot and trsv even when we link with parallel MKL library and disable the dynamic control of thread configuration [16].

On the other side, for compute bound functions as *gemm*, impact of the CNR feature on performance is very important. For both the sequential and the parallel case in Figures A.5c and A.6c respectively, extra cost is up to twice when we set "MKL_CBWR" to SSE4_2, and it goes up to about 3 times for older instruction sets. Note that when CNR is turned off, the most recent supported instruction set is used. Therefore the extra cost of using older instruction sets should be higher if the processor supports newer instruction sets.

To explain this behavior we exhibit in Figure A.7 the performance of matrix multiplication for different values of "MKL_CBWR" for environment D that supports AVX2 instruction set. The extra cost of using the oldest instruction set compared to AVX2 goes up to 5 times in both sequential and parallel cases.

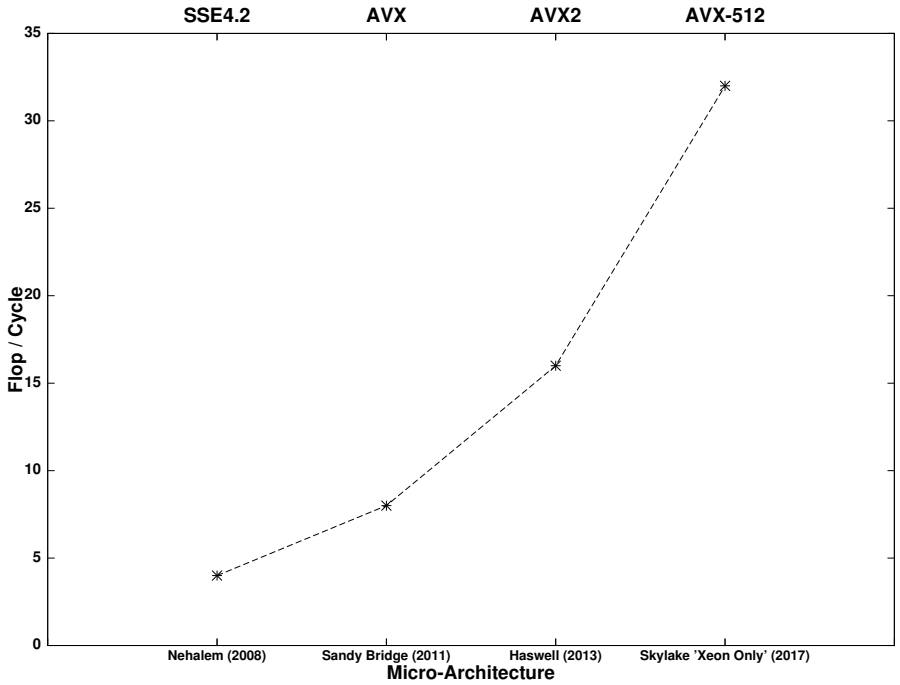


Figure A.8: Floating Point Capabilities of a Single Core on Intel Processors

This behavior raises important questions about the future proofness of this solution. According to current trend on Intel processors, the floating point performance of a single core is doubling every 3 years as shown in Figure A.8. Hence, the performance

that we drop for keeping numerical reproducibility of a program will grow exponentially.

A.3 CONCLUSION

In this appendix we have discussed the CNR feature (Conditional Numerical Reproducibility) available since release 11.0 of Intel MKL library. This feature aims at getting reproducible numerical results from run to run. Numerical reproducibility is ensured only when the same number of threads is used. To ensure reproducibility on different processors CNR unifies the used instruction set using the common best advanced one. Therefore, lot of performance have to be discarded to get reproducibility between old and new processors. Since the floating-point performance of Intel processors doubles every 3 years, extra cost of using CNR feature for CPU bound functions would grow exponentially in the future.

B

WORK METHODOLOGY

B.1 INTRODUCTION

This appendix details the experimental methodology and frameworks of this work Software and hardware combinations that define our test environments is first introduced in Section B.2. Section B.3 exhibits how we perform running time evaluations. And finally we present in Section B.4 how are generated the data data for the different problems that we deal with.

