Mixed precision algorithms for low-rank matrix and tensor approximations

Matthieu Robeyns

▶ To cite this version:

Matthieu Robeyns. Mixed precision algorithms for low-rank matrix and tensor approximations. Computer Science [cs]. Université Paris-Saclay, 2024. English. NNT : 2024UPASG095 . tel-04885863

HAL Id: tel-04885863
https://theses.hal.science/tel-04885863v1

Submitted on 14 Jan 2025

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

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

universite PARIS-SACLAY

Mixed precision algorithms for low-rank matrix and tensor approximations

Algorithmes en précision mixte pour des approximations de rang faible de matrices et tenseurs

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 : sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Référent : Faculté des sciences d'Orsay
Graduate School : Informatique et sciences du numérique.

Thèse préparée dans l'unité de recherche **Laboratoire interdisciplinaire des sciences du numérique (Université Paris-Saclay, CNRS)**, sous la direction de Pr. **Marc BABOULIN**, le co-encadrement de Dr. **Oguz KAYA**, et le co-encadrement de Dr. **Théo MARY**.

Thèse soutenue à Paris-Saclay, le 10 Décembre 2024, par

Matthieu ROBEYNS

Composition du jury

Membres du jury avec voix délibérative

Sylvie Boldo Directrice de recherche, Inria, Laboratoire Méthodes Formelles, Saclay	Présidente du Jury et Examinatrice
Rio Yokota Professeur, Tokyo Institute of Technology (Japon)	Rapporteur et Examinateur
Anthony Nouy Professeur, École Centrale de Nantes	Rapporteur et Examinateur
Mariya Ishteva Professeur, Katholieke Universiteit Leuven (Belgique)	Examinatrice
Emmanuel Agullo Chargé de recherche, Inria, LABRI, Bordeaux	Examinateur

THESE DE DOCTORAT

NNT : 2024UPASG095

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre : Algorithmes en précision mixte pour des approximations de rang faible de matrices et tenseurs

Mots clés : Calcul en précision mixte, Calcul haute performance, Calcul tensoriel, Réduction de données.

Résumé : La gestion des données est souvent réalisée par des objets mathématiques tels que les matrices et les tenseurs, qui sont la généralisation des matrices à plus de deux dimensions. Certains domaines d'application nécessitent de stocker trop d'éléments, créant des tenseurs trop grands ; ce problème est connu sous le nom de *curse of dimensionality*. Des méthodes mathématiques telles que les approximations de rang faible ont été développées pour réduire la dimensionnalité de ces objets malgré un coût très élevé en temps de calcul. De plus, de nouvelles architectures informatiques telles que les GPU nous permettent d'effectuer des calculs rapidement, notamment lors de calculs en faible précision. Combiner ces nouvelles architectures avec l'approximation de rang faible est une solution malgré la qualité des résultats altérée par la faible précision. Cette thèse vise à proposer des algorithmes d'approximation de rang faible stables en faible précision tout en conservant l'accélération inhérente au calcul en faible précision, ce qui est réalisable grâce au calcul en précision mixte.

Nous avons développé une méthode générale d'approximation de tenseurs en précision mixte en calculant d'abord une approximation en faible précision et en l'affinant itérativement avec une précision supérieure pour maintenir la qualité du résultat. Sachant que cette accélération provient principalement des architectures GPU, plus précisément d'unités de calcul spécialisées appelées *tensor cores*, nous avons développé une méthode générale d'approximation matricielle pour les architectures GPU en précision mixte utilisant ces *tensor cores*. Notre méthode maintient la qualité du résultat, mais au prix d'une approximation de dimension supérieur à celle des applications standards. Pour compenser cet écart, des méthodes de recompression de dimension existent pour différents formats de tenseurs. Notre contribution finale propose un cadre pour l'analyse de stabilité des opérations de réseaux tenseurs communs, qui nous guide également vers une méthode de recompression stable.

Title : Mixed precision algorithms for low-rank matrix and tensor approximations

Keywords : Data reduction, High performance computing, Tensor computations, Mixed precision computations.

Abstract : Data management is often done by mathematical objects such as matrices and tensors, which are the generalization of matrices to more than two dimensions. Some application domains require too many elements to be stored, creating tensors too large ; this problem is known as the *curse of dimensionality*. Mathematical methods such as low-rank approximations have been developed to reduce the dimensionality of these objects despite a very high cost in computation time. Moreover, new computer architectures such as GPUs allow us to perform computations quickly, especially when computing with low precision. Combining these new architectures with low-rank approximation is a solution despite the quality of the results being impaired by low precision. This thesis aims to propose low-rank approximation algorithms that are stable in low precision while maintaining the speedup inherent in low-precision computation, which is feasible thanks to mixed-precision computation.

We have developed a general method for mixed-precision tensor approximation by first computing a low-precision approximation and iteratively refining it with higher precision to maintain the quality of the result. Knowing that this speedup comes mainly from GPU architectures, more precisely from specialized computing units called *tensor cores*, we have developed a general matrix approximation method for mixed-precision GPU architectures using these *tensor cores*. Our method maintains the quality of the result but at the expense of a higher-dimensional approximation than standard applications. To compensate for this gap, dimension recompression methods exist for different tensor formats. Our final contribution proposes a framework for the stability analysis of common tensor network operations, which also guides us towards a stable recompression method.

Remerciements
Acknowledgements

Tout d'abord je voudrais remericier mes encadrants, à commencer par mon directeur de thèse Marc Baboulin qui m'a montré de bien des façons les responsabilités d'un doctorant et les charges administratives afférentes. Je remercie également mon co-encadrant Oguz Kaya qui a été mon tuteur principal durant ces trois années et qui m'a aidé à comprendre les enjeux de nos contributions visant simultanément le domaine informatique et mathématique. Il a permis de m'ouvrir à de nouveaux horizons en acceptant de me recruter pour cette thèse qu'il a financée et aussi me donner la possibilité de créer mon avenir professionnel en me laissant faire de l'enseignement hors contrat. However, je ne peux pas oublier de remercier mon dernier encadrant, Théo Mary qui malgré la distance a su me guider avec brio sur les aspects mathématiques de mes travaux. Il m'a initié à de nouvelles méthodes de calculs et d'enseignement. Mes encadrants m'on conforté dans l'idée que la recherche est aussi une expérience sociale en me présentant à nombreux de leurs collègues et amis, tout aussi passionés qu'eux. Mention spéciale à Simplice Donfack, un ingénieur de recherche du CEA qui nous a permis de travailler sur les performances de nos codes avec le cluster JeanZay. J'espère que l'avenir fera qu'on se recroisera pour de nouveaux projets, aussi agréable que celui que nous venons de finaliser ensemble.

Je remercie aussi les membres du jury qui ont accepté de lire et d'évaluer ce manuscrit. Donc un grand merci à eux, à savoir : Sylvie Boldo en tant que présidente du jury, Rio Yokota et Anthony Nouy en tant que rapporteurs, et enfin, Mariya Ishteva et Emmanuel Agullo en tant qu'examinateurs.

J'aimerais aussi remercier un bon nombre de collègues de bureaux que ce soit tant sur le plan administratif que matériel ou simplement de la recherche. Donc pour ne pas faire de jaloux et pour n'oublier personne, je les remercie par Equipe en commencant par l'Equipe ParSys bien évidemment. Ensuite dans les Equipes les plus proches on a A&O et GALaC avec qui on a pu créer le Séminaire Sé-PAG, qui est une démarche des plus passionnantes pour le développement inter-équipe. Je remercie aussi l'Equipe Ex-Situs pour leur bienvaillance et nos amitiés crées bien qu'ils étaient un étage au-dessus. Je ne peux pas oublier de remercier l'Equipe SAMI, SPIL, GRAFH et Hasard pour tout ce qui est lié à la partie administrative de la thèse. Enfin, je remercie l'Equipe PEQUAN du Lip6 pour m'avoir accueilli comme l'un des leurs à Jussieu et avoir pu présenter et écouter des présentations de qualité.

Oté ! Le moment lé venu de remercier la famille ! Merci à zot ! Tou la ban famille là, avec moman, papa, dada, nénène, kouzin et autres. Mi aime à zot ! Sans zot, mi noré pas pu arrivé là où mi lé.

Pour fermer la marche et m'épauler comme ils l'ont toujours fait, je remercie mes amis, mes proches, mes sangs, mes rheys comme dirait les jeuns. Beaucoup trop nombreux pour les citer, ils se reconnaitront en lisant ce message : Merci beaucoup à vous ! On a encore beaucoup de chemin à faire ensemble malgré les distances et que ce soit par nos hobbies, passions, ou mentalités, ce sera toujours un bon moment à partager x).

Table of contents

List of Figures **xiii**

List of Tables **xiii**

List of Algorithms **xv**

List of Acronyms **xvii**

Introduction **1**

1 Background **5**

1.1 Basic notations and definitions 5
1.2 Low-rank approximation arithmetic for matrices 8
1.2.1 Singular value decomposition 8
1.2.2 QR decomposition 9
1.2.3 Randomized methods 11
1.3 Low-rank approximation arithmetic for tensors 13
1.3.1 Tree tensor networks 13
1.3.2 Kernels for low-rank approximation on tensors 16
1.4 Floating point arithmetic 19
1.4.1 Properties 19
1.4.2 Commonly available floating point arithmetic 20
1.5 Mixed precision for low-rank approximations 21
1.5.1 Mixed precision for low-rank approximation for matrices 22
1.5.2 Mixed precision for low-rank approximation for tensors 22

2 Mixed precision iterative refinement for low-rank approximations **25**

2.1 Introduction 25
2.1.1 Notations 26
2.2 Iterative refinement for low-rank approximations 26
2.2.1 Error analysis 30
2.2.2 Complexity analysis 32
2.3 Application to matrix low-lank approximation 37
2.3.1 Truncated QRCP decomposition 37
2.3.2 Randomized SVD decomposition 39
2.4 Application to tensor low-rank approximation 40
2.4.1 Discussion of the error and complexity 41
2.5 Numerical experiments 42
2.5.1 Experimental setting 42
2.5.2 Experimental results 43

2.5.3 Role of θ_ℓ . . . 44
2.5.4 Results on real-life data . . . 48
2.5.5 Estimation of the time cost and role of ω_ℓ . . . 49
2.6 Conclusion . . . 52

3 Mixed precision randomized low-rank approximation with GPU tensor cores 53
3.1 Introduction . . . 53
3.2 Background . . . 54
3.2.1 Randomized LRA . . . 54
3.3 Mixed precision randomized LRA on GPU tensor cores . . . 55
3.3.1 GEMM kernel . . . 56
3.3.2 QR kernel . . . 57
3.3.3 Randomized LRA . . . 57
3.4 Experiments . . . 59
3.4.1 Experimental setting . . . 59
3.4.2 Performance and accuracy of kernels . . . 59
3.4.3 Mixed precision randomized LRA . . . 60
3.4.4 Iterative refinement . . . 62
3.5 Conclusion . . . 64

4 Numerical stability of tree tensor network operations, and a stable rounding algorithm 67
4.1 Introduction . . . 67
4.2 The framework . . . 68
4.2.1 Definitions and notations . . . 68
4.2.2 Operations on tree tensor networks . . . 69
4.2.3 α-normalized networks . . . 70
4.3 Stability analysis . . . 72
4.3.1 Model . . . 73
4.3.2 Local errors . . . 74
4.3.3 Global error . . . 76
4.4 Examples of stable tensor computations . . . 78
4.4.1 `full` . . . 78
4.4.2 `compress` . . . 80
4.4.3 `orthog` . . . 81
4.5 A general stable rounding algorithm . . . 82
4.5.1 The proposed algorithm . . . 82
4.5.2 Comparison with existing tensor rounding algorithms . . . 85
4.6 Numerical experiments . . . 86
4.6.1 Experimental setting . . . 86
4.6.2 Validating the stability of Algorithme 4.4 . . . 87
4.6.3 Comparison with Gram SVD . . . 88
4.7 Conclusion . . . 90

5 Conclusion **91**
5.1 Contributions . . . 91
5.2 Perspective . . . 92
5.3 Acknowledgements . . . 93

6 Publications **95**
6.1 Submitted articles . . . 95
6.2 Published articles . . . 95
6.3 Conference communications . . . 95

7 Appendix **97**
7.1 Résumé en français . . . 97

Bibliography **103**

List of Figures

1.1 Standard representation of data structure. 5
1.2 Concatenation of two tensors $\mathcal{X} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$, and $\mathcal{Y} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ along the diagonal of $\mathcal{Z} \in \mathbb{R}^{(m_1+n_1)\times(m_2+n_2)\times(m_3+n_3)}$ (top). Concatenation of two tensors $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, and $\mathcal{Y} \in \mathbb{R}^{n_1 \times n_4 \times n_3}$ along the common dimension n_1 and n_3 to obtain $\mathcal{Z} \in \mathbb{R}^{n_1 \times (n_2+n_4) \times n_3}$ (bottom). 7
1.3 Application of `reshape` kernels on $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ (as matricize kernel) to obtain a matrix $X \in \mathbb{R}^{n_1 \times n_2 n_3}$ (left); then apply a `reshape` kernel on X (as tensorize kernel) to bring back $\mathcal{X}$. . . 7
1.4 Low-rank approximation of a matrix $X \in \mathbb{R}^{m \times n}$ into factors $U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{n \times r}$. 8
1.5 SVD *decomposition* of a matrix $X \in \mathbb{R}^{m \times n}$ into factors $U \in \mathbb{R}^{m \times r}, \Sigma \in \mathbb{R}^{r \times r}$, and $V \in \mathbb{R}^{n \times r}$ where U and V are orthogonal (line inside). 9
1.6 Representation of QR *orthogonalization* of $X \in \mathbb{R}^{m \times n}$ into factors $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ (top) and QR *approximation* of $X \in \mathbb{R}^{m \times n}$ into factors $Q \in \mathbb{R}^{m \times r}, R \in \mathbb{R}^{r \times n}$ (bottom), where Q is orthogonal (line inside). 9
1.7 Randomized part of randomized low-rank approximation of $A \in \mathbb{R}^{m \times n}$. 12
1.8 Tensor network diagram of data object. 13
1.9 Tree tensor network representation 14
1.10 A d-order tensor (left) and its representation in Tucker, tensor-train (TT), and hierarchical Tucker (HT) decomposition (right) using tensor network diagram. 15
1.11 Floating point representation with a example of single precision. 19
1.12 Floating point precision comparison. 20
1.13 *Low precision* floating point representation. 21

2.1 Illustration of the LRA algorithm. 29
2.2 Three types of singular value distributions used in the experiments. 43
2.3 Convergence of Algorithme 2.1 for three types of matrices (with different singular value distributions, see Figure 2.2) and for two different `lra` kernels (QRCP or randomized SVD). The number next to each marker indicates the rank of F_i after recompression. 45

2.4 Convergence of Algorithme 2.1 for three types of tensors (depending on the singular value distribution, see Figure 2.2) and with Tensor-Train Singular Value Decomposition (TTSVD) or high-order singular value decomposition (HOSVD) as `lra` kernel. The numbers next to each marker indicate the rank vector of F_i after recompression. 46
2.5 Convergence of Algorithme 2.1 for three types of tensors (depending on the singular value distribution, see Figure 2.2) and with hierarchical Tucker singular value decomposition (HTSVD) as `lra` kernel. The numbers next to each marker indicate the rank vector of F_i after recompression. 47
2.6 Convergence of Algorithme 2.1 for a real-life matrix (Poisson) of size 253×252, for different `lra` kernels. The numbers next to each marker indicate the rank of F_i after recompression. . . . 50
2.7 Convergence of Algorithme 2.1 for a real-life tensor (H2CO) of size $17 \times 17 \times 13 \times 13 \times 9 \times 9$, for different `lra` kernels. The numbers next to each marker indicate the rank of F_i after recompression. 51

3.1 Performance of the GEMM and QR kernels. 61
3.2 Performance and accuracy of `randLRA` *without* refinement. (In (b), $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ completely overlap both for Householder QR and Cholesky QR). 63
3.3 Performance and accuracy of `randLRA` *with* refinement. (In (b), $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ completely overlap both for Householder QR and Cholesky QR). 65

4.1 Illustration of the `matricize` and `tensorize` kernels. 70
4.2 Illustration of the `split` and `merge` kernels. 70
4.3 Illustration of `full` and `compress`. 79
4.4 Accuracy of Algorithme 4.4 depending on the truncation threshold τ and the floating-point precision u. The text labels next to the markers indicate the compression ratio between a given variant in lower precision and the reference compression obtained in double precision, for the same value of τ (if this ratio is equal to 1 we omit the label). 87
4.5 Comparison between Algorithme 4.4 and Gram SVD depending on the truncation threshold τ and the floating-point precision u. The text labels next to the markers indicate the compression ratio between a given variant and the reference compression obtained with Algorithme 4.4 in double precision, for the same value of τ (if this ratio is equal to 1 we omit the label). 89