B.2 TEST ENVIRONMENTS

We have considered in this work four frameworks described in Table B.1. Tests with environment A validates the performance of the algorithms on CPU in the sequential and parallel cases. With environment B we validate the portability and the efficiency of presented algorithms many-core Intel Xeon Phi accelerator. We use OpenMP library [63] to implement a parallel version of our algorithms for both CPU and Xeon Phi accelerator. Environment C is the very new occigen supercomputer[1]. It allows to test the distributed memory parallel version of our algorithms. On this latter, the OpenMP library is used for multithreading on the same socket, and OpenMPI is used to exchange data between different sockets. Only dot product is tested on environment C in Chapter 5. Finally, we use environment D to analyze the CNR feature of the Intel MKL library in Appendix A.

Note that Xeon Phi accelerator does not support the same instruction set as CPUs. Therefore, different intrinsic instructions are used for vectorization [32].

1 https://www.cines.fr/calcul/materiels/occigen/

Environment A (workstation)	
Processor	Dual Intel Xeon E5-2650 v2 16 cores (8 per socket). Hyper-threading disabled. L1/L2 = 32/256 KB per core. L3 = 20 MB per socket.
Bandwidth	59,7 GB/s.
Compiler	Intel ICC 16.0.0.
Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops.
Libraries	Intel OpenMP 5, Intel MKL 11.3.
ISA	AVX.
Environment B (Intel Xeon Phi accelerator)	
Processor	Intel Xeon Phi 7120 accelerator, 60 cores, 4 threads per core. L1/L2 = 32/512 KB per core.
Bandwidth	352 GB/s.
Compiler	Intel ICC 16.0.0.
Options	-O3 -mmic -fp-model double -fp-model strict -funroll-all-loops.
Libraries	Intel OpenMP 5. Intel MKL 11.3.
Environment C (Occigen Supercomputer)	
Processor	4212 Intel Xeon E5-2690 v3 (12 cores per socket). L1/L2 = 32/256 KB per core. L3 = 30 MB per socket.
Bandwidth	68 GB/s.
Compiler	Intel ICC 15.0.0.
Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops.
Libraries	Intel OpenMP 5. Intel MKL 11.2. OpenMPI 1.8.
ISA	AVX2.
Environment D (Laptop)	
Processor	Intel Core I7-4500U, 2 cores, 4 threads. L1/L2 = 32/256 KB per core. L3 = shared 4 MB.
Bandwidth	25,6 GB/s.
Compiler	Intel ICC 16.0.0.
Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops.
Libraries	Intel OpenMP 5. Intel MKL 11.3.
ISA	AVX2.

Table B.1: Experimental frameworks

B.3 PERFORMANCE EVALUATION

Measuring run time performance of a program is very important and challenging on todays machines. Performance analysis is never perfectly accurate. However, this measure should at least be accurate enough to be trusted. Recent processors provide hardware counters to provide more reliable performance measures. We use the `rdtsc` assembly instruction to access to the register that counts the number of cycles. This latter is called before and after the tested function, and the difference between two calls estimate the running time of the function. To minimize measurement overhead, no software API is used and assembly instruction is coded directly in C using `asm` instruction. We also run the experiment several times and take only the minimal measured value (16 times for summation and level 1 BLAS and 8 times for level 2 BLAS). Insignificant running time measures are excluded.

B.4 DATA GENERATION

We present in this section our input data generators for summation, dot product, matrix vector multiplication and triangular solver.

B.4.1 *Summation and Dot Product Data Generator*

We use the generator from [44] to generate dot products with the requested size and condition number. A modified version of this algorithm is used to generate condition dependent data for summation. The principle is to generate a floating-point vector p such that the value $\sum_{i=1}^{n} |p_i| / |\sum_{i=1}^{n} p_i|$ equals (approximately) the requested condition number. For this, random floating-point numbers with wide exponent range are generated. Afterwards, cancellations are introduced to eliminate the larger values and reduce the value of $|\sum_{i=1}^{n} p_i|$ compared to $\sum_{i=1}^{n} |p_i|$.