List of Tables

2.1 Relative error η_i and $\texttt{rank}(F_i)$ at different steps i and for different values of θ_ℓ, for TTSVD (using fp16 as u_ℓ and exponential distribution of singular values). 48
2.2 Relative error η_i and $\texttt{rank}(F_i)$ at different steps i and for different values of θ, for QRCP decomposition (using fp16 as u_ℓ and exponential distribution of singular values). 49

List of Algorithms

1.1 QRCP decomposition. 10
1.2 Randomized low-rank approximation fixed-rank variant. 11
1.3 Randomized SVD fixed-accuracy decomposition. 12
1.4 `lra` algorithm for a tensor. 17
1.5 `full` algorithm 18
1.6 `orthog` algorithm for tensor 18
2.1 Iterative refinement for Low-rank approximations (LRA). 28
2.2 `recompress` algorithm using truncated QRCP decomposition. . 38
2.3 Randomized SVD decomposition. 39
2.4 `decompress` algorithm for a tensor decomposition. 41
3.1 Randomized low-rank approximation fixed-rank variant. 54
3.2 Randomized LRA with iterative refinement. 56
3.3 Cholesky QR kernel implementation on GPU. 58
3.4 Mixed precision `randLRA` on GPU tensor cores. 58
3.5 Mixed precision `randLRA` on GPU tensor cores, with iterative refinement. 58
4.1 `full`() 78
4.2 `compress`() 80
4.3 `orthog`() 81
4.4 `round` 82
4.5 `truncate`($\mathcal{U}$) 84

List of Acronyms

ARF	approximation de rang faible
CP	CANDECOMP/PARAFAC
CPU	Central Processing Unit
FMA	Fused Multiply-Add
FPGA	Field-Programmable Gate Array
GEMM	GEneral Matrix Multiplication
GMRES	Generalized Minimal Residual
GPU	Graphics Processing Unit
HOSVD	high-order singular value decomposition
HT	hierarchical Tucker
HTSVD	hierarchical Tucker singular value decomposition
IEEE	Institute of Electrical and Electronics Engineers
IR	Iterative Refinement
LRA	Low-rank approximations
QRCP	QR Column Pivoting
SVD	singular value decomposition
TFLOPS	number of Tera floating-point operations per second
TPU	Tensor Processing Unit
TT	tensor-train
TTN	Tree Tensor Network
TTSVD	Tensor-Train Singular Value Decomposition

Introduction

Linear algebra is a fundamental tool in many scientific domains such as physics, chemistry, biology, and engineering [1, Part IV]. In computer science, its importance is even more pronounced with matrices, as it is the foundation of many algorithms and data structures. In recent years, the amount of data to be processed has grown exponentially, leading to new data structures such as tensors, which generalize matrices to orders higher than two. Recent research domains such as signal processing, image processing, quantum chemistry, and machine learning use tensors extensively [2, 3, 4, 5, 6, 7]. This increasing amount of data has led to a significant challenge in data computing : the *curse of dimensionality* that can be tackled via a method called Low-rank approximations (LRA). Our work will focus on the development of new algorithms for computing LRA in low precision arithmetic, which is a new and promising direction for accelerating linear algebra computations on modern hardware.

LRA is a powerful tool used in many scientific applications to reduce the dimension of large-scale data [7, 8, 9, 10]. For example, an $n \times n$ matrix X may be approximated by a low-rank product UV^T of $n \times r$ matrices U and V, reducing the initial storage cost from $O(n^2)$ to $O(nr)$. For tensors this storage cost is even more critical [7, 6, 8]—$O(n^d)$ for a dth order tensor. Tensor LRA methods decompose the full tensor as a product of tensors of lower order and lower rank; several low-rank tensor formats have been proposed like the Tucker [11, 12], the tensor-train (TT) [9] and the hierarchical Tucker (HT) [13, 14] formats.

However, computing matrix or tensor low-rank decomposition is a computationally intensive task; it represents the bottleneck of many LRA-based applications. Therefore, developing efficient algorithms for computing LRA is a crucial problem that has been the subject of many studies [15, 7, 8, 6].

A new possibility to accelerate the computation of LRA is to use *low precision arithmetic*, which provides significant performance benefits on modern hardware [16]. Especially half precision floating-point arithmetic such as the Institute of Electrical and Electronics Engineers (IEEE) fp16 and bfloat16 formats achieve very high speed on Graphics Processing Unit (GPU) accelerators with speed-up of up to $4\times$ compared to double precision (fp64) [17]. However, low precision degrades the accuracy of the computations; for example, half precision arithmetic provides, at best, between 3 and 4 digits of accuracy, depending on the format. Many applications require computing LRA with higher accuracy [15, 8].

This motivates the need for *mixed precision algorithms*, which combine multiple precision formats with the goal of achieving the high performance

of the low precision while preserving the high accuracy of the higher precision [18, 19]. Contrary to other linear algebra routines such as the solution of linear systems, there has been relatively little work on designing mixed (or even low) precision algorithms for LRA.

Thus, the main objective of this thesis is to develop mixed precision algorithms for LRA that are both accurate and efficient. This objective has been achieved by three main contributions, which are explained below.

LRA with iterative refinement As first contribution, we propose a new method for computing LRA in mixed precision arithmetic defined in Chapter 2. Our approach is applicable to basically any LRA algorithm, involving either matrices or tensors. It is reminiscent of the iterative refinement framework used for solving linear systems [20] : the idea is to first compute a LRA in low precision, then evaluate the error (or residual) from this first LRA, and re-apply the same LRA kernel to this error term to obtain a correction term that is used to refine the accuracy of the LRA. This can be repeated iteratively to reach any level of desired accuracy. The refined LRA is obtained as the sum of the original low precision LRA and the correction term, and is thus of larger yet still of low-rank. In order to contain the rank growth and maintain the optimal rank throughout the iterations, our method employs a "recompression" strategy [14, 21] that is performed in high precision but whose cost stays asymptotically smaller than that of LRA and become also a subject of our work as last contribution in Chapter 4. We carry out an error analysis of our method based on a general parameterized error model that only assumes that we have numerically stable implementations of the basic kernels used in our algorithm (LRA, matrix multiplication, and recompression). We show that the precision used for the LRA kernel—which is the computational bottleneck of the whole method—only affects the convergence speed of the process, but not its attainable accuracy. In order to assess under which conditions we can expect our method to be beneficial, we perform a complexity analysis that measures the cost of the method as a function of the numerical rank of the input as well as the speed ratio between the low and high precision arithmetic. We identify two situations where our method has a strong potential. The first is when the numerical rank of the input is small at low accuracy levels, which means that the singular values of the matrix or tensor are rapidly decaying ; in this case, the first iterations of our method becomes inexpensive. The second is when the hardware provides fast low precision matrix multiply–accumulate units [17], which allows computing the low precision LRA at a very high speed.

Mixed precision randomized LRA on GPU Random projection methods are simple and robust techniques for reducing the dimensionality of data while preserving data structure [22]. Moreover, the matrix operations at the heart of these methods make them highly suitable for exploiting accelerators such as GPUs [23]. Hence, we investigate to what extent these very fast low preci-

sion units can be exploited for accelerating randomized projection methods, especially in matrix case. The new contribution is the design of a new mixed precision randomized LRA method, with a performance and accuracy analysis showing that the proposed method is able to exploit GPU tensor cores reliably and efficiently. Our method is based on three key ideas : The first idea consists in *performing the matrix–matrix products (GEneral Matrix Multiplication (GEMM) kernel) in mixed precision arithmetic using the tensor cores*, since these operations represent the asymptotic bottleneck of the method. We compare several GEMM variants depending on how the conversions between fp32 and fp16 are handled, and identify one variant in particular that achieves the best performance–accuracy trade-off. Then, having significantly accelerated the GEMM operations, we observe that the orthonormalization step (QR kernel), despite requiring an asymptotically negligible number of flops, becomes the new performance bottleneck. Then the second idea is to *switch the orthonormalization method from the standard Householder QR to a CholeskyQR algorithm* [24], which mainly relies on GEMM and is therefore much more efficient on GPUs. We mitigate the inherent instability of CholeskyQR by performing it in fp64 rather than fp32 arithmetic. This leads to a mixed precision randomized LRA method employing three precisions (fp16, fp32, and fp64). We show that this method can be up to $8\times$ faster than the standard randomized LRA method [22] in fixed precision fp32 arithmetic and achieves an average accuracy of order 10^{-2}, which may be sufficient for some applications. Then the third idea is to *use our iterative refinement method for LRA proposed in the previous contribution above to improve the accuracy of the method*. We show that with refinement, the accuracy of this method improves significantly to an average of order 10^{-5}, while still being up to $2.2\times$ faster than the standard LRA method in fp32 arithmetic.

Stability of operations on low-rank tensor decomposition The LRA with iterative refinement proposed in the first contribution requires stable kernels, including a recompression kernel. This assumption motivates us as the last contribution to study the stability of tensor kernels computation, which has not been studied. This lack of analysis can be explained by mainly two reasons : first, the truncation errors tend to dominate the rounding errors when high precision is used, so that the latter have been traditionally neglected ; and second, there is a wide range of different tensor formats, and their associated algorithms can be very complex to analyze. Importantly, the first reason is becoming less and less valid due to the growing prevalence of low precision arithmetic, which offer significant performance benefits on modern hardware. And the second reason makes it even more critical to develop a rounding error analysis precisely to identify which tensor algorithms are stable and which will become problematic in low precision. Our contribution towards tackling these questions is two-fold. First, we propose a *general framework ba-*

sed on a tree tensor network format that encompasses the most important tensor formats (TT, Tucker, and HT) and we define the essential operations that are needed to express some of the most common computations of interest. We then carry out an error analysis of an abstract computation in this framework and identify conditions to guarantee its stability. In particular, a key condition is that the norm of the tensor should be tightly concentrated around a single node of the network, a property that we will formalize in Chapter 4. This property is notably satisfied when all nodes of the network except one are semi-orthogonal. Our analysis shows that if this property is maintained throughout the computation then the error introduced by each operation is controlled in terms of the norm of the global tensor. This first work can thus be used to establish the stability of a wide range of tensor computations that can be expressed in our framework. Our second work is to *use our framework to propose a general rounding algorithm.* In view of the conclusions of our error analysis, the algorithm is careful to maintain the semi-orthogonality of all nodes except one throughout the computation, and is thus guaranteed to be stable. The algorithm works for any tree topology of tensor and can thus be applied to a wide range of tensor formats. We compare this rounding algorithm with the existing Gram SVD-based rounding for hierarchical Tucker tensors proposed in [14], which is unstable [25]. We show that our algorithm can significantly improve the accuracy of the rounding in finite precision arithmetic, and can thus more reliably exploit the low precision arithmetic available on modern hardware.

The thesis is organized as follows. In Chapter 1 we define basic concepts on matrix and tensor LRA and the related work on this topic. Then, Chapters 2 to 4 are dedicated to the three main contributions of the thesis, respectively. Finally, we conclude with a summary of the contributions made in this thesis and their possible perspectives in Chapter 5.

1 - Background

This chapter provides all elements that will be used in this thesis and work related to the subject of computing Low-rank approximations (LRA) on both matrices or tensors.

1.1 . Basic notations and definitions

Tensors denoted $\mathcal{X}$ along this chapter are the generalization of matrices, where a matrix is a two-dimensional array of data, and a tensor is a multidimensional array of data with several dimensions greater than two. The order of a tensor is its number of dimensions, and the size of a dimension is the number of elements it contains. Figure 1.1 represents tensors from the lowest order (scalars, on the left) to the highest order (tensors, on the right). Tensors' elements are indexed by the order they have (as matrices) ; for example, the element $x_{i,j,k}$ is the element at the ith row, jth column, and kth depth of the tensor $\mathcal{X}$.

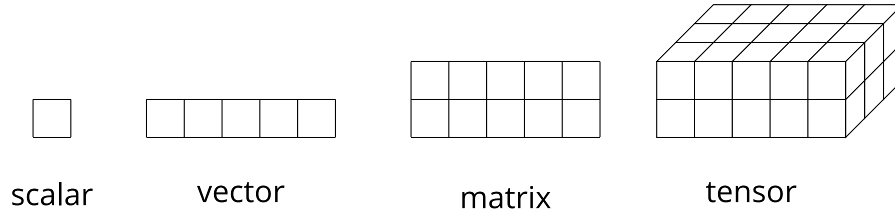


Figure 1.1 – Standard representation of data structure.

The standard norm associated with tensor is the *Frobenius* norm, defined as

$$\|\mathcal{X}\|_F = \sqrt{\sum_{i_1,i_2,\ldots,i_d=1}^{n_1,n_2,\ldots,n_d} x^2_{i_1,i_2,\ldots,i_d}}. \tag{1.1}$$

We omit the subscript F when the context is clear, e.g. $\|\cdot\| = \|\cdot\|_F$, and apply this norm also for matrices. Frobenius norm has some additional properties like the sub-multiplicative property : $\|\mathcal{X}\mathcal{Y}\| \leq \|\mathcal{X}\|\|\mathcal{Y}\|$; moreover we have $\|I_n\| = \sqrt{n}$, where I_n is the identity matrix of order n. This thesis will use the Frobenius norm to measure our error's approximation ; especially the *relative error* will be used, which measures the quality of the approximation, where $\mathcal{Y}$ is an approximation of $\mathcal{X}$ and is defined as

$$\text{Err}(\mathcal{Y}) = \frac{\|\mathcal{X} - \mathcal{Y}\|}{\|\mathcal{X}\|}. \tag{1.2}$$

The main reason for using relative error is the "scale independent", meaning scaling $\mathcal{X}$ and $\mathcal{Y}$ by a factor $\alpha \in \mathbb{R}$ will not change the error.

Addition of tensors are done element-wise such that $\mathcal{X} + \mathcal{Y} = \mathcal{Z}$ implies

$$\mathcal{Z}_{i_1,\dots,i_d} = \mathcal{X}_{i_1,\dots,i_d} + \mathcal{Y}_{i_1,\dots,i_d}, \tag{1.3}$$

where $\mathcal{X}, \mathcal{Y}$, and $\mathcal{Z}$ are in $\mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$. It is straightforward to consider subtraction to be the inverse operation of addition.

Multiplication of tensors, also called *contraction*, can be done along one or more common dimensions to generate a new tensor without dimensions contracted. Given $\mathcal{X} \in \mathbb{R}^{m_1 \times \cdots \times r_1 \times \cdots \times m_d}$, and $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times r_1 \times \cdots \times n_f}$ two tensors with a common dimension r_1, the contraction of them along the dimension r_1 is denoted $\times_{\{r_1\}}$ and produces a new tensor $\mathcal{Z} = \mathcal{X} \times_{\{r_1\}} \mathcal{Y}$, where $\mathcal{Z} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_f}$. Each element of the new tensor $\mathcal{Z}$ is computed as

$$\mathcal{Z}_{i_1,\dots,i_d,j_1,\dots,j_f} = \sum_{k=1}^{r_1} \mathcal{X}_{i_1,\dots,i_d,k} \mathcal{Y}_{k,j_1,\dots,j_f}. \tag{1.4}$$

If the dimension of the contraction is evident from the context, we omit the symbol, e.g., $\mathcal{Z} = \mathcal{X}\mathcal{Y}$.

The *orthogonality* of a tensor is defined along one or more dimensions of the tensor ; for example if the tensor $\mathcal{X}$ is orthogonal along the ith dimension, then the contraction of the tensor with a tensor $\mathcal{Y}$ along the ith dimension gives $\|\mathcal{X}\mathcal{Y}\| = \|\mathcal{Y}\|$.

Another operation is the *concatenation* of tensors, which is the operation to concatenate two tensors along a subset of dimensions. Figure 1.2 top represent this concatenation where the tensor $\mathcal{X} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$ can be concatenated with $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ along the diagonal of the tensor $\mathcal{Z} \in \mathbb{R}^{(m_1+n_1) \times \cdots \times (m_d+n_d)}$ (for $d = 3$). For each dimension $i_k = m_k + n_k, \forall k \in \{1, \dots, d\}$, elements of the tensor $\mathcal{Z}$ is computed as

$$\mathcal{Z}_{i_1,\dots,i_d} = \begin{cases} \mathcal{X}_{i_1,\dots,i_d}, & \text{if } i_k \leq m_k, \forall k \in \{1,\dots,d\} \\ \mathcal{Y}_{i_1-m_1,\dots,i_d-m_d}, & \text{if } i_k > m_k, \forall k \in \{1,\dots,d\} \\ 0, & \text{otherwise} \end{cases} \tag{1.5}$$

If $\mathcal{X}$ and $\mathcal{Y}$ have common dimensions, the concatenation can be done along those dimensions instead, as shown at the bottom of Figure 1.2.