B.4.2 *Matrix Vector Multiplication Data Generator*

We use the same algorithm to generate a matrix A and a vector x such that all the dot products $a^{(i)} \cdot x$ ($a^{(i)}$ is the i^{th} row of i) depend on the requested condition number. Note that we do not consider the condition number of the matrix A, but we focus on the condition number of the dot products $a^{(i)} \cdot x$ that are more significant in practice.

B.4.3 *Triangular Systems Data Generator*

Contrary to previous generators, we generate condition dependant data for triangular systems in an empirical way that does not rely on solid mathematical properties. Different approaches are used to generate matrices with small, medium or large condition numbers.

B.4.3.1 *Large Condition Numbers*

Viswanath and Trefethen demonstrated that the condition number of randomly generated triangular matrices grows exponentially with the size n [67] . For $n = 1000$, this property is used to generate triangular systems with a condition number larger than 10^{12}.

B.4.3.2 *Small Condition Numbers*

To generate large triangular matrices with small condition numbers, we first generate a random full dense matrix A. Edelman has shown that the condition number of a randomly generated full dense matrix grows linearly with its size [17]. Afterwards, the QR decomposition is performed to split A into two matrices Q and R, where Q is an orthogonal matrix and R is an upper triangular matrix. Finally we rotate the matrix R as shown in Figure B.1 to get a lower triangular matrix L. We observe here that $cond(L) \approx cond(A)$ which grows linearly with the matrix size n. For $n = 1000$ This process provides matrices with condition number smaller than 10^6.

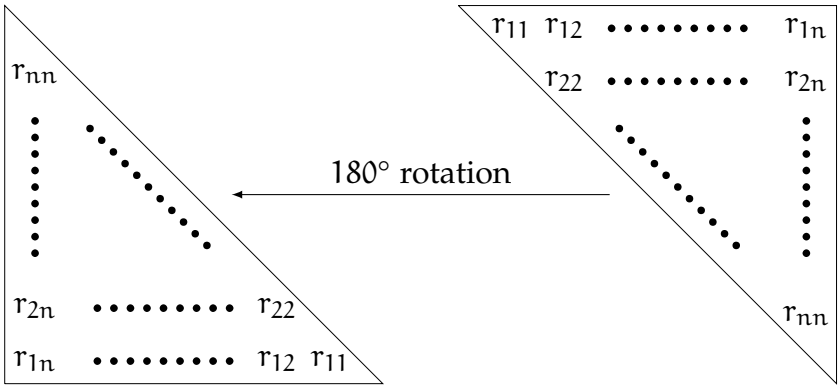


Figure B.1: Matrix 180° rotation

B.4.3.3 *Medium Condition Numbers*

Finally to generate matrices with condition numbers between 10^6 and 10^{12}, we generate first a lower triangular matrix with a small condition number, then we introduce some random perturbations that increase its condition.

BIBLIOGRAPHY

[1] *A Reference Implementation for Extended and Mixed Precision BLAS*. URL: http://crd.lbl.gov/~xiaoye/XBLAS.

[2] *ACML – AMD Core Math Library*. http://developer.amd.com/tools-and-sdks/archive/compute/amd-core-math-library-acml/. [Online; accessed 01-January-2017].

[3] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/. [Online; accessed 01-January-2017].

[4] Javier D Bruguera and Tomás Lang. "Floating-point fused multiply-add: reduced latency for floating-point addition." In: *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. IEEE. 2005, pp. 42–51.

[5] D. Chapp, T. Johnston, and M. Taufer. "On the Need for Reproducible Numerical Accuracy through Intelligent Runtime Selection of Reduction Algorithms at the Extreme Scale." In: *2015 IEEE International Conference on Cluster Computing*. 2015, pp. 166–175. DOI: 10.1109/CLUSTER.2015.34.