The *reshape* operation modifies the shape of a tensor along given dimensions as illustrated in Figure 1.3. From left to right, the tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ is reshaped into a matrix $X \in \mathbb{R}^{n_1 \times n_2 n_3}$ by permutation of data ; this application also represents the *matricize* method. The matricize operation of tensor $\mathcal{X} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_f}$ into $X \in \mathbb{R}^{m \times n}$, where $m = m_1 \times \cdots \times m_d$ and $n = n_1 \times \cdots \times n_f$, is defined as :

$$X_{i,j} = \mathcal{X}_{i_1,\dots,i_d,j_1,\dots,j_f}, \quad \text{where } \begin{cases} i = 1 + \sum_{k=1}^{d} (i_k - 1) \prod_{l=1}^{k} m_l \\ j = 1 + \sum_{k=1}^{f} (j_k - 1) \prod_{l=1}^{k} n_l \end{cases} \tag{1.6}$$

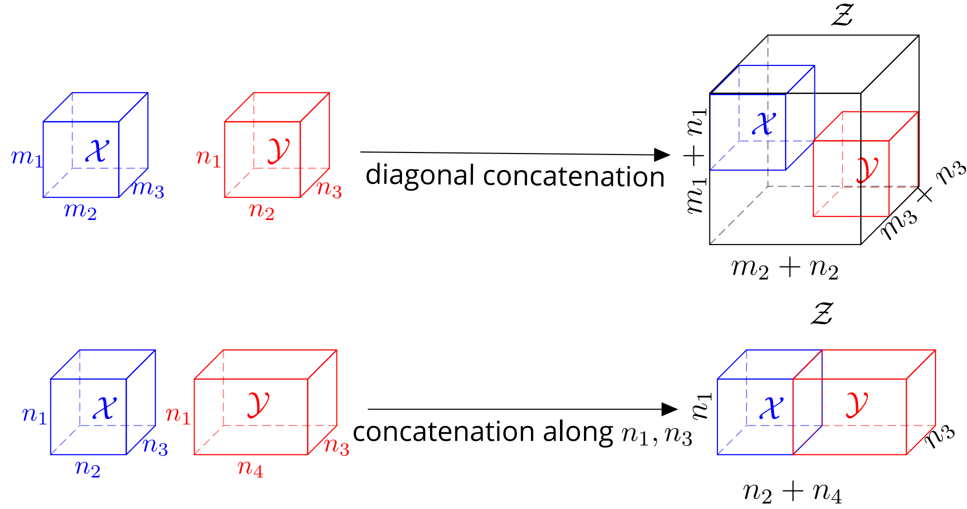


Figure 1.2 – Concatenation of two tensors $\mathcal{X} \in \mathbb{R}^{m_1 \times m_2 \times m_3}$, and $\mathcal{Y} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ along the diagonal of $\mathcal{Z} \in \mathbb{R}^{(m_1+n_1)\times(m_2+n_2)\times(m_3+n_3)}$ (top). Concatenation of two tensors $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, and $\mathcal{Y} \in \mathbb{R}^{n_1 \times n_4 \times n_3}$ along the common dimension n_1 and n_3 to obtain $\mathcal{Z} \in \mathbb{R}^{n_1 \times (n_2+n_4) \times n_3}$ (bottom).

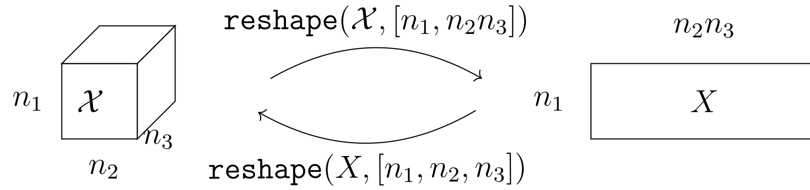


Figure 1.3 – Application of `reshape` kernels on $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ (as matricize kernel) to obtain a matrix $X \in \mathbb{R}^{n_1 \times n_2 n_3}$ (left); then apply a `reshape` kernel on X (as tensorize kernel) to bring back $\mathcal{X}$.

In the other direction, the matrix X is reshaped into the tensor $\mathcal{X}$ by another permutation of data; this application also represents the *tensorize* method. The tensorize operation of matrix $X \in \mathbb{R}^{m \times n}$ into $\mathcal{X} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_f}$, where $m = m_1 \times \cdots \times m_d$ and $n = n_1 \times \cdots \times n_f$, is defined as :

$$\mathcal{X}_{i_1,\ldots,i_d,j_1,\ldots,j_f} = X_{i,j}, \quad \text{where} \begin{cases} i_k = \left\lfloor \frac{i}{\prod_{l=1}^{k-1} m_l} \right\rfloor & \text{mod } m_k, \\ j_k = \left\lfloor \frac{j}{\prod_{l=1}^{k-1} n_l} \right\rfloor & \text{mod } n_k. \end{cases} \tag{1.7}$$

1.2 . Low-rank approximation arithmetic for matrices

The rank of matrix $X \in \mathbb{R}^{m\times n}$ is defined by the smaller integer r such they exist factors $U \in \mathbb{R}^{m\times r}$, and $V \in \mathbb{R}^{n\times r}$, such as $X = UV^T$. This definition of the rank, denoted $\texttt{rank}(X)$, is the non-numerical (or exact) rank of a matrix X [26]. On the other hand, for any *relative accuracy* $\varepsilon > 0$, the numerical rank of X is the smallest integer r_ε such that they exist factors $U \in \mathbb{R}^{m\times r_\varepsilon}$, and $V \in \mathbb{R}^{n\times r_\varepsilon}$, such that

$$\|X - UV^T\| \leq \varepsilon \|X\|. \tag{1.8}$$

Hereinafter we denote r_ε simply as r when ε is clear from the context.

Low-rank approximation is the method to decompose a matrix $X \in \mathbb{R}^{m\times n}$ into smaller factors $U \in \mathbb{R}^{m\times r}$, $V \in \mathbb{R}^{n\times r}$ with the smallest numerical rank r possible, as shown in Figure 1.4.

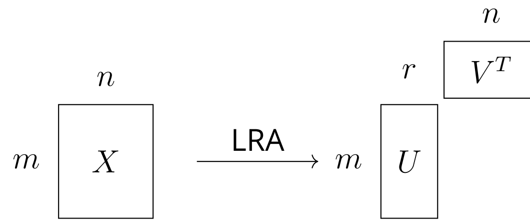


Figure 1.4 – Low-rank approximation of a matrix $X \in \mathbb{R}^{m\times n}$ into factors $U \in \mathbb{R}^{m\times r}, V \in \mathbb{R}^{n\times r}$.

1.2.1 . Singular value decomposition

A first tool to compute the LRA of a matrix is the singular value decomposition (SVD), which is a well-known method to compute the optimal LRA of a matrix. The SVD of a matrix $X \in \mathbb{R}^{m\times n}$ is defined as $X = U\Sigma V^T$, where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{m\times n}$ is a *rectangular diagonal matrix* with the singular values of X. The singular values are ordered in decreasing order on the diagonal of Σ. For any unitarily invariant norm, "the" optimal LRA algorithm for matrices is to compute their SVD and truncate it to the desired accuracy, a result known as Eckart-Young theorem [27]. Specifically, given the SVD $X = U\Sigma V^T$, the optimal rank-k approximation of X is

$$\arg \min_{\substack{M\in\mathbb{R}^{m\times n}\\ \texttt{rank}(M)=k}} \|X - M\| = U_k \Sigma_k V_k^T, \tag{1.9}$$

where $U_k\Sigma_k V_k^T$ is the truncated SVD of X, formed by the first k singular vectors and values only, Figure 1.5 represent this decomposition.

Even if the SVD is optimal, the computation is expensive with a cost $O(mn^2)$ flop.

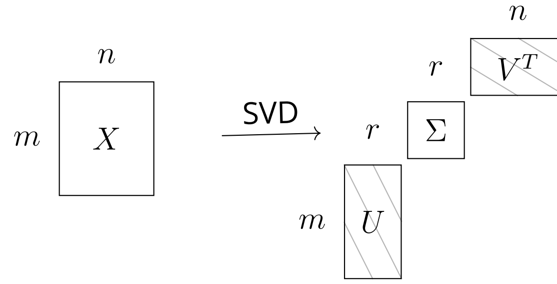


Figure 1.5 – SVD *decomposition* of a matrix $X \in \mathbb{R}^{m\times n}$ into factors $U \in \mathbb{R}^{m\times r}, \Sigma \in \mathbb{R}^{r\times r}$, and $V \in \mathbb{R}^{n\times r}$ where U and V are orthogonal (line inside).

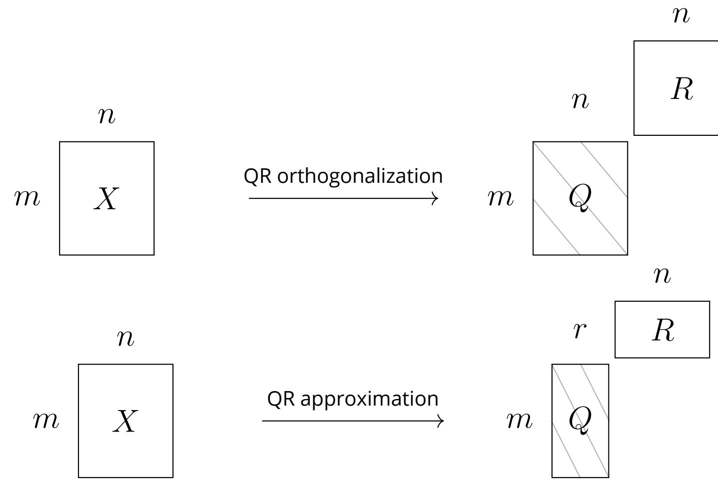


Figure 1.6 – Representation of QR *orthogonalization* of $X \in \mathbb{R}^{m\times n}$ into factors $Q \in \mathbb{R}^{m\times n}, R \in \mathbb{R}^{n\times n}$ (top) and QR *approximation* of $X \in \mathbb{R}^{m\times n}$ into factors $Q \in \mathbb{R}^{m\times r}, R \in \mathbb{R}^{r\times n}$ (bottom), where Q is orthogonal (line inside).

1.2.2 . QR decomposition

On the other hand, a cheaper approximation alternative to the SVD are the QR methods, which are also well-known for LRA on matrices with better performance in terms of computational speed. QR decomposition encompasses various different methods, in this thesis we focus on *Householder QR* one [26]. It decomposes a matrix $X \in \mathbb{R}^{m\times n}$ into factors $Q \in \mathbb{R}^{m\times n}$ and $R \in \mathbb{R}^{n\times n}$ where Q is an orthonormal matrix and R a upper triangular matrix (Figure 1.6).

In the context of LRA, it is beneficial to add a column pivoting phase in Householder QR decomposition to obtain a *QR Column Pivoting (QRCP)* method such as

$$XP = QR, \tag{1.10}$$

where $P \in \mathbb{R}^{n \times n}$ is a permutation matrix [28, 29]. Thanks to pivoting, the QRCP decomposition can be used as a rank-revealing algorithm because the norm of the trailing submatrix at each step k of the QR factorization, $\|XP - Q_k R_k\|$, is monotonically decreasing. Therefore, we can stop the QR factorization as soon as this norm becomes smaller than the target tolerance ε. We obtain

$$X \approx Q_k V_k^T, \quad Q_k \in \mathbb{R}^{m \times k}, \quad V_k = P R_k^T \in \mathbb{R}^{n \times k}, \tag{1.11}$$

where $k = \mathtt{rank}(X, \varepsilon)$. To avoid the need to apply the permutation P each time we need to apply X, we form and store $V_k = P R_k^T$.

Algorithm 1.1 QRCP decomposition.

Input : $X \in \mathbb{R}^{m \times n}$, a target accuracy ε.
Output : $Q_k \in \mathbb{R}^{m \times k}$ and $V_k \in \mathbb{R}^{n \times k}$ such that $X \approx Q_k V_k^T$.

1 : **for** $j = 1 : n$ **do**
2 : $\quad colnorm(j) = \|X_{(:,j)}\|$
3 : **end for**
4 : **for** $k = 1 : \min(m, n)$ **do**
5 : $\quad$ **if** $\sqrt{colnorm(k:n)^2} \leq \varepsilon$ **then**
6 : $\quad\quad k = k - 1$
7 : $\quad\quad$ **break**
8 : $\quad$ **end if**
9 : $\quad p = \max(colnorm(k:n))$
10 : $\quad$ Swap columns k and p of R and $colnorm$
11 : $\quad v = R(k:n, k)$
12 : $\quad \sigma = \mathtt{sign}(v(1))\|v\|$
13 : $\quad v(1) = v(1) + \sigma;$
14 : $\quad v = v/\|v\|;$
15 : $\quad Q_{(:,k:n)} = Q_{(:,k:n)} - 2Q_{(:,k:n)} v v^T$
16 : $\quad R_{(k:n,:)} = R_{(k:n,:)} - 2 v v^T R_{(k:n,:)}$
17 : $\quad colnorm(k+1:n) = \sqrt{colnorm(k+1:n)^2 - R(k, k+1:n)^2}$
18 : **end for**
19 : $Q_k = Q_{(:,1:k)}$
20 : $V_k = P R^T_{(1:k,:)}$

Algorithm 1.1 describes the QRCP decomposition of a matrix $X \in \mathbb{R}^{m \times n}$ at a target accuracy ε. First, we compute the norm of each column of the matrix X in Lines 1 and 3 as a shutoff parameter. Then, for each iteration k, we verify if the norm of the trailing submatrix is smaller than the target accuracy ε. If this is the case, we stop the algorithm and return the factors Q_k and V_k by taking $k = k - 1$ as the previous iteration larger than ε (Line 6). Otherwise, we compute the Householder vector (that corresponds to the Householder

reflector $I - 2vv^T)$ v in Lines 9 and 14 to update the matrix Q and R as shown in Line 15 and Line 16. The Line 17 updates the $colnorm$ variables from $k+1$ to n; note that in practice Line 17 can suffer from severe numerical cancellation and should be implemented in a more careful way as advised in [30].

The performance of the QRCP method turn around $4mnk + o(mnk)$ flops [24], where k is the rank of the matrix.

1.2.3 . Randomized methods

Algorithm 1.2 Randomized low-rank approximation fixed-rank variant.

Input : $A \in \mathbb{R}^{m \times n}$, the target rank k, the oversampling p.
Output : $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$.

1 : $\Omega \leftarrow \texttt{randn}(n, k+p)$
2 : $B \leftarrow A\Omega$
3 : $Q \leftarrow \texttt{qr}(B)$
4 : $YZ^T \leftarrow \texttt{lra}(A^TQ, k)$
5 : $X = QZ$

Among the many possible methods to compute LRA, randomized ones have encountered much success due to their ability to mainly rely on efficient matrix-matrix products. In this thesis, we focus on randomized LRA based on Gaussian sampling [22], as outlined in Algorithm 1.3. This method generates a random Gaussian matrix $\Omega \in \mathbb{R}^{n \times \ell}$ and projects the matrix A onto it by computing the matrix-matrix product $B = A\Omega$ (line 2). The dimension ℓ is equal to $k + p$, where p is a small oversampling parameter (typically, $p \leq 10$). Then, the matrix B is orthonormalized using the QR factorization and only keeps the orthonormal part Q (Line 3). This matrix Q satisfies $A \approx QQ^TA$. And therefore, the LRA of A can be computed by decomposing the matrix A^TQ (Line 4). Indeed, if we compute $YZ^T \approx A^TQ$ then $A \approx QQ^TA \approx XY^T$, where $X = QZ$ (Line 5). The kernel `lra` in Line 4 can be any standard LRA method such as SVD or QRCP method.

We note that many alternative variants of Algorithm 1.2 are possible; for example, the sampling may be performed differently (e.g., via a fast Fourier transform), or we may compute specific types of LRA (e.g., SVD) by further decomposing inside the `lra` kernel. In addition, while Algorithm 1.2 is a fixed-rank algorithm, fixed-accuracy variants have also been proposed [22, 31], in which an accuracy threshold ε is prescribed and the rank k is adaptively discovered by the algorithm.

Algorithm 1.3 is such an adaptive algorithm—increasing the rank until the prescribed accuracy ε is reached—proposed by Martinsson and Voronin [31].

The algorithm consists of two phases. The first phase (Lines 3 to 11) iteratively builds a QB decomposition of the matrix such that $\|A - QB\| \leq \varepsilon\|A\|$,

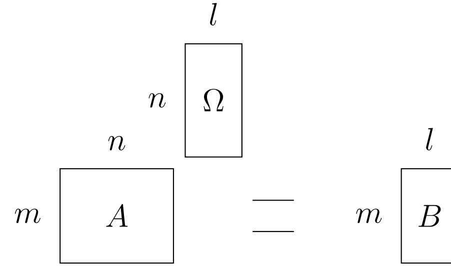


Figure 1.7 – Randomized part of randomized low-rank approximation of $A \in \mathbb{R}^{m \times n}$.