[6] Chemseddine Chohra, Philippe Langlois, and David Parello. "Level 1 Parallel RTN-BLAS: Implementation and Efficiency Analysis." In: *SCAN: Scientific Computing, Computer Arithmetic and Validated Numerics*. Available at http://www.scan2014.uni-wuerzburg.de/fileadmin/10030000/scan2014/talks/E1_2.pdf. Wurzburg, Germany, Sept. 2014. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-01095172.

[7] Chemseddine Chohra, Philippe Langlois, and David Parello. "Parallel experiments with RARE-BLAS." In: *SYNASC: Symbolic and Numeric Algorithms for Scientific Computing*. Available at https://hal-lirmm.ccsd.cnrs.fr/lirmm-01349698/document. Timisoara, Romania, Sept. 2016. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-01349698.

[8] Chemseddine Chohra, Philippe Langlois, and David Parello. "Reproducible, Accurately Rounded and Efficient BLAS." In: *REPPAR: Reproducibility in Parallel Computing*. Grenoble, France, Aug. 2016.

[9] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. "Numerical Reproducibility for the Parallel Reduction on Multi- and Many-core Architectures." In: *Parallel Comput.* 49.C (Nov. 2015), pp. 83–97. ISSN: 0167-8191. DOI: 10.1016/j.parco.2015.09.001.

[10] Theodorus J. Dekker. "A Floating-Point Technique for Extending the Available Precision." In: *Numer. Math.* 18 (1971), pp. 224–242.

[11] James W. Demmel and Hong Diep Nguyen. "Fast Reproducible Floating-Point Summation." In: *Proc. 21th IEEE Symposium on Computer Arithmetic*. Austin, Texas, USA. 2013.

[12] James W. Demmel and Hong Diep Nguyen. "Toward Hardware Support for Reproducible Floating-Point Computation." In: *SCAN'2014*. Würzburg, Germany, 2014.

[13] James W. Demmel and Hong Diep Nguyen. "Parallel Reproducible Summation." In: 64.7 (2015), pp. 2060–2070. DOI: 10.1109/TC.2014.2345391.

[14] James Demmel, Peter Ahrens, and Hong Diep Nguyen. *Efficient Reproducible Floating Point Summation and BLAS*. Tech. rep. UCB/EECS-2016-121. EECS Department, University of California, Berkeley, 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html.

[15] James Demmel, Yozo Hida, William Kahan, Xiaoye S Li, Sonil Mukherjee, and E Jason Riedy. "Error bounds from extra-precise iterative refinement." In: *ACM Transactions on Mathematical Software (TOMS)* 32.2 (2006), pp. 325–351.

[16] Todd E. *MKL DYNAMIC*. https://software.intel.com/en-us/node/528704. [Online; accessed 14-October-2016]. 2012.

[17] Alan Edelman. "Eigenvalues and Condition Numbers of Random Matrices." In: *SIAM Journal on Matrix Analysis and Applications* 9.4 (1988), pp. 543–560. DOI: 10.1137/0609045. URL: http://dx.doi.org/10.1137/0609045.

[18] *Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL*. http://www-03.ibm.com/systems/power/software/essl/. [Online; accessed 01-January-2017].

[19] *ExBLAS – Exact BLAS*. https://exblas.lip6.fr/. [Online; accessed 01-October-2015].

[20] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Second. Baltimore, MD, USA: Johns Hopkins University Press, 1989.

[21] Stef Graillat, Christoph Lauter, Ping Tak Peter Tang, Naoya Yamanaka, and Shin'ichi Oishi. "Efficient Calculations of Faithfully Rounded L2-Norms of n-Vectors." In: *ACM Trans. Math. Softw.* 41.4 (Oct. 2015), 24:1–24:20. ISSN: 0098-3500. DOI: 10.1145/2699469.

[22] J. Hao. *Using KMP_AFFINITY to Create OpenMP* Thread Mapping to OS proc IDs*. https://software.intel.com/en-us/articles/using-kmp-affinity-to-create-openmp-thread-mapping-to-os-proc-ids. 2012.