Algorithm 1.3 Randomized SVD fixed-accuracy decomposition.

Input : $A \in \mathbb{R}^{m \times n}$, a target accuracy ε, and a block size b.
Output : $X \in \mathbb{R}^{m \times k}$ and $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$.

1 : Initialize Q and B to empty matrices.
2 : $n_A = \|A\|$
3 : **repeat**
4 : Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n \times b}$.
5 : $Y = A\Omega$
6 : $Q_b = \mathtt{qr}(Y - Q(Q^T Y))$
7 : $B_b = Q_b^T A$
8 : $Q \leftarrow [Q\ Q_b]$
9 : $B \leftarrow \begin{bmatrix} B \\ B_b \end{bmatrix}$
10 : $A \leftarrow A - Q_b B_b$
11 : **until** $\|A\| \leq \varepsilon n_A$
12 : $[\overline{X}, Y] \leftarrow \mathtt{Svd}(B, \varepsilon)$.
13 : $X = Q\overline{X}$

where $Q \in \mathbb{R}^{m \times k}$ has orthonormal columns and where the dimension k is hopefully close to $\mathtt{rank}(A, \varepsilon)$. To do so, a random Gaussian matrix $\Omega \in \mathbb{R}^{n \times b}$ is drawn for a given block size b and used to sample the matrix $Y = A\Omega$ (Line 5). Then Y is added to the basis Q while preserving the orthonormality : this can for example be accomplished using the Gram-Schmidt algorithm (Line 6). This process is repeated until the columns of Q are a sufficiently good approximation of the column space of A, that is, until $\|A - QQ^T A\|$ is small. In order to efficiently compute this quantity, the matrix $B = Q^T A$ is formed block by block (Line 7). Then, in the second phase (Lines 12 and 13), a truncated SVD of A can easily be obtained by computing the truncated SVD of the lower dimensional matrix $B \in \mathbb{R}^{k \times n}$: if $B = \overline{X}\Sigma V^T$, then $A = XY$ with $X = Q\overline{X}$ and $Y = \Sigma V^T$. In Line 12, we have chosen the SVD as the LRA kernel ; in practice,

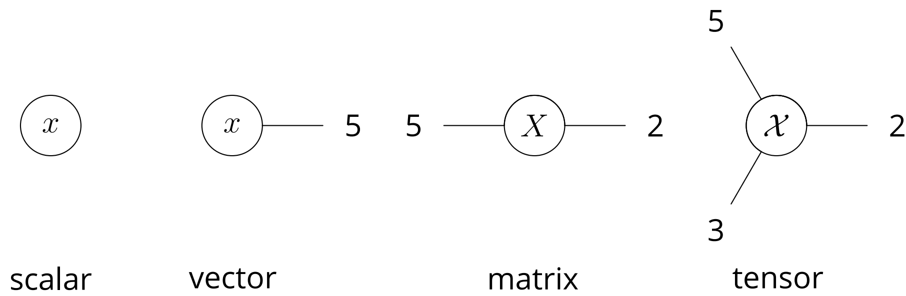


Figure 1.8 – Tensor network diagram of data object.

any LRA kernel with truncation at accuracy ε can be used.

1.3 . Low-rank approximation arithmetic for tensors

The number of elements to store in a tensor is exponential in its order, making it challenging to compute, $O(n^d)$ elements for a d order tensor. This problem called *curse of dimensionality* is tackled by *tensor decomposition* that provide a LRA of a tensor, which separates a high dimensional tensor into lower-dimensional tensors (called factors) with *outer dimensions*—representing the dimension of the original tensor—and *inner dimensions*—common dimensions between factors to generate the original tensor once contracted.

Usually, we use a cubic representation to represent a high-order tensor, as shown in Figure 1.1, but this representation is unsuitable for a very high-order tensor. The tensor network diagram solves this problem, where a circle (named *node*) represents a tensor, and lines represent the dimensions, named *edges* ; thus, a node's degree is the order of the associated tensor. Figure 1.8 show the tensor network diagram of data structure in Figure 1.1.

[TODO]Emm. Ag. (E.A.) : Peut-être expliquer pourquoi on ne parle pas de CP ?

1.3.1 . Tree tensor networks

The Tree Tensor Network (TTN) $\boldsymbol{\mathcal{X}}$ represents $\mathcal{X}$ as a tree—denoted with bold calligraphic upper case letters—, a connected acyclic undirected graph [32], as shown in Figure 1.9. There are two types of edges in a TTN : inner edges that connect two nodes and outer (or "dangling") edges that are only connected to one node. Tree nodes can have both inner and outer edges. Two nodes connected by an inner edge represent the contraction of the corresponding tensors along the dimension that connects them. We define *inner nodes* as nodes connected to at least two other nodes and *outer nodes* as nodes connected to only one node with at least one outer dimension. Therefore, the whole network represents a *full* tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ of order d, the

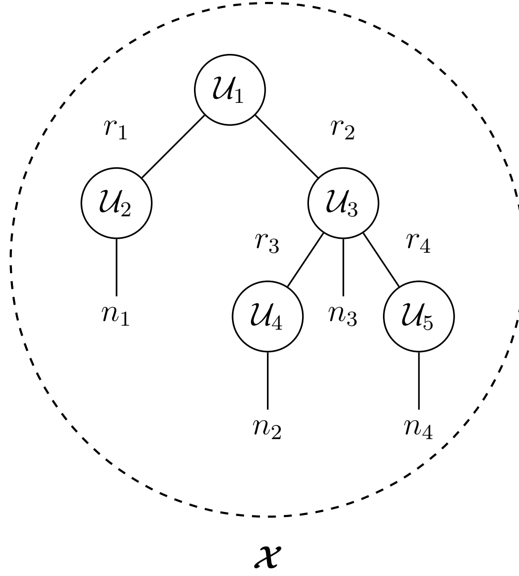


Figure 1.9 – Tree tensor network representation

number of dangling edges of $\boldsymbol{\mathcal{X}}$. This representation can also express all the other tensor formats we investigated during this thesis.

We define $\|\boldsymbol{\mathcal{A}}\| = \|\mathcal{A}\|$, that is, the norm of the network is the norm of the full tensor it represents. As a result the usual submultiplicativity of the Frobenius norm is preserved for tensor networks :

$$\|\boldsymbol{\mathcal{A}\mathcal{B}}\| = \|\mathcal{A}\mathcal{B}\| \leq \|\mathcal{A}\|\|\mathcal{B}\| = \|\boldsymbol{\mathcal{A}}\|\|\boldsymbol{\mathcal{B}}\|. \tag{1.12}$$

Like this, we can define the *semi-orthogonality* of a node in a TTN as the orthogonality of the tensor along a set of specific dimension.

Tucker format The Tucker format represents a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ as a core tensor $\mathcal{G} \in \mathbb{R}^{r_1 \times \cdots \times r_d}$ linked with d matrices $U^{(1)} \in \mathbb{R}^{n_1 \times r_1}$, . . ., $U^{(d)} \in \mathbb{R}^{n_d \times r_d}$ representing the leaves of the network with orthonormal columns.A standard application is to have all matrices $U^{(i)}$ orthonormal and keep the non orthogonal part in the core tensor $\mathcal{G}$. Each element of a Tucker tensor is given by

$$\mathcal{X}_{i_1,\ldots,i_d} = \sum_{\alpha_1,\ldots,\alpha_d}^{r_1,\ldots,r_d} \mathcal{G}_{\alpha_1,\ldots,\alpha_d} U^{(1)}_{i_1,\alpha_1} \ldots U^{(d)}_{i_d,\alpha_d}. \tag{1.13}$$

In TTN representation, the Tucker format is a tree with one inner nodes and d outer nodes, where the outer nodes are connected to the core tensor along their inner dimension r_i, represented in top-right part of Figure 1.10.

Tensor Train format The TT format [9] represent a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ as a sequence of 3rd-order tensors $\mathcal{U}^{(1)} \in \mathbb{R}^{r_0 \times n_1 \times r_1}$, . . ., $\mathcal{U}^{(d)} \in \mathbb{R}^{r_{d-1} \times n_d \times r_d}$ where $r_0 = r_d = 1$. The first and third dimensions represent the ranks of the

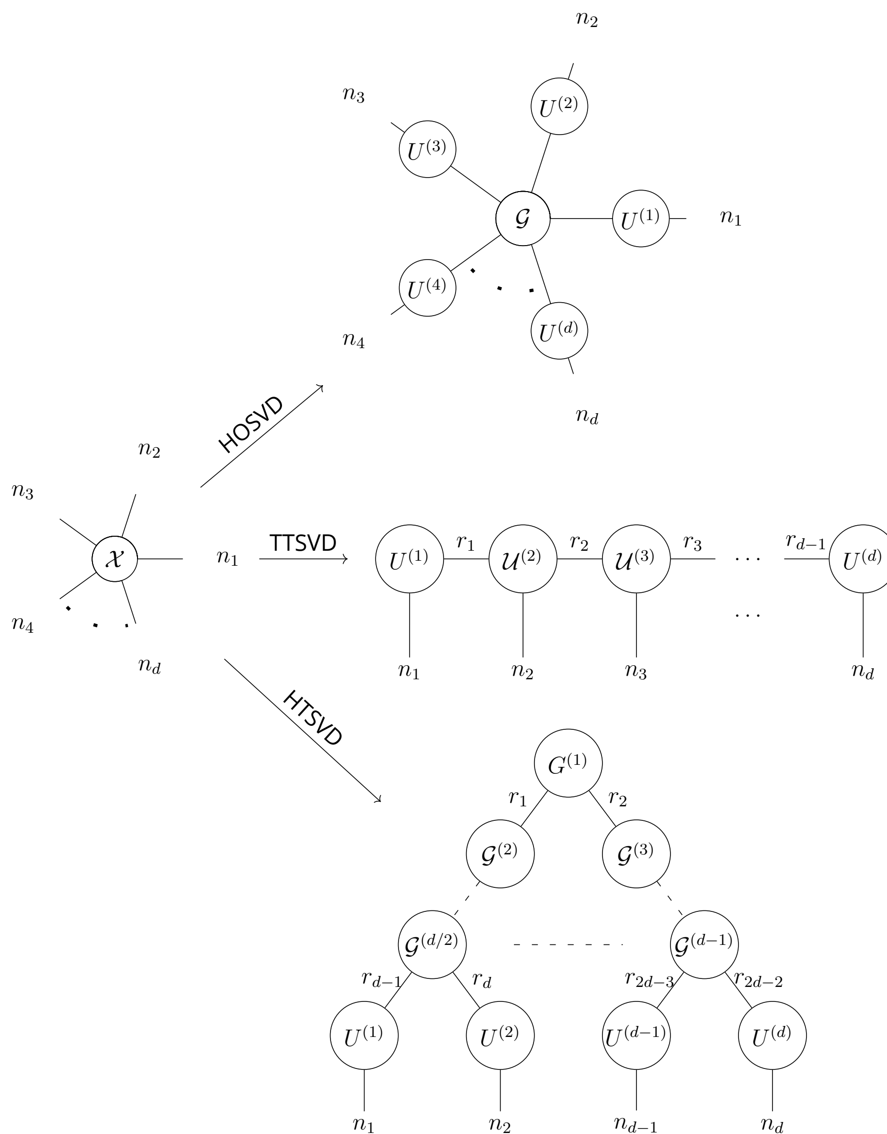


Figure 1.10 – A d-order tensor (left) and its representation in Tucker, tensor-train (TT), and hierarchical Tucker (HT) decomposition (right) using tensor network diagram.

tensor (inner dimension), and the second dimension is the outer dimension. Each element of a TT tensor is given by

$$\mathcal{X}_{i_1,\dots,i_d} = \sum_{\alpha_0,\dots,\alpha_d}^{r_0,\dots,r_d} U^{(1)}_{\alpha_0,i_1,\alpha_1} \mathcal{U}^{(2)}_{\alpha_1,i_2,\alpha_2} \dots \mathcal{U}^{(d-1)}_{\alpha_{d-2},i_{d-1},\alpha_{d-1}} U^{(d)}_{\alpha_{d-1},i_d,\alpha_d}. \tag{1.14}$$

In TTN representation, the TT format is a "degenerate" binary tree with a single path, and nodes connect only with two other nodes (except for bound nodes), represented in middle-right part of Figure 1.10.

Hierarchical Tucker format The HT format [13, 14] is a specific TTN [32] without outer dimension in inner nodes, represented in bottom-right part of Figure 1.10. Thanks to its tree structure, the exponential growth of the Tucker format is reduced to a polynomial growth where the degree of the polynomial depends on the tree's arity. The arity of the tree can be arbitrarily high, but in this thesis, we will focus on binary trees, which means a node can be connected up to three other nodes. In this case, leaf nodes of the tree are matrices $U^{(1)} \in \mathbb{R}^{n_1 \times r_{d-1}}, \dots, U^{(d)} \in \mathbb{R}^{n_d \times r_{2d-2}}$ and the rest are 3rd-order tensors $\mathcal{G}^{(1)} \in \mathbb{R}^{r_0 \times r_1 \times r_2}, \dots, \mathcal{G}^{(d-1)} \in \mathbb{R}^{r_{d-2} \times r_{2d-3} \times r_{2d-2}}$, where $\mathcal{G}^{(1)}$ is the root and $r_0 = 1$. Each element of a HT tensor is given by

$$\mathcal{X}_{i_1,\dots,i_d} = \sum_{\alpha_0,\dots,\alpha_{2d-2}}^{r_0,\dots,r_{2d-2}} \mathcal{G}^{(1)}_{\alpha_0,\alpha_1,\alpha_2} \dots \mathcal{G}^{(d-1)}_{\alpha_{d-2},\alpha_{2d-3},\alpha_{2d-2}} U^{(1)}_{i_1,\alpha_{d-1}} \dots U^{(d)}_{i_d,\alpha_{2d-2}}. \tag{1.15}$$

Each format has its own advantages and disadvantages, and the choice of the format depends more on the application and the tensor's properties the user wants. In terms of storage cost, the Tucker format represents a node of order d with small dimensions r and matrices of shape $n \times r$; thus the storage is $O(dnr + r^d)$ instead of $O(n^d)$ from the full tensor. The TT format is more efficient in terms of storage, with a cost of $O((d-2)nr^2 + 2nr)$, compared to HT which is the heavier when $d > 2$, with a total of $O(dnr + (d-2)r^3 + r^2)$ elements to store.

1.3.2 . Kernels for low-rank approximation on tensors

Full tensor to LRA The LRA on the tensor is a generalization of the LRA on the matrix, with the decomposition specification providing a tensor format different from the original tensor. A LRA on a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$ reduces the tensor into smaller factors $\mathcal{U}^{(1)} \in \mathbb{R}^{n_1 \times r_1}$, $\mathcal{U}^{(2)} \in \mathbb{R}^{n_2 \times r_2}$, ..., $\mathcal{U}^{(d)} \in \mathbb{R}^{n_d \times r_d}$, where $r_i \leq n_i$ is the rank of $\mathcal{X}$, which will be detailed later. The number of factors is at least equal to the order of the tensor d, and the number of inner dimensions r depends on the decomposition. The high-order singular value decomposition (HOSVD) [12, 33], Tensor-Train Singular Value Decomposition (TTSVD), and hierarchical Tucker singular value decomposition (HTSVD) are decomposition methods to represent the Tucker format [11], TT format [9], and HT format [14, 34] respectively.

Algorithm 1.4 `lra` algorithm for a tensor.

Input : a tensor $\mathcal{X}$ and a target accuracy ε.
Output : $\boldsymbol{\mathcal{X}} = \mathcal{U}_1, \ldots \mathcal{U}_m$: a low-rank tensor decomposition of $\mathcal{X}$.

1: $\bar{\varepsilon} \leftarrow \varepsilon/\sqrt{d}$
2: **for** each node $\mathcal{U}_i = \mathcal{U}_1, \ldots \mathcal{U}_{m-1}$ of the prescribed structure **do**
3: $\quad X = \texttt{reshape}(\mathcal{X}, r)$ {for a suitable dimension r}
4: $\quad UY = \texttt{lra}(X, \bar{\varepsilon})$
5: $\quad \mathcal{U}_i = \texttt{reshape}(U)$
6: $\quad \mathcal{X} = \texttt{reshape}(Y)$
7: **end for**
8: $\mathcal{U}_m = \mathcal{X}$ {The remaining $\mathcal{X}$ is the root node.}

Algorithm 1.4 is a high-level pseudocode of a fixed-accuracy LRA algorithm for a tensor $\mathcal{X}$. Knowing the algorithms are independent of the tensor format, we use the TTN format to represent all the above-mentioned algorithms. It encompasses the three decomposition formats HOSVD [12, 33], TTSVD [9], and HTSVD [13, 14] with adjusted arrangements on kernels `reshape` and `lra` to obtain the desired format. In particular, Line 3 is equivalent to the matricize kernel for the tensor $\mathcal{X}$, and Lines 5 and 6 are tensorize kernels. Figure 1.10 represent each decomposition starting from a tensor $\mathcal{X}$ with the TTN representation (left part), where the corresponding `lra` method yields a network of core tensors (right part). This thesis also calls the `lra` kernel the `compress` kernel in a TTN environment.

The definition of the numerical rank explained in Section 1.2 extends to the Tucker, TT, and HT representations with $\texttt{rank}(X, \varepsilon)$ and $\texttt{rank}(X)$ as the vector of length m whose coefficients correspond to the inner dimensions connecting the underlying matrices and/or tensors in the decomposition, where m is the number of inner dimension of the corresponding format. Thus, the rank vector is determined by a vector of error tolerances ε_i, $i = 1\colon m$, whose sum yields the target accuracy $\varepsilon = \sqrt{\sum_{i=1}^{m} \varepsilon_i^2}$. In practice, we use $\varepsilon_i = \varepsilon/\sqrt{m}$.

LRA to full tensor To reconstruct the full tensor $\mathcal{X}$ from factors $\mathcal{F}_X$, we use the *full* method which is a sequence of contractions of all factors along their inner dimensions to generate the full tensor $\mathcal{X}$, as shown in Algorithm 1.5. In a TTN environment we call the full as `decompress` kernel.

Orthogonalization We need to orthogonalize all factors except one to quickly analyze the results of the LRA on a tensor. Hence, we possess a `orthog` kernel for a TTN, where all nodes except one become orthogonal along a prescribed dimension. Algorithm 1.6 is a high-level pseudocode of the orthogonalization algorithm for a TTN. The orthogonalization is performed by a QR factorization on the matricized tensor in the direction of the parent node, as

Algorithm 1.5 `full` algorithm

Input : $\mathcal{F}_X$: factors.
Output : $\mathcal{X}$: the full tensor.

1 : Choose a node $\mathcal{U}$.
2 : **while** $\mathcal{U}$ is not the only node left **do**
3 :   Let $\mathcal{V}$ be a node adjacent to $\mathcal{U}$.
4 : $\mathcal{U} = \mathcal{U} \times_{\{r\}} \mathcal{V}$ {for a suitable dimension r}
5 : **end while**

shown in Line 7. Then, the orthogonal part is tensorize to update the node $\mathcal{U}$ (Line 8). The non-orthogonal part R is contracted with the parent node $\mathcal{P}$ to transfer the non-orthogonal part to the parent node $\mathcal{P}$ until we reach the root (Line 9).

Algorithm 1.6 `orthog` algorithm for tensor

Input : $\boldsymbol{\mathcal{X}}$: a tensor.
Output : $\boldsymbol{\mathcal{X}}$: the tensor orthogonalized.

1 : **for** each node $\mathcal{U}$ from leaves to root **do**
2 :   **if** $\mathcal{U}$ is the root **then**
3 :   **return**
4 :   **else**
5 : $\mathcal{P} = \texttt{parent}(\mathcal{U})$
6 : $U = \texttt{reshape}(\mathcal{U}, \mathcal{P})$ {Matricize in the direction of the parent}
7 : $QR = \text{qr}(U)$ {QR factorization ; Q is semi-orthogonal}
8 : $\mathcal{U} = \texttt{reshape}(Q)$
9 : $\mathcal{P} = R \times_{\{r\}} \mathcal{P}$ {Contraction non-orthgonal part with parent for a suitable dimension r}
10 :   **end if**
11 : **end for**

Low rank arithmetic on TTN Addition in TTN format is $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}}$ where the network represents the tensor $\mathcal{C} = \mathcal{A} + \mathcal{B}$; note that its nodes are *not* the sum of the nodes of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, but rather their diagonal concatenation along the dimensions corresponding to inner edges.

An important case where the addition is simplified is when $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ only differ by one node : if $\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{X}}_1 \mathcal{C} \boldsymbol{\mathcal{X}}_2$ and $\boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}}_1 \mathcal{D} \boldsymbol{\mathcal{X}}_2$, then

$$\boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}}_1 \mathcal{C} \boldsymbol{\mathcal{X}}_2 + \boldsymbol{\mathcal{X}}_1 \mathcal{D} \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1 (\mathcal{C} + \mathcal{D}) \boldsymbol{\mathcal{X}}_2. \tag{1.16}$$

Rounding Some applications require a *rounding* method to reduce the inner dimensions of tensor networks by respecting an error threshold ε, just like the compress algorithm, yet without forming the full tensor [14, 34, 9,

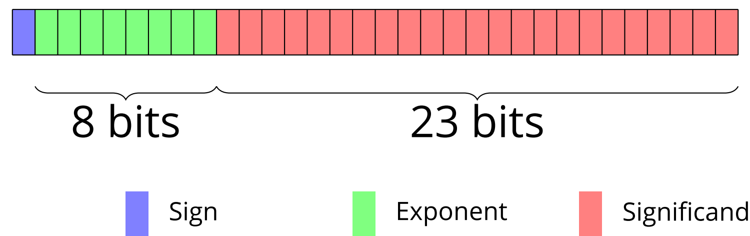


Figure 1.11 – Floating point representation with a example of single precision.

21]. An algorithm for TTN further discussion on its stability will be detailed in Chapter 4.

1.4 . Floating point arithmetic

In computer science, the floating point arithmetic is a tool to represent real numbers in a computer. Compared to exact arithmetic, floating point arithmetic approximates real numbers because it is limited by the number of bits it can use to represent real numbers.

We focus on the Institute of Electrical and Electronics Engineers (IEEE) 754 standard representation of numbers, where we outline its interest in representing numbers in *low precision* and how the low precision can benefit LRA methods.

1.4.1 . Properties

A floating point number x is represented as

$$x = (-1)^s m \times \beta^{e-t}, \tag{1.17}$$

where s is the sign bit, e is the *exponent* (a range between $[e_{\text{min}}, e_{\text{max}}]$), β is the *base*, t the *precision*, and m is the *mantissa* (also call *significand*). Figure 1.11 gives an example of a representation of floating point arithmetic for 32 bits (single precision) with IEEE 754 representation (explain below). We work on binary representation, so $\beta = 2$, and consider *noramlized* system such that $\beta^{t-1} \leq m \leq \beta^t - 1$, to obtain a range of $[\beta^{e_{\text{min}}-1}, \beta^{e_{\text{max}}}(1-\beta^{-t})]$ possible values. In particular, try to represent a number x outside its possible value range provide a specific behavior dependent on the *rounding* rule. The rounding rule is a rule to approximate the value of a number to the nearest representable number, which will introduce *rounding error* corresponding to the inexactness of the value compared to exact arithmetic.

The *accuracy* represents how good the approximation is in terms of relative error (Section 1.1), whereas the *precision* is accuracy obtained by standard operations (addition, subtraction, multiplication, division) [26]. For example,

	Name	Range	Precision	Exponent bits	Mantissa bits	Performance (Tflops/s) on NVIDIA GPU A100 [1]
fp64	double	$10^{\pm 308}$	1×10^{-16}	11	52	19.5
fp32	single	$10^{\pm 38}$	6×10^{-8}	8	23	19.5
fp16	half	$10^{\pm 5}$	5×10^{-4}	5	10	312
bfloat16		$10^{\pm 38}$	4×10^{-3}	8	7	
fp8 (e4m3)	quarter	$10^{\pm 2}$	6×10^{-2}	4	3	NA
fp8 (e5m2)		$10^{\pm 5}$	1×10^{-1}	5	2	

[1] Details in Section 1.4.2

Figure 1.12 – Floating point precision comparison.

taking an infinitely long value, such as $\pi = 3.14159265358979323846\ldots$, defining a "good" accuracy of π is a tricky question depending on multiple factors. The definition strongly depends on the user and its domain, where 4 digits can be enough for some applications, or 10 digits cannot be enough for others. As seen in the previous section, the truncation threshold ε represents this require precision by the user. It also depends on the precision of the computation controlled by the rounding errors, notates $u = \frac{1}{2}\beta^{1-t}$ and calls *unit roundoff*.

We need to know the error of the approximation to control the computation and generate algorithms. We denote $\hat{x}$ the approximation of x in the floating point arithmetic, also call *computed* x, such that

$$\hat{x} = x(1+\delta), \quad \text{with} \quad |\delta| \leq u, \tag{1.18}$$

where u is the *unit roundoff*. Moreover, doing operations on floating point numbers can also generate errors like

$$\hat{z} = x \text{ op } y = (x \text{ op } y)(1+\delta), \quad \text{with} \quad |\delta| \leq u, \tag{1.19}$$

where op represents any standard operation (addition, subtraction, multiplication, division).

We can increase the number of bits in the mantissa or the exponent, which increases the number of representable numbers, but it also increases the storage space and the computational cost. Hence, choosing the number of bits without considering hardware restriction is challenging depending on the application for the best trade-off between accuracy and cost.

1.4.2 . Commonly available floating point arithmetic

The IEEE 754 standard [35] defines the floating point arithmetic used in most modern computers, with much precision available, illustrated in Figure 1.12. The difference between precision is the number of bits used in the exponent and the mantissa, represented in Figure 1.13. Most commonly used

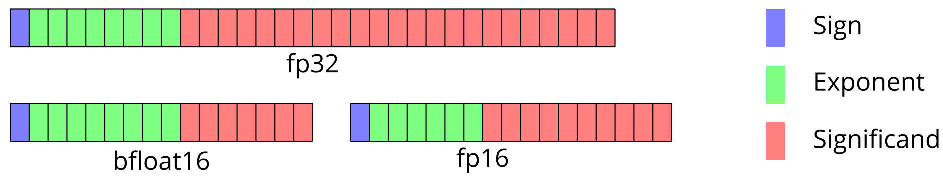


Figure 1.13 – *Low precision* floating point representation.

are the single precision (fp32) and double precision (fp64) formats, which respectively use 32 and 64 bits and are considered in *high precision* domain.

Most computations are done in *high precision* arithmetic, i.e. double precision or single precision, to increase the accuracy of the computation. In contrast, recent technological advances allow for significant speedups using *low precision* arithmetic [18]. Low precision arithmetic is not a fixed term to describe a specific format, but rather a range of formats that use fewer bits than the "standard" high precision of the application we treat.

Recent hardware such as Graphics Processing Unit (GPU)s (NVIDIA, AMD, or Intel) performs well in low precision. Equip with so-called tensor core units that can perform matrix multiplication in low precision fp16 arithmetic up to 16.4 faster than in standard fp32 arithmetic [17], as shown in Figure 1.12. Central Processing Unit (CPU)s also benefit from low precision arithmetic, but it is less relevant than the speed-up from the tensor core unit. Other hardware like Tensor Processing Unit (TPU) [36] or Field-Programmable Gate Array (FPGA) [37] can also use low precision, but they are rarely present in standard hardware. If the above hardware is unavailable, low precision can be simulated with higher precision and still obtain interesting results [38].

Keeping low precision throughout the computation will reduce storage costs and improve performance, but the approximation error will also increase. This motivates the use of mixed precision arithmetic, which is a technique to combine both high and low-precision arithmetic inside computation to tackle the loss of accuracy obtained by the low precision computation. The emergence of low precision on modern hardware has generated much recent interest in mixed precision algorithms, with many successful examples in numerical linear algebra ; the survey [18] gives an overview of this field. In this thesis, we review specifically the state-of-the-art on using mixed precision for LRA.

1.5 . Mixed precision for low-rank approximations

In this section, we highlight the application of mixed precision arithmetic to LRA methods in two folds.

1.5.1 . Mixed precision for low-rank approximation for matrices

Contrary to other linear algebra routines such as the solution of linear systems, there has been relatively little work on designing mixed (or even low) precision algorithms for LRA. Amestoy et al. [39] describe a mixed precision matrix LRA that partitions the low-rank factors into several block columns stored in different precision depending on the singular values of the matrix; this approach can make use of low precision for matrices with rapidly decaying singular values. A similar approach is proposed by Ooi et al. [40] for $\mathcal{H}$-matrices.

It is especially natural to seek to exploit the high-speed low precision available on GPU hardware to accelerate randomized LRA, which mainly rely on matrix multiplications. However, the literature on these methods has mainly focused on either exact arithmetic or fixed precision arithmetic, where all the operations are performed in the same precision. To the best of our knowledge, only three recent papers depart from an exact or fixed arithmetic context to propose mixed precision variants of randomized LRA : Connolly, Higham, and Pranesh [41], Ootomo and Yokota [42], and Buttari, Mary, and Pacteau [43].

Connolly, Higham, and Pranesh [41] propose a mixed precision variant of the adaptive randomized SVD algorithm of Martinsson and Voronin [31]. This variant relies on the observation that the norm of the matrix deflated with the current LRA may rapidly decrease, which makes it possible to switch the computation to lower precision. This observation is linked to the decay of the singular values of the matrix, which is also exploited by Amestoy et al. [39]. Similarly, Buttari, Mary, and Pacteau [43] propose a mixed precision truncated QRCP with either classical or randomized pivoting, which is based on the same principle of exploiting singular value decay. In this article, we do not consider adaptive (fixed-accuracy) variants of randomized LRA and do not assume any decay of the singular values of the matrix.

Ootomo and Yokota [42] propose a mixed precision variant of fixed-rank randomized SVD (Algorithm 1.3). This variant relies on the observation that the random Gaussian matrix Ω can be represented in fp16 arithmetic without endangering the stability of the computation. As a result, the matrix products $B = A\Omega$ can be efficiently performed using GPU tensor cores by computing $B = A_1\Omega + A_2\Omega$, where $A \approx A_1 + A_2$ and both A_1 and A_2 are stored in fp16. This approach exploits multiword arithmetic to emulate fp32 arithmetic using fp16 computations, see also Fasi et al. [44] for further details about multiword arithmetic.

1.5.2 . Mixed precision for low-rank approximation for tensors

There have also been very few attempts to develop mixed precision LRA for tensors. We can mention a recent work by Yang et al. [45] that proposes

an iterative CANDECOMP/PARAFAC (CP) decomposition using mixed precision stochastic gradient descent.

In a context different from LRA, the recent work of Agullo et al. [46] is worth mentioning : they consider the solution of linear systems $Ax = b$ via Generalized Minimal Residual (GMRES), where A can be approximated under tensor format [47]. This approach can then benefit from a mixed precision implementation of GMRES.

2 - Mixed precision iterative refinement for low-rank approximations

2.1 . Introduction

In this chapter, we propose a new method for computing Low-rank approximations (LRA) in mixed precision arithmetic. Our approach is applicable to basically any LRA algorithm, involving either matrices or tensors. It is reminiscent of the Iterative Refinement (IR) framework used for solving linear systems : the idea is to first compute a LRA in low precision, then evaluate the error (or residual) from this first LRA, and re-apply the same LRA kernel to this error term to obtain a correction term that is used to refine the accuracy of the LRA. This can be repeated iteratively to reach any level of desired accuracy. The refined LRA is obtained as the sum of the original low precision LRA and the correction term, and is thus of larger yet still of low-rank. In order to contain the rank growth and maintain the optimal rank throughout the iterations, our method employs a "recompression" strategy that is performed in high precision but whose cost stays asymptotically smaller than that of LRA.

We carry out an error analysis of our method based on a general parameterized error model that only assumes that we have numerically stable implementations of the basic kernels used in our algorithm (LRA, matrix multiplication, and recompression). Under this model, we prove that the method can reach a high accuracy while performing most of the operations in low precision. In particular, we show that the precision used for the LRA kernel—which is the computational bottleneck of the whole method—only affects the convergence speed of the process, but not its attainable accuracy. In order to assess under which conditions we can expect our method to be beneficial, we perform a complexity analysis that measures the cost of the method as a function of the numerical rank of the input as well as the speed ratio between the low and high precision arithmetic. We identify two situations where our method has a strong potential. The first is when the hardware provides fast low precision matrix multiply-accumulate units [17] (see Section 1.4.2), which allow for computing the low precision LRA at very high speed. The second is when the numerical rank of the input is small at low accuracy levels, which means that the singular values of the matrix or tensor are rapidly decaying; in this case, the first iterations of our method becomes inexpensive.

We apply our method to various low-rank matrix and tensor decomposition : singular value decomposition (SVD), QR, Tucker, hierarchical Tucker (HT),and tensor-train (TT). We perform some MATLAB experiments to confirm that our method is able to compute a LRA at any desired level of accuracy,

while mostly using the low precision arithmetic.

The rest of this chapter is organized as follows. In Section 2.2, we describe our main algorithm of IR for LRA and provide the corresponding error and complexity analysis. Then we apply this method to various LRA algorithms, namely QR Column Pivoting (QRCP) and randomized SVD for matrices, and high-order singular value decomposition (HOSVD), Tensor-Train Singular Value Decomposition (TTSVD), and hierarchical Tucker singular value decomposition (HTSVD) for tensors in Section 2.3 and Section 2.4, respectively. We validate our method experimentally in Section 2.5. Finally, we provide concluding remarks in Section 2.6.