[23] Yun He and Chris H. Q. Ding. "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications." In: *Proceedings of the 14th International Conference on Supercomputing*. ICS '00. Santa Fe, New Mexico, USA: ACM, 2000, pp. 225–234. ISBN: 1-58113-270-0. DOI: 10.1145/335231.335253. URL: http://doi.acm.org/10.1145/335231.335253.

[24] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd. SIAM, 2002, pp. xxx+680. ISBN: 0-89871-521-0.

[25] J. D. Hogg. "A Fast Dense Triangular Solve in CUDA." In: *SIAM Journal on Scientific Computing* 35.3 (2013), pp. C303–C322. DOI: 10.1137/12088358X. URL: http://dx.doi.org/10.1137/12088358X.

[26] *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York: Institute of Electrical and Electronics Engineers, Aug. 2008, p. 58. ISBN: 0-7381-5753-8 (paper), 0-7381-5752-X (electronic). DOI: http://doi.ieeecomputersociety.org/10.1109/IEEESTD.2008.4610935.

[27] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York: Institute of Electrical and Electronics Engineers, Aug. 2008, p. 58. ISBN: 0-7381-5753-8 (paper), 0-7381-5752-X (electronic). DOI: http://doi.ieeecomputersociety.org/10.1109/IEEESTD.2008.4610935.

[28] R. Iakymchuk, D. Defour, S. Collange, and S. Graillat. "Reproducible Triangular Solvers for High-Performance Computing." In: *2015 12th International Conference on Information Technology - New Generations*. 2015, pp. 353–358. DOI: 10.1109/ITNG.2015.63.

[29] Intel. *New Microarchitecture for 4th Gen Intel Core Processor Platforms*. Tech. rep. available at URL = http://www.intel.fr/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf. 2013.

[30] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Tech. rep. available at URL = http://www.intel.fr/content/www/fr/fr/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. 2016.

[31] *Intel® Math Kernel Library (Intel® MKL) | Intel® Software*. https://software.intel.com/en-us/intel-mkl. [Online; accessed 01-January-2017].

[32] *Intrinsics*. https://software.intel.com/en-us/node/523351. [Online; accessed 18-October-2016].

[33] *Introduction to Conditional Numerical Reproducibility (CNR)*. https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr. [Online; accessed 11-October-2016].

[34] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Third. Reading, MA, USA: Addison-Wesley, 1998, pp. xiii+688. ISBN: 0-201-89684-2.

[35] Ulrich Kulisch and Van Snyder. "The Exact Dot Product As Basic Tool for Long Interval Arithmetic." In: *Computing* 91.3 (Mar. 2011), pp. 307–313. ISSN: 0010-485X. URL: http://dx.doi.org/10.1007/s00607-010-0127-7.

[36] P. Langlois, R. Nheili, and C. Denis. "Recovering Numerical Reproducibility in Hydrodynamic Simulations." In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. 2016, pp. 63–70. DOI: 10.1109/ARITH.2016.27.

[37] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jummy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. "Design, Implementation and Testing of Extended and Mixed Precision BLAS." In: *ACM Transactions on Mathematical Software* 28.2 (June 2002), pp. 152–205.

[38] Robert Lucas, J Ang, K Bergman, S Borkar, W Carlson, L Carrington, G Chiu, R Colwell, W Dally, J Dongarra, et al. "Top ten exascale research challenges." In: *DOE ASCAC Subcommittee Report* (2014).

[39] Vlăduţiu Mircea. "On the Design of Floating Point Units for Interval Arithmetic." available at URL = http://www.acsa.upt.ro/public/Amaricai_prop.pdf. 2007.

[40] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, p. 572. ISBN: 978-0-8176-4704-9.

[41] Radford M. Neal. "Fast exact summation using small and large superaccumulators." In: *CoRR* abs/1505.05571 (2015).

[42] Hong Diep Nguyen, James Demmel, and Peter Ahrens. *ReproBLAS: Reproducible BLAS*. http://bebop.cs.berkeley.edu/reproblas/.