2.1.1 . Notations

In this chapter we will denote X the object of interest, a matrix or a tensor, and as F its low-rank factors that we seek to compute. Note that our main IR method described in the next section does not require any specific knowledge or property of the tensor decomposition, and simply denotes its low-rank factors as F for either Tucker, TT, HT formats.

Two types of parameters are used to control the accuracy of the LRA : ε and u. Actually, in our IR method, we will have four parameters ε, u, ε_ℓ, and u_ℓ, where ε_ℓ and u_ℓ denote the truncation threshold and unit roundoff used by the low precision LRA, whereas ε and u denote the truncation threshold and unit roundoff used by the rest of the operations performed in high precision, and correspond to the final target accuracy. Hence, $\varepsilon_\ell \geq \varepsilon$ and $u_\ell \geq u$.

2.2 . Iterative refinement for low-rank approximations

Considering the problem of computing low-rank factors F of a matrix or tensor X satisfying a relative accuracy ε, that is, $\|X - F\| \leq \varepsilon\|X\|$. In finite precision arithmetic, we must carefully select the precision in order to achieve this accuracy. In a uniform precision context (where we perform all computations in the same precision), we must use a precision whose unit roundoff u is safely smaller than ε, that is, $u = \theta\varepsilon$, with $\theta \leq 1$ some parameter. Indeed, we may then expect the computed factors $\widehat{F}$ to satisfy

$$\|X - \widehat{F}\| \leq (\varepsilon + cu)\|X\| = (1 + c\theta)\varepsilon\|X\| \tag{2.1}$$

for some constant c, where the term $c\theta$ accounts for the effect of rounding errors and can be made as small as needed by decreasing θ (decreasing u, that is, increasing the precision).

For applications requiring a relatively high accuracy (small values of ε), this uniform precision approach therefore cannot make use of lower precision. This motivates us to propose a mixed precision method, outlined below, which uses low precision to compute a first approximate set of factors F_0, and then refines them into more accurate factors F_1 as follows.

1. Compute low-rank factors F_0 of X at low accuracy ε_ℓ and in low precision $u_\ell = \theta\varepsilon_\ell$.
2. Compute the error $E = X - F_0$ in high precision u.
3. Compute low-rank factors F_E of E at low accuracy ε_ℓ and in low precision $u_\ell = \theta\varepsilon_\ell$.
4. Update the low-rank factors of X to $F_1 = F_0 + F_E$ in high precision u.

This approach is based on the observation that, while the first factors F_0 will achieve a low accuracy ε_ℓ relative to $\|X\|$,

$$\|X - F_0\| \le \varepsilon_\ell \|X\|, \tag{2.2}$$

the factors F_E of the error E will achieve a low accuracy ε_ℓ relative to $\|E\|$,

$$\|E - F_E\| \le \varepsilon_\ell \|E\|. \tag{2.3}$$

Neglecting for now the effect of rounding errors, we have $\|E\| = \|X - F_0\| \le \varepsilon_\ell \|X\|$, and therefore the combined factors $F_0 + F_E$ will satisfy an accuracy of about ε_ℓ^2 :

$$\|X - F_1\| = \|X - F_0 - F_E\| = \|E - F_E\| \le \varepsilon_\ell \|E\| \le \varepsilon_\ell \|X - F_0\| \le \varepsilon_\ell^2 \|X\|. \tag{2.4}$$

This idea is therefore based on computing additive updates $F_0 + F_E$ in mixed precision. This is reminiscent of the IR for linear systems $Ax = b$, which is also based on additive updates of the solution x. Indeed, for $Ax = b$, step 1 corresponds to computing an initial solution x_0, step 2 corresponds to computing the residual $r = b - Ax_0$, step 3 corresponds to solving another linear system $Ad = r$ for a correction term d, and step 4 updates $x_1 = x_0 + d$. The term “iterative refinement” is most commonly used for the solution of linear systems, but is also found in the least squares, singular value, or eigenvalue problems. Therefore, we believe that the method proposed in this article should be called “iterative refinement for LRA”.

One of the crucial insights that makes this method effective is that the error E has a low numerical rank whenever X has one too, since it is the sum of X and F_0, which is low-rank by construction. To be specific, we will prove in Section 2.2.2 that $\texttt{rank}(E, \varepsilon_\ell)$ is at most $2\,\texttt{rank}(X, \varepsilon_\ell^2)$ and, for the same reason, the rank of the refined factors F_1 is bounded by $3\,\texttt{rank}(X, \varepsilon_\ell^2)$. The factors F_1 can then be cheaply recompressed into factors of optimal rank $\texttt{rank}(X, \varepsilon_\ell^2)$ in high precision u by exploiting their (suboptimal) low-rank structure.

If accuracy higher than ε_ℓ^2 is needed, we can apply the method again on F_1 to obtain an accuracy of ε_ℓ^3, and so on; we obtain Algorithm 2.1, which repeats this process until the desired accuracy is achieved. An iteration of the algorithm is illustrated in Figure 2.1. Algorithm 2.1 is described within a general IR framework that uses an arbitrary LRA algorithm : $\texttt{lra}(X, \varepsilon)$ returns low-rank factors F of X at accuracy ε. We also require a `decompress` kernel which

transforms low-rank factors F back to a full-size object, and a `recompress` kernel which takes low-rank factors F and ε as input, and computes their optimal LRA at accuracy ε (that is, the LRA of the smallest rank that achieves an accuracy of at least ε). We consider these three abstract kernels in order for algorithm 2.1 to be as general as possible. In particular, this will allow us to apply it to different LRA methods, both for matrices and tensors. However, to be concrete, we will also discuss specific examples in sections 2.3 and 2.4. Namely, we will consider five different `lra` kernels : QRCP and randomized SVD for matrices, and HOSVD, TTSVD, and HTSVD for tensors. For matrices, the `decompress` kernel is simply a matrix-matrix product; for tensors, it is similarly a contraction of the low-rank tensor along all inner dimensions. Finally, the `recompress` kernel depends on the `lra` kernel : for each case, we will show how to adapt the `lra` kernel (which takes as input a full object) to obtain the `recompress` kernel (which takes as input a low-rank object); see sections 2.3 and 2.4 for more details.

Algorithm 2.1 Iterative refinement for LRA.

Input : X, the matrix or tensor of interest;
ε, the desired relative accuracy for the approximation;
n_{it}, the maximum number of iterations;
`lra`, a low-rank approximation algorithm;
`decompress`, `recompress` : decompression and recompression algorithms.

Output : F, the low-rank factors X.

1 : Compute $F = \texttt{lra}(X, \varepsilon_\ell)$ in precision u_ℓ.
2 : **for** $i = 1$ **to** n_{it} **do**
3 : Compute $E = X - \texttt{decompress}(F)$ in precision u.
4 : Compute $\alpha = \|E\|$ in precision u.
5 : If $\alpha \leq \varepsilon\|X\|$, exit.
6 : Scale $E \leftarrow \alpha^{-1}E$ in precision u.
7 : Compute $F_E = \texttt{lra}(E, \varepsilon_\ell)$ in precision u_ℓ.
8 : Scale back $F_E \leftarrow \alpha F_E$ in precision u.
9 : Compute $F = \texttt{recompress}(F + F_E, \varepsilon_\ell^{i+1})$ in precision u.
10 : **end for**

Algorithm 2.1 incorporates two additional steps to make the method effective. First, the error E is scaled by the inverse of its norm (Line 6) before computing its low-rank factors F_E, and then scaled back after (Line 8). This is done to prevent the elements of E from underflowing when converted to the lower precision, in the case where the arithmetic uses not only a reduced number of bits in the significand but also in the exponent, such as is the case for the Institute of Electrical and Electronics Engineers (IEEE) half pre-

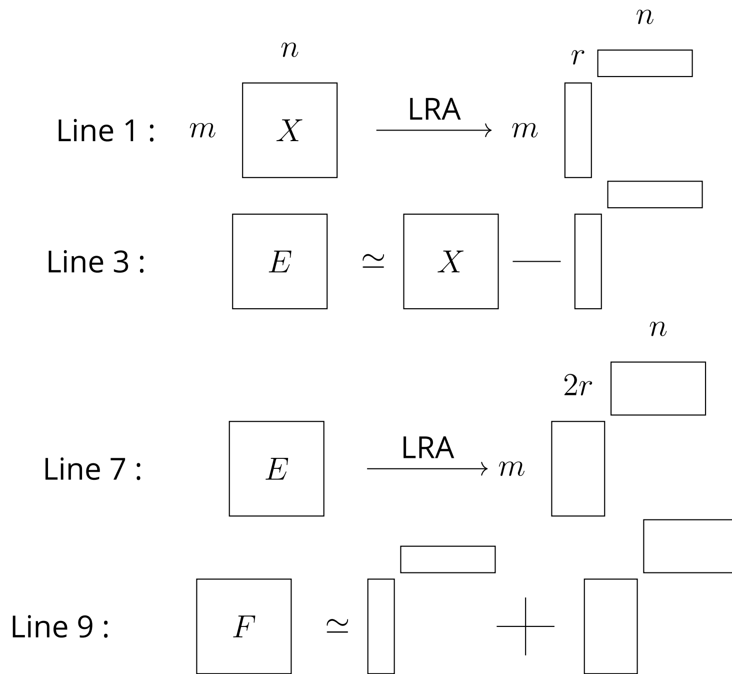


Figure 2.1 – Illustration of the LRA algorithm.

cision fp16 arithmetic. We also note that the elements of X itself are also susceptible to underflow or overflow; in this case, a similar scaling should be applied to X. Second, the factors F are recompressed (Line 9) to their optimal low-rank representation to prevent the rank from growing out of control during the iterations. Indeed, as mentioned above, the rank of F is bounded by $3\,\texttt{rank}(X, \varepsilon_\ell^2)$ after one iteration. Therefore, in the absence of recompression, the rank would grow as $3^k\,\texttt{rank}(X, \varepsilon_\ell^2)$ after k iterations, and quickly make the method unaffordable. This `recompress` kernel can be implemented for an asymptotically negligible cost for many LRA algorithms, including all those considered in this article—has done in Chapter 4.

Algorithm 2.1 has four parameters that control its accuracy : ε, ε_ℓ, u, and u_ℓ.

- ε indicates the target accuracy for the final factors, and is prescribed by the user as input to Algorithm 2.1.
- u is the unit roundoff of the high precision, which should be taken to be the lowest possible such that u is still safely smaller than ε : $u = \theta\varepsilon$, $\theta \le 1$.
- u_ℓ is the unit roundoff of the low precision, which is used to perform the most expensive parts of the computation, the calls to `lra`. Its choice depends on both the available arithmetics on the target hardware, and the

target accuracy ε. Indeed, lowering the precision makes each iteration faster but requires more of them.

- Finally, ε_ℓ is the tolerance used by the `lra` kernel; since `lra` is performed in precision u_ℓ, ε_ℓ should be set such that u_ℓ is safely smaller than ε_ℓ, that is, $\varepsilon_\ell = u_\ell/\theta_\ell$, $\theta_\ell \le 1$. We note that this is necessary because using a ε_ℓ too close to u_ℓ prevents most `lra` algorithms from reliably detecting the correct rank, due to the noise introduced by rounding errors.

In the rest of this section, we first perform an error analysis to determine the attainable accuracy and convergence rate of Algorithm 2.1. We then perform a complexity analysis to determine under which conditions the algorithm can be expected to be faster than a standard uniform precision `lra` performed entirely in high precision u.

2.2.1 . Error analysis

In order to carry out the error analysis of Algorithm 2.1, we will use the standard model of floating-point arithmetic defined in Section 1.4.1 [26, sect. 2.2]. In addition, we also need to make the following three assumptions on the numerical behavior of the `lra`, `decompress`, and `recompress` kernels. First, we assume that computing $F = \texttt{lra}(X, \varepsilon_\ell)$ in precision u_ℓ yields computed factors $\widehat{F}$ satisfying

$$\|X - \widehat{F}\| \le (\varepsilon_\ell + b_1 u_\ell)\|X\| = (1 + b_1\theta_\ell)\varepsilon_\ell\|X\|. \tag{2.5}$$

Second, we assume that computing $F = \texttt{decompress}(F)$ in precision u yields computed factors $\widehat{F}$ satisfying

$$\|F - \widehat{F}\| \le b_2 u\|F\| = b_2\theta\varepsilon\|F\|. \tag{2.6}$$

Third, we assume that computing $F = \texttt{recompress}(F, \varepsilon)$ in precision u yields computed factors $\widehat{F}$ satisfying

$$\|F - \widehat{F}\| \le (\varepsilon + b_3 u)\|F\| = (1 + b_3\theta)\varepsilon\|F\|. \tag{2.7}$$

In (2.5)–(2.7), the error bounds are parameterized by the constants b_1, b_2, and b_3, which will usually depend on the dimensions of X.

For simplicity, we ignore any rounding errors associated with the scaling by $\alpha = \|E\|$, which are negligible and can in fact be prevented by rounding α to the nearest power of two (in binary floating-point arithmetic).

We are now ready to prove the following result.

Theorem 2.1. *If Algorithm 2.1 is applied to X with `lra`, `decompress`, and `recompress` kernels satisfying Equation* (2.5)–*Equation* (2.7)*, then after k iterations, the computed factors $\widehat{F}$ satisfy*

$$\|X - \widehat{F}\| \le (\phi^{k+1} + \xi + O(\varepsilon_\ell\varepsilon))\|X\|, \tag{2.8}$$

with $\phi = (1 + b_1\theta_\ell)\varepsilon_\ell + (b_2 + 1)\theta\varepsilon$ and $\xi = \big(1 + (b_2 + b_3 + 2)\theta\big)\varepsilon$.

Proof. Defining $\widehat{F}_i$ as the computed factors after i iterations, our goal is to obtain a bound of the form

$$\|X - \widehat{F}_{i+1}\| \le \phi\|X - \widehat{F}_i\| + \xi\|X\|, \quad \phi < 1, \tag{2.9}$$

which will allow us to conclude that the error contracts by a factor ϕ at each iteration, until it converges to its maximum attainable accuracy ξ.

Given $\widehat{F}_i$, the first step is to decompress it and compute E at Line 3; by Equation (2.6), the computed $\widehat{E}$ satisfies

$$\|X - \widehat{F}_i - \widehat{E}\| \le \theta\varepsilon\big((b_2+1)\|\widehat{F}_i\| + \|X\|\big), \tag{2.10}$$

with an extra term $\theta\varepsilon(\|\widehat{F}_i\| + \|X\|)$ on the right-hand side coming from the rounding error incurred by the subtraction. Using the triangle inequality $\|\widehat{F}_i\| \le \|X - \widehat{F}_i\| + \|X\|$, we can rearrange Equation (2.10) as

$$\|X - \widehat{F}_i - \widehat{E}\| \le \theta\varepsilon\big((b_2+1)\|X - \widehat{F}_i\| + (b_2+2)\|X\|\big). \tag{2.11}$$

Then we compute $\texttt{lra}(\widehat{E}, \varepsilon_\ell)$ at Line 7; by Equation (2.5) the computed $\widehat{F}_E$ satisfies

$$\|\widehat{E} - \widehat{F}_E\| \le (1 + b_1\theta_\ell)\varepsilon_\ell\|\widehat{E}\|. \tag{2.12}$$

By using Equation (2.11) and the triangle inequality, we rearrange Equation (2.12) as

$$\begin{aligned} \|\widehat{E} - \widehat{F}_E\| &\le (1 + b_1\theta_\ell)\varepsilon_\ell\big(\|X - \widehat{F}_i\| + \|X - \widehat{F}_i - \widehat{E}\|\big) &\quad (2.13)\\ &= (1 + b_1\theta_\ell)\varepsilon_\ell\|X - \widehat{F}_i\| + O(\varepsilon_\ell\varepsilon), &\quad (2.14) \end{aligned}$$

where we do not keep track explicitly of high order terms in $O(\varepsilon_\ell\varepsilon)$ for the sake of readability. Finally, we obtain the next iterate F_{i+1} by recompressing $\widehat{F}_i + \widehat{F}_E$ at Line 9; by Equation (2.7), the computed $\widehat{F}_{i+1}$ satisfies

$$\|\widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| \le (1 + b_3\theta)\varepsilon\|\widehat{F}_i + \widehat{F}_E\|. \tag{2.15}$$

By using Equation (2.11), Equation (2.14), and the triangle inequality, we rearrange Equation (2.15) as

$$\begin{aligned} \|\widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| &\le (1 + b_3\theta)\varepsilon\big(\|X - \widehat{F}_i - \widehat{E}\| + \|\widehat{E} - \widehat{F}_E\| + \|X\|\big) &\quad (2.16)\\ &= (1 + b_3\theta)\varepsilon\|X\| + O(\varepsilon_\ell\varepsilon). &\quad (2.17) \end{aligned}$$

Using the triangle inequality together with Equation (2.11), Equation (2.14), and Equation (2.17), we obtain

$$\begin{aligned} \|X - \widehat{F}_{i+1}\| &= \|X - \widehat{F}_i - \widehat{E} + \widehat{E} - \widehat{F}_E + \widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| &\quad (2.18)\\ &\le \|X - \widehat{F}_i - \widehat{E}\| + \|\widehat{E} - \widehat{F}_E\| + \|\widehat{F}_i + \widehat{F}_E - \widehat{F}_{i+1}\| &\quad (2.19)\\ &\le \phi\|X - \widehat{F}_i\| + \xi\|X\| + O(\varepsilon_\ell\varepsilon), &\quad (2.20) \end{aligned}$$

with

$$\phi = (1 + b_1\theta_\ell)\varepsilon_\ell + (b_2 + 1)\theta\varepsilon \tag{2.21}$$

and

$$\xi = \big(1 + (b_2 + b_3 + 2)\theta\big)\varepsilon. \tag{2.22}$$

Noting that the first factors $\widehat{F}_0$ computed at Line 1 satisfy by Equation (2.5)

$$\|X - \widehat{F}_0\| \leq (1 + b_1\theta_\ell)\varepsilon_\ell\|X\| \leq \phi\|X\| \tag{2.23}$$

concludes the proof. □

Theorem 2.1 states that the approximation error $\|X - \widehat{F}\|$ contracts by a factor $\phi = O(\varepsilon_\ell)$ at each iteration, until it reaches its maximum attainable accuracy $\xi = O(\varepsilon)$. Thus, after k iterations, the error is of order $\varepsilon_\ell^{k+1} + \varepsilon$, which means that we can actually estimate in advance approximately how many iterations are needed to achieve the desired accuracy ε (up to constants) :

$$n_{\text{it}} = \log(\varepsilon)/\log(\varepsilon_\ell) - 1. \tag{2.24}$$

It is worth noting that, unlike IR for linear systems, there is no dependence on the condition number of X and thus the only condition for Algorithm 2.1 to converge is that $\phi < 1$, which should always be the case as long as θ_ℓ is sufficiently small. This is explained by the fact that the speed of convergence of Algorithm 2.1 depends on the backward error $\|A - XY^T\|$, rather than the forward one. Therefore, Algorithm 2.1 is extremely general : it can be applied to any matrix or tensor of low numerical rank, and it can make use of potentially very low precisions.

From a numerical perspective, Algorithm 2.1 is therefore quite appealing. We next discuss under which conditions it is also attractive from a computational perspective.

2.2.2 . Complexity analysis

The cost of Algorithm 2.1 will mainly depend on two factors : the relative speed for computing in different precisions on the target hardware, and the numerical ranks of the objects encountered during the iterations (X,E, and F).

We begin by bounding these ranks solely as a function of the numerical ranks of X at given accuracies. In order to do so, we will need to bound the numerical rank of $A+B$ as a function of that of A and B by using the following lemma. In what follows, recall that when X is a tensor, $\texttt{rank}(X, \varepsilon)$ is a vector (see Section 2.1.1) and all the (in)equalities involving this quantity should be interpreted component wise.

Lemma 2.1.

$$\textit{rank}(A + B, \varepsilon) \leq \textit{rank}\Big(A, \frac{\varepsilon\|A + B\|}{2\|A\|}\Big) + \textit{rank}\Big(B, \frac{\varepsilon\|A + B\|}{2\|B\|}\Big). \tag{2.25}$$

Proof. For the purpose of this proof only, we introduce an alternative definition of numerical rank that measures accuracy in an absolute sense, rather than a relative one. Let $\texttt{rankabs}(X, \varepsilon)$ be the smallest integer r_ε such that there exists F of rank r_ε satisfying $\|X - F\| \leq \varepsilon$. The $\texttt{rankabs}$ operator satisfies

$$\texttt{rankabs}(X, \varepsilon\|X\|) = \texttt{rank}(X, \varepsilon) \tag{2.26}$$

and

$$\texttt{rankabs}(A + B, \varepsilon) \leq \texttt{rankabs}(A, \varepsilon/2) + \texttt{rankabs}(B, \varepsilon/2) \tag{2.27}$$

by the triangle inequality. Therefore, we have that

$$\begin{align}
\texttt{rank}(A + B, \varepsilon) &= \texttt{rankabs}(A + B, \varepsilon\|A + B\|) \tag{2.28}\\
&\leq \texttt{rankabs}(A, \varepsilon\|A + B\|/2) + \texttt{rankabs}(B, \varepsilon\|A + B\|/2) \tag{2.29}\\
&= \texttt{rank}(A, \varepsilon\|A + B\|/2\|A\|) + \texttt{rank}(B, \varepsilon\|A + B\|/2\|B\|). \tag{2.30}
\end{align}$$

□

Let us consider the ith iteration of Algorithm 2.1. Using Equation (2.25), we can bound the rank of $E = X - F$ as

$$\texttt{rank}(E, \varepsilon_\ell) \leq \texttt{rank}(X, \varepsilon_\ell\|E\|/2\|X\|) + \texttt{rank}(F). \tag{2.31}$$

Since F is the low-rank factorization of X after i iterations, it achieves an accuracy of ε_ℓ^i and so

$$\texttt{rank}(F) = \texttt{rank}(X, \varepsilon_\ell^i) \tag{2.32}$$

(we assume here that $\varepsilon_\ell^i \leq \varepsilon$ as otherwise the method would be stopped). After i iterations, $\|E\| \leq \varepsilon_\ell^i\|X\|$ and so overall Equation (2.31) becomes

$$\texttt{rank}(E, \varepsilon_\ell) \leq \texttt{rank}(X, \varepsilon_\ell^{i+1}/2) + \texttt{rank}(X, \varepsilon_\ell^i). \tag{2.33}$$

Since $\texttt{rank}(X, \varepsilon') \leq \texttt{rank}(X, \varepsilon)$ for $\varepsilon \leq \varepsilon'$, assuming that $\varepsilon \leq \varepsilon_\ell^{i+1}/2$, we can obtain a simpler but weaker version of Equation (2.33),

$$\texttt{rank}(E, \varepsilon_\ell) \leq 2\,\texttt{rank}(X, \varepsilon). \tag{2.34}$$

Thus, the rank of E at any iteration is at most twice as large as the numerical rank of X at accuracy ε.

With a similar argument, the rank of the refined factors $F + F_E$ is bounded by

$$\begin{align}
\texttt{rank}(F + F_E) &\leq \texttt{rank}(F) + \texttt{rank}(F_E) \tag{2.35}\\
&= \texttt{rank}(F) + \texttt{rank}(E, \varepsilon_\ell) \tag{2.36}\\
&\leq \texttt{rank}(X, \varepsilon_\ell^{i+1}/2) + 2\,\texttt{rank}(X, \varepsilon_\ell^i), \tag{2.37}
\end{align}$$

where Equation (2.37) follows from Equation (2.32) and Equation (2.33). Again, a simpler but weaker bound is

$$\mathtt{rank}(F + F_E) \leq 3\,\mathtt{rank}(X, \varepsilon), \tag{2.38}$$

showing that at any iteration the rank of the factors before recompression is at most three times the numerical rank of X at accuracy ε.

Now that we have bounded the ranks of the objects appearing in Algorithm 2.1, we are ready to analyze its cost. We denote as p and s the product and the sum of the dimensions of X, respectively. Thus, if $X \in \mathbb{R}^{m\times n}$ is a matrix, $p = mn$ and $s = m + n$; if $X \in \mathbb{R}^{n_1\times\cdots\times n_d}$ is a tensor of order d, $p = \prod n_i$ and $s = \sum n_i$. For readability, we only report the dominant term for the cost of each assumption/line, and assume a large scale setting, so that $p \gg s$.

We make the following assumptions on the flops (floating-point operations) cost of the different kernels :

$$\text{Flops } \mathtt{lra}(X, \varepsilon) = c_1 p\,\mathtt{rank}(X, \varepsilon)_1 + o(p), \tag{2.39a}$$

$$\text{Flops } \mathtt{decompress}(F) = c_2 p\,\mathtt{rank}(F)_1 + o(p), \tag{2.39b}$$

$$\text{Flops } \mathtt{recompress}(F, \varepsilon) = o(p). \tag{2.39c}$$

For tensors, the cost of `lra` and `decompress` depends on the order in which the dimensions are treated (see Section 2.4). Here we assume they are treated in the natural order, without loss of generality since the dimensions can be arbitrarily reordered. In this case, the dominant cost of `lra` and `decompress` is proportional to the first coefficient of the rank vectors, denoted as $\mathtt{rank}(X, \varepsilon)_1$. Note that for matrices, the rank is a scalar so that $\mathtt{rank}(X, \varepsilon)_1 = \mathtt{rank}(X, \varepsilon)$.

Let us now measure the flops cost of the ith iteration of Algorithm 2.1.

- Line 3 : $c_2 p\,\mathtt{rank}(X, \varepsilon_\ell^i)_1$ flops.
- Line 4 : $2p$ flops (for the Frobenius norm).
- Line 6 : p flops.
- Line 7 : $c_1 p\big(\mathtt{rank}(X, \varepsilon_\ell^{i+1}/2)_1 + \mathtt{rank}(X, \varepsilon_\ell^i)_1\big)$ flops.
- Line 8 : $o(p)$ flops.
- Line 9 : $o(p)$ flops.

The dominant steps are the calls to `lra` (Line 7), which is performed in low precision, and to `decompress` (Line 3), which is performed in high precision. The computation of α and the scaling by α (Lines 4 and 6) could also be significant if the ranks are very small.

Summing the costs of these dominant steps across all n_{it} iterations, plus the cost of the initial `lra` (Line 1), gives a total flops cost of

$$\begin{aligned} \text{Flops IR } &= c_1 p\,\texttt{rank}(X,\varepsilon_\ell)_1 + p\sum_{i=1}^{n_{\text{it}}}\Big(c_1\,\texttt{rank}(X,\varepsilon_\ell^{i+1}/2)_1 \\ &\quad + c_1\,\texttt{rank}(X,\varepsilon_\ell^{i})_1 + c_2\,\texttt{rank}(X,\varepsilon_\ell^{i})_1 + 3\Big). \end{aligned} \tag{2.40}$$

Since some of these flops are performed in low precision and some in high precision, we must account for the different speeds of different arithmetics. To do so, we ponderate the low precision flops by a weight $\omega_\ell < 1$ which indicates the relative performance ratio between the low and the high precision. We obtain

$$\begin{aligned} \text{Time IR } &\propto \omega_\ell c_1 p\,\texttt{rank}(X,\varepsilon_\ell)_1 + p\sum_{i=1}^{n_{\text{it}}}\Big(\omega_\ell c_1\,\texttt{rank}(X,\varepsilon_\ell^{i+1}/2)_1 \\ &\quad + \omega_\ell c_1\,\texttt{rank}(X,\varepsilon_\ell^{i})_1 + c_2\,\texttt{rank}(X,\varepsilon_\ell^{i})_1 + 3\Big), \end{aligned} \tag{2.41}$$

where the "Time" formula should be taken as a rough estimator of the time performance of the algorithm, although in practice the actual execution time depends on a number of other complex factors such as the arithmetic intensity of the operations, the data locality, and the parallelism.

This cost is to be compared with the cost of simply computing $\texttt{lra}(X,\varepsilon)$ in the high precision u, given by

$$\text{Time Ref. } \propto c_1 p\,\texttt{rank}(X,\varepsilon)_1. \tag{2.42}$$

This complexity analysis reveals two situations where Algorithm 2.1 can outperform the uniform precision approach.

Numerical ranks $\texttt{rank}(X,\varepsilon_\ell^{i})$ rapidly decreasing as ε_ℓ increases The first situation is when the numerical ranks of X at accuracy lower than the final target ε are much smaller than $\texttt{rank}(X,\varepsilon)$. This can certainly be the case in some applications. For example, in the case of matrices, the numerical rank of X at any given accuracy is determined by its singular values : $r = \texttt{rank}(X,\varepsilon_\ell)$ is the number of singular values that need to be kept for the truncated SVD to approximate X with accuracy at least ε_ℓ. Thus, if the singular values decay rapidly, $\texttt{rank}(X,\varepsilon_\ell)$ will in general be significantly smaller than $\texttt{rank}(X,\varepsilon)$. The most extreme example is a matrix with one large singular value and all the remaining $\texttt{rank}(X,\varepsilon)-1$ singular values just above $\varepsilon\|X\|$. In this extreme case, all the iterations except the last will only need to compute on rank-1 matrices, so the overall cost of the method will be dominated by that of the last iteration, which is roughly $\omega_\ell c_1 p\,\texttt{rank}(X,\varepsilon)$, and so is always smaller than Equation (2.42). In a more realistic setting where the singular values decay exponentially, we may still expect Equation (2.41) to be smaller than Equation (2.42) even for reasonably traditional values of ω_ℓ.

This situation also extends to tensors, although we do not have such a simple characterization as one based on singular values. Essentially, if the underlying matrices used in the tensor representation exhibit rapidly decaying singular values, then the rank vectors $\mathtt{rank}(X, \varepsilon_\ell)$ will take much smaller values than $\mathtt{rank}(X, \varepsilon)$ when $\varepsilon_\ell \gg \varepsilon$.

Very fast low precision arithmetic (or very slow high precision arithmetic) The second situation is when the low precision arithmetic is much faster than the high precision one, that is, when $\omega_\ell \ll 1$. This is becoming increasingly common for low precisions on modern hardware, especially accelerators. For example, the fp16 and bfloat16 arithmetics can be up to 16 times faster than fp32 arithmetic on recent NVIDIA Graphics Processing Unit (GPU)s. Similarly, fp16 arithmetic can be up to 8 times faster than fp32 on the AMD Instinct MI250X GPUs.

Moreover, such hardware provides another crucial feature which allows the `decompress` kernel to also be performed in low precision. Indeed, while Algorithm 2.1 requires the `decompress` kernel Line 3 to be performed in high precision, this kernel has the particularity of taking as input the factors F, which are stored in low precision. Therefore, what we really need is to compute in high precision with low precision numbers ; this happens to be an easier task than the more general problem of computing in high precision with high precision numbers. In fact, an increasing range of modern hardware provides the capability to perform this task inexpensively. For example, the NVIDIA GPUs are equipped with so-called tensor core units that can carry out matrix multiplication with half precision (16-bit) matrices at the accuracy of single precision (fp32) arithmetic but at the much higher speed of half precision arithmetic. Similar instructions are available on several other architectures, which have been analyzed by Blanchard et al. [17] under a common framework called block Fused Multiply-Add (FMA). In our context, a block FMA unit therefore allows all the dominant operations to be performed at the speed of the low precision, which is much higher than that of the high precision. In this situation, the time cost of Algorithm 2.1 can be much lower than Equation (2.42), even in the worst case scenario where the ranks of all objects throughout the iterations attain their upper bound of $2\,\mathtt{rank}(X, \varepsilon)$ for E. Indeed, in this case, we have

$$\text{Time IR} \propto \omega_\ell p\,\mathtt{rank}(X, \varepsilon)_1 \big(c_1(2n_{\text{it}} + 1) + n_{\text{it}} c_2\big) + 3p. \tag{2.43}$$

Neglecting the $3p$ term, Equation (2.43) is smaller than Equation (2.42) if

$$\omega_\ell(2n_{\text{it}} + 1 + n_{\text{it}} c_2 / c_1) \le 1. \tag{2.44}$$

For many LRA algorithms, including all those considered in this article, the cost of compressing (`lra` kernel) a full object is higher than the cost of decompressing it (`decompress` kernel), that is, $c_1 > c_2$. Therefore, Equation (2.44) is certainly satisfied if $\omega_\ell(3n_{\text{it}} + 1) \le 1$.

To illustrate this cost analysis, let us take the NVIDIA GPU tensor cores as an example, for which fp16 arithmetic ($u_\ell \approx 3 \times 10^{-4}$) is up to 16 times faster ($\omega_\ell = 1/16$) than fp32 arithmetic ($u \approx 6 \times 10^{-8}$). Since $u_\ell^2 \approx u$, a single refinement step ($n_{\text{it}} = 1$) suffices to recover fp32 accuracy. Thus, the term $\omega_\ell(3n_{\text{it}} + 1)$ is equal to $4/16 = 1/4$, which suggests that a speedup of up to a factor $4\times$ could be expected.

Clearly, the two situations discussed above are not exclusive, so in practice it is very possible that we have both smaller ranks and a fast low precision. To further assess under which condition Algorithm 2.1 can outperform the standard approach, we now specialize it to specific matrix or tensor algorithms in Section 2.3 and Section 2.4.