[43] Stuart Franklin Oberman. "Design Issues in High Performance Floating Point Arithmetic Units." UMI Order No. GAX97-14163. PhD thesis. Stanford, CA, USA, 1997.

[44] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. "Accurate sum and dot product." In: *SIAM J. Sci. Comput.* 26.6 (2005), pp. 1955–1988.

[45] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic: Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001, pp. xiv+104. ISBN: 0-89871-482-6.

[46] Katsuhisa Ozaki, Takeshi Ogita, Shin'ichi Oishi, and Siegfried M. Rump. "Generalization of error-free transformation for matrix multiplication and its application." In: *Nonlinear Theory and Its Applications, IEICE* 4.1 (2013), pp. 2–11. DOI: 10.1587/nolta.4.2.

[47] Douglas M. Priest. "Algorithms for Arbitrary Precision Floating Point Arithmetic." In: *Proc. 10th IEEE Symposium on Computer Arithmetic*. Ed. by Peter Kornerup and David W. Matula. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp. 132–143.

[48] Douglas M. Priest. "On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations." URL = ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z. PhD thesis. Berkeley, CA, USA: Mathematics Department, University of California, Nov. 1992, p. 126.

[49] Rezaur Rahman. *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[50] Robert Reed. "An Introduction to the ® Xeon Phi™ Coprocessor." 2013. URL: https://software.intel.com/sites/default/files/Slides-Intel-Xeon-Phi-coprocessor-hardware-and-software-architecture-introduction.pdf.

[51] Robert W. Robey, Jonathan M. Robey, and Rob Aulwes. "In search of numerical consistency in parallel programming." In: *Parallel Computing* 37.4–5 (2011), pp. 217 –229. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2011.02.009. URL: http://www.sciencedirect.com/science/article/pii/S0167819111000238.

[52] Siegfried M. Rump. "Ultimately fast accurate summation." In: *SIAM J. Sci. Comput.* 31.5 (2009), pp. 3466–3502.

[53] Siegfried M. Rump. personal communication. Feb. 20, 2014.

[54] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. "Accurate floating-point summation – Part I: Faithful rounding." In: *SIAM J. Sci. Comput.* 31.1 (2008), pp. 189–224.

[55] Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi. "Accurate floating-point summation – Part II: Sign K-Fold Faithful and Rounding to Nearest." In: *SIAM J. Sci. Comput.* 31.2 (2008), pp. 1269–1302.

[56] Ahmed H. Sameh and Richard P. Brent. "Solving Triangular Systems on a Parallel Computer." In: *SIAM Journal on Numerical Analysis* 14.6 (1977), pp. 1101–1113. DOI: 10.1137/0714076. URL: http://dx.doi.org/10.1137/0714076.

[57] Jonathan Shewchuk. "Adaptive precision floating-point arithmetic and fast robust geometric predicates." In: *Discrete Comput. Geom.* 18.3 (1997). ACM Symposium on Computational Geometry (Philadelphia, PA, 1996), pp. 305–363. ISSN: 0179-5376.

[58] Robert D. Skeel. "Scaling for Numerical Stability in Gaussian Elimination." In: *J. ACM* 26.3 (July 1979), pp. 494–526. ISSN: 0004-5411. DOI: 10.1145/322139.322148. URL: http://doi.acm.org/10.1145/322139.322148.

[59] *Specifying Code Branches | Intel® Software - Intel® Developer Zone*. https://software.intel.com/en-us/node/528578.

[60] Pat H. Sterbenz. *Floating-Point Computation*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1974, pp. xiv+316. ISBN: 0-13-322495-3.

[61] M. Taufer, O. Padron, P. Saponaro, and S. Patel. "Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs." In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–9. DOI: 10.1109/IPDPS.2010.5470481.

[62] *The GNU MPFR library*. URL = http://www.mpfr.org/. [Online; accessed 08-July-2016]. 2004.

[63] *The OpenMP API Specification for Parallel Programming*. http://www.openmp.org/. [Online; accessed 01-January-2017].

[64] *Threading Building Blocks*. https://www.threadingbuildingblocks.org/. [Online; accessed 01-January-2017].

[65] R. Todd. *Run-to-Run Numerical Reproducibility with the Intel® Math Kernel Library and Intel® Composer XE 2013*. Tech. rep. Intel Corporation, 2013.

[66] Oreste Villa, Daniel Chavarria-Miranda, Vidhya Gurumoorthi, Andrés Márquez, and Sriram Krishnamoorthy. "Effects of floating-point non-associativity on numerical computations on massively multithreaded systems." In: *Proceedings of Cray User Group Meeting (CUG)*. 2009.

[67] D. Viswanath and L. N. Trefethen. "Condition Numbers of Random Triangular Matrices." In: *SIAM Journal on Matrix Analysis and Applications* 19.2 (1998), pp. 564–581. DOI: 10.1137/S0895479896312869. URL: http://dx.doi.org/10.1137/S0895479896312869.

[68] N. Yamanaka, T. Ogita, S.M. Rump, and S. Oishi. "A parallel algorithm for accurate dot product." In: *Parallel Comput.* 34.6–8 (2008), pp. 392 –410. ISSN: 0167-8191.

[69] Yong-Kang Zhu and Wayne B. Hayes. "Correct rounding and hybrid approach to exact floating-point summation." In: *SIAM J. Sci. Comput.* 31.4 (2009), pp. 2981–3001. ISSN: 1064-8275.

[70] Yong-Kang Zhu and Wayne B. Hayes. "Algorithm 908: Online Exact Summation of Floating-Point Streams." In: *ACM Trans. Math. Software* 37.3 (2010), 37:1–37:13.

[71] Yong-Kang Zhu, Jun-Hai Yong, and Guo-Qin Zheng. "A New Distillation Algorithm for Floating-Point Summation." In: *SIAM Journal on Scientific Computing* 26.6 (2005), pp. 2066–2078. DOI: 10.1137/030602009. URL: http://dx.doi.org/10.1137/030602009.

[72] *cuBLAS :: CUDA Toolkit Documentation - NVIDIA Documentation*. http://docs.nvidia.com/cuda/cublas/. [Online; accessed 01-January-2017].

ABSTRACT

Numerical reproducibility failures rise in parallel computation because floating-point summation is non-associative. Massively parallel systems dynamically modify the order of floating-point operations. Hence, numerical results might change from one run to another. We propose to ensure reproducibility by extending as far as possible the IEEE-754 correct rounding property to larger computing sequences. We introduce RARE-BLAS a reproducible and accurate BLAS library that benefits from recent accurate and efficient summation algorithms. Solutions for level 1 (asum, dot and nrm2) and level 2 (gemv and trsv) routines are designed. Implementations relying on parallel programming API (OpenMP, MPI) and SIMD extensions are proposed. Their efficiency is studied compared to optimized library (Intel MKL) and other existing reproducible algorithms.

RÉSUMÉ

Le problème de non-reproductibilté numérique surgit dans les calculs parallèles principalement à cause de la non-associativité de l'addition flottante. Les environnements parallèles changent dynamiquement l'ordre des opérations. Par conséquent, les résultats numériques peuvent changer d'une exécution à une autre. Nous garantissons la reproductibilité en étendant autant que possible l'arrondi correct à des séquences de calculs plus importantes que les opérations arithmétique exigées par le standard IEEE-754. Nous introduisons RARE-BLAS une implémentation des BLAS qui est reproductible et précise en utilisant les transformations sans erreur et les algorithmes de sommation appropriés. Nous présentons dans cette thèse des solutions pour le premier (asum, dot and nrm2) et le deuxième (gemv and trsv) niveaux des BLAS. Nous développons une implémentation de ces solutions qui utilise les interfaces de programmation parallèles (OpenMP et MPI) et les jeu d'instructions vectorielles. Nous comparons l'efficacité de RARE-BLAS à une bibliothèque optimisé (Intel MKL) et à des solutions reproductibles existantes.