2.3 . Application to matrix low-lank approximation

In this section we specialize Algorithm 2.1 to the case where $X \in \mathbb{R}^{m\times n}$ is a matrix. We focus on two widely popular choices : the truncated QR decomposition with column pivoting (QRCP, Section 2.3.1), and the randomized SVD (Section 2.3.2). For each algorithm, we discuss their use as `lra` kernel in our mixed precision IR approach (Algorithm 2.1) : in particular, we check that the algorithm satisfies the assumptions Equation (2.5)–Equation (2.7) of the error analysis and the assumptions Equation (2.39a)–Equation (2.39c) of the complexity analysis. We also explain how to perform the `recompress` kernel based on the specific `lra` choice.

2.3.1 . Truncated QRCP decomposition

Our first choice of `lra` algorithm is a truncated QRCP decomposition. As explained in Section 1.2.2, QRCP decomposes the original matrix $X \in \mathbb{R}^{m\times n}$ as

$$XP = QR \tag{2.45}$$

where $Q \in \mathbb{R}^{m\times n}$ is a matrix with orthonormal columns, $R \in \mathbb{R}^{n\times n}$ is an upper triangular matrix, and $P \in \mathbb{R}^{n\times n}$ is a permutation matrix.

We can implement a suitable `recompress` algorithm using this truncated QRCP decomposition. Several versions are possible ; we describe in Algorithm 2.2 the one that we will use in this article. Given a (non-optimal) truncated QRCP decomposition $Q_{in}V_{in}^T$, Algorithm 2.2 recompresses it into an optimal LRA as follows. First, a thin QR factorization $\overline{Q}R$ of V_{in} is computed. Then, the product $Q_{in}R^T$ is formed and a truncated QRCP decomposition $Q_{out}\overline{V}^T$ is computed at the desired target accuracy ε. This yields the recompressed left factor Q_{out} ; the recompressed right factor V_{out} is obtained by forming $\overline{Q}\overline{V}$.

We now check that the assumptions Equation (2.5)–Equation (2.7) of the error analysis are satisfied for truncated QRCP and determine the value of the constants in the error bounds. For Equation (2.5), we can analyze the

Algorithm 2.2 `recompress` algorithm using truncated QRCP decomposition.

Input : a truncated QRCP decomposition $Q_{in}V_{in}^T$ of the form Equation (1.11);
ε, the target accuracy.

Output : a recompressed QRCP decomposition $Q_{out}V_{out}^T$.

1 : Compute the thin QR factorization $\overline{Q}R = V_{in}$.
2 : Compute the truncated QRCP decomposition $Q_{out}\overline{V}^T = Q_{in}R^T$.
3 : $V_{out} \leftarrow \overline{Q}\overline{V}$

truncated QRCP by separating the truncation and rounding errors. As explained above, we can control the size of the truncation error by stopping the QRCP decomposition once the approximation error falls below the desired threshold ε_ℓ, so that in exact arithmetic we obtain

$$Q_k R_k = XP + E, \quad \|E\| \le \varepsilon_\ell \|X\|. \tag{2.46}$$

To account for the rounding errors, we use standard analysis of QR decomposition [26, p. 361], which shows that the computed QR factors satisfy the backward error bound

$$\widehat{Q}_k \widehat{R}_k = XP + \Delta X + E, \quad \|\Delta X\| \le \sqrt{k}\widetilde{\gamma}_{mk}\|X\|, \tag{2.47}$$

where $\widetilde{\gamma}_{mk} = cmku_\ell/(1 - cmku_\ell) \simeq cmku_\ell$, for a modest constant c independent of the dimensions of the problem. Note that this bound is valid even with column pivoting, since computing the QRCP decomposition $XP = QR$ of X is equivalent to computing the unpivoted QR decomposition of XP. Assumption Equation (2.5) is thus satisfied with $b_1 \simeq cmk\sqrt{k}$. The `decompress` kernel is a standard matrix multiplication between the low-rank factors Q and V hence assumption Equation (2.6) is satisfied with $b_2 = k$ by [26, p. 71]. Finally, to bound the error introduced by the `recompress` kernel we must analyze Algorithm 2.2. Since the algorithm simply consists of standard QR, QRCP, and matrix multiplication operations, this analysis is straightforward and relies on standard error bounds from the literature. Although we omit it here for the sake of conciseness, we have performed this analysis and found that Equation (2.7) is satisfied with $b_3 \simeq cn(k^{3/2} + \widetilde{k}^{3/2}) + k$, where k is the rank of $Q_{in}V_{in}^T$ before recompression and $\widetilde{k}$ is the rank of $Q_{out}V_{out}^T$ after recompression.

Finally, we discuss the cost of Algorithm 2.1 when using truncated QRCP decomposition. The truncated QRCP decomposition Equation (1.11) can be computed in $4mnk + o(mnk)$ flops [24], so that assumption Equation (2.39a) is satisfied with $c_1 = 4$. The `decompress` kernel is a matrix multiplication which requires $2mnk$ flops, so that assumption Equation (2.39b) is satisfied with $c_2 =$

2. For the `recompress` kernel, Algorithm 2.2 requires $(6m+2n)k^2 + o(mk^2 + nk^2)$ flops, so that assumption Equation (2.39c) is satisfied since $(m+n)k^2 = o(mn)$.

2.3.2 . Randomized SVD decomposition

Our second choice of LRA algorithm is the randomized SVD decomposition. Several variants have been proposed; in this chapter we use the one described in Algorithm 1.3 in Section 1.2.3, which was proposed by Martinsson and Voronin [31]. For the sake of commodity, we recall the algorithm again in Algorithm 2.3.

Algorithm 2.3 Randomized SVD decomposition.

Input : $X \in \mathbb{R}^{m\times n}$, a target accuracy ε_ℓ, and a block size b.
Output : a truncated SVD $U\Sigma V^T$ decomposition of X.

1 : Initialize Q and B to empty matrices.
2 : $n_X = \|X\|$
3 : **repeat**
4 : Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times b}$.
5 : $Y = X\Omega$
6 : $Q_b = \mathtt{qr}(Y - Q(Q^T Y))$
7 : $B_b = Q_b^T X$
8 : $Q \leftarrow [Q\ Q_b]$
9 : $B \leftarrow \begin{bmatrix} B \\ B_b \end{bmatrix}$
10 : $X \leftarrow X - Q_b B_b$
11 : **until** $\|X\| \leq \varepsilon_\ell n_X$
12 : Compute the truncated SVD decomposition $B \approx \overline{U}\Sigma V^T$ at accuracy ε_ℓ.
13 : $U = Q\overline{U}$

We chose to use this specific randomized SVD variant because it presents several advantages. It is blocked, which allows for an efficient implementation on modern hardware. Moreover, it provides a cheap yet reliable error estimation. Thanks to blocking and error estimation, Algorithm 2.3 can adaptively reveal the numerical rank of the matrix at the requested accuracy ε_ℓ; no a priori knowledge on the rank is thus necessary. Finally, we have experimentally observed Algorithm 2.3 to be more accurate than other randomized SVD variants that we have tested.

In order to perform the `recompress` operation using randomized SVD, Algorithm 2.3 needs to be slightly adapted. The algorithm takes as input a (non-optimal) low-rank matrix $X = U\Sigma V^T$ and seeks to recompress it, without forming the full X. Lines 5 and 7 of Algorithm 2.3 are matrix products and

can thus exploit the low-rank structure of X. Line 10 is a subtraction between two low-rank matrices, which requires no operations (the low-rank factors can simply be concatenated). The only difficulty lies on Line 11, which requires computing $\|X\|$ to control the error and stop the algorithm. In order to compute $\|X\|$ without forming X, we orthonormalize one of the two low-rank factors and compute the norm of the other one.

We now check if the assumptions Equation (2.5)–Equation (2.7) of the error analysis are satisfied for the randomized SVD. To do so, we rely on the error analysis of Connolly, Higham, and Pranesh [41], which determines error bounds for Algorithm 2.3 in floating-point arithmetic. Note that their analysis assumes that the deterministic SVD on Line 12 is computed with a numerically stable algorithm. By [41, Corollary 2.4], assumption Equation (2.5) is satisfied with $b_1 \simeq \sqrt{n}mk$. Assumption Equation (2.6) is satisfied with $b_2 = k + 1$ since decompressing $U\Sigma V^T$ can be achieved with a matrix-matrix product and a scaling. Finally, the analysis of [41] does not directly apply to the `recompress` version of the algorithm discussed above, but we expect its numerical behavior to be similar to the original algorithm.

Next, we discuss the cost of Algorithm 2.3. If X is a full $m \times n$ matrix, Algorithm 2.3 costs $6mnk + o(mnk)$ flops [31, Eqn. (25)], so that assumption Equation (2.39a) is satisfied with $c_1 = 6$. It is easy to extend [31, Eqn. (25)] to the case where $X = U\Sigma V^T$ is represented under low-rank form to check that the `recompress` variant of Algorithm 2.3 described above has a cost in $O((m+n)k^2) = o(mn)$ flops, so that assumption Equation (2.39c) is indeed satisfied. Finally, assumption Equation (2.39b) is satisfied with $c_2 = 2$ since the `decompress` kernel is simply a matrix-matrix product.

2.4 . Application to tensor low-rank approximation

In this section, we explore the application of our method to three different tensor decompositions, namely Tucker [33], TT [9], and HT [14]. All three decompositions provide direct methods for the `lra` and `recompress` kernels that can guarantee a prescribed accuracy ε, as well as fast `decompress` routines, rendering them amenable to Algorithm 2.1. Description of the decompositions and their corresponding `lra`, `recompress`, and `decompress` kernels are given in Chapter 1. We will more discuss on an error and complexity analysis with respect to the assumptions Equation (2.5)–Equation (2.7) and Equation (2.39a)–Equation (2.39c).

The HOSVD [33], TTSVD [9], and HTSVD [14] algorithms can be used as `lra` kernels to compute the Tucker, TT, and HT decompositions of a given full tensor X, respectively. All three are direct methods that can achieve any prescribed accuracy ε in exact arithmetic. To the best of our knowledge, the effect of rounding errors on these methods in finite precision arithmetic has not

been analyzed in the literature. While these methods are therefore not known to be stable, empirical experiments suggest that they still reliably achieve an accuracy of order ε when run in a finite precision with a unit roundoff u sufficiently smaller than ε. Formally proving this result is a further contribution done in Chapter 4 for a specific framework that encompasses rounding methods for both decompositions.

Algorithm 2.4 `decompress` algorithm for a tensor decomposition.

Input : a low-rank tensor decomposition $\mathcal{F}_X$.
Output : the full tensor $\mathcal{X}$ corresponding to $\mathcal{F}_X$.

1 : $\mathcal{X} \leftarrow \mathcal{F}_X[1]$
2 : **for** $i = \bar{d}$ **to** 1 in reverse order **do**
3 : $\quad \mathcal{X} \leftarrow \mathcal{X} \times_{\{r_i\}} \mathcal{F}_X[i+1]$
4 : **end for**

As the `lra` kernel isolates the cores through a series of $\bar{d}$ SVDs on $\mathcal{X}$, the `decompress` kernel performs successive tensor contractions on neighboring cores of $\mathcal{F}_X$ through the inner dimensions that link them to obtain the full tensor $\mathcal{X}$. In Algorithm 2.4, we provide a generic `decompress` kernel that performs these contractions in the reverse order of SVDs in Algorithm 1.4 for simplicity. This is done on Line 3 where $\times_{\{r_i\}}$ represents the contraction/multiplication across the inner dimension of size r_i. In practice, these contractions could be performed in an arbitrary order among the $\bar{d}$ inner dimensions as tensor contraction is associative. But in terms of performance, the order may have an impact on the time computation for some formats, e.g. TT formats.

Finally, for the TT format we use the standard `recompress` kernel (called "rounding" in the tensor literature) which relies on a sequence of QRs and SVDs on the matricizations of the cores (we refer to [9] for a detailed description). For HT and Tucker decompositions, we employ an adaptation of this TT `recompress` kernel to these formats, instead of the standard `recompress` kernel proposed in [14] mainly to avoid the Gram matrix formation therein.

2.4.1 . Discussion of the error and complexity

For the error analysis, as previously mentioned, tensor computations are experimentally observed to behave stably in finite precision arithmetic, but a formal proof of this fact is not known. The `lra` kernel is based on a chain of SVDs, the `decompress` kernel is based on a chain of matrix multiplies, and the `recompress` kernel on a chain of QRs, SVDs, and matrix multiplies. Despite the fact that all of these basic linear algebra building blocks have stable implementations, it is not sufficient to directly conclude on the stability of the overall tensor computation, because the composition of stable algorithms is not necessarily stable [48]. Therefore, the stability of tensor algorithms is an open problem that deserves a dedicated study.

For the complexity analysis of the `lra` step, we assume that the first SVD reduces the size of the tensor significantly, that is, to $o(p)$, which renders the cost of subsequent SVDs negligible. In this case, we can use the same constant $c_1 = 6$ as in the case of randomized SVD in Section 2.3.2. With the same assumption, the cost of `decompress` in Algorithm 2.4 will also be dominated by the last contraction, which is essentially a matrix multiplication on permuted tensors across the first inner dimension. Thus, we can similarly use the constant $c_2 = 2$ as in Section 2.3.2, since the cost of the previous contractions is in $o(p)$ in this case as well. Even without this assumption, we can find constants $c_1 = 6t$ and $c_2 = 2t$ with the same $t > 1$, since intermediate tensor sizes in each SVD and contraction steps in Algorithm 1.4 and Algorithm 2.4 stay the same. This keeps the cost ratio of these two steps constant across all $\bar{d}$ dimensions yielding the same t for c_1 and c_2; we skip further details for brevity. Finally, the cost of `recompress` remains in $o(p)$ as it involves QRs and SVDs on the matricizations of 2nd or 3rd order cores in the decomposition, whose size is in $o(p)$ by the low-rank assumption. Therefore, we satisfy the assumptions Equation (2.39a)–Equation (2.39c) of the complexity analysis in the tensor case.

2.5 . Numerical experiments

2.5.1 . Experimental setting

We now test our IR approach experimentally. We developed a MATLAB code that implements Algorithm 2.1 and can use as `lra` kernel any of the matrix and tensor LRA algorithms discussed in the previous two sections : QRCP, randomized SVD, HOSVD, TTSVD, and HTSVD.

For the matrix algorithms (QRCP and randomized SVD), we use our own implementation. For the tensor algorithms (HOSVD, TTSVD, and HTSVD), we rely on the implementations from the libraries described in [49], [50], and [51], respectively, with some adjustments. We use MATLAB version R2019a throughout the experiments.

In the experiments, the high precision u is set to double precision (fp64 arithmetic, with unit roundoff $u = 2^{-53} \approx 1 \times 10^{-16}$) and the use of various low precisions u_ℓ is simulated with the chop library of Higham and Pranesh [38] : single precision (fp32 arithmetic, with unit roundoff $u_\ell = 2^{-24} \approx 6 \times 10^{-8}$) and half precision (fp16 and bfloat16 arithmetics, with unit roundoff $u_\ell = 2^{-11} \approx 5 \times 10^{-4}$ and $u_\ell = 2^{-8} \approx 4 \times 10^{-3}$, respectively).

For the low-rank truncation thresholds, we set $\varepsilon = 10^{-13}$ and $\varepsilon_\ell = \theta_\ell u_\ell$, where $\theta_\ell \leq 1$ is a scaling factor necessary to control the effect of rounding errors on the ability of the LRA to detect the correct numerical rank. We analyze in detail the role of this parameter θ_ℓ in Section 2.5.3; based on the conclusions of this analysis, we have set $\theta_\ell = 2^{-1}$ for all LRA kernels except ran-

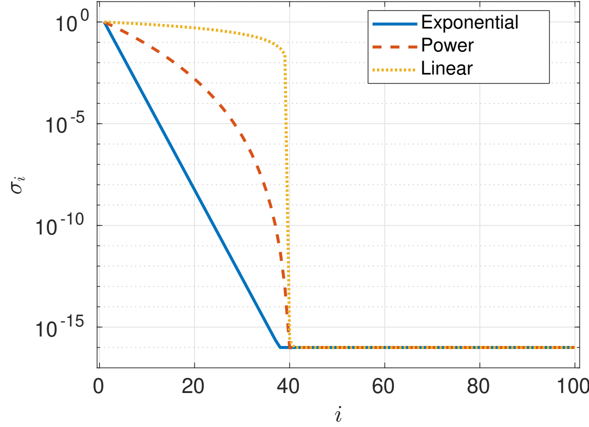


Figure 2.2 – Three types of singular value distributions used in the experiments.

domized SVD and QRCP, for which we have used $\theta_\ell = 2^{-2}$ and $\theta_\ell = 2^{-3}$, respectively.

To test the algorithms, we use randomly generated matrices and tensors with various singular value distributions. More precisely, we compare three types of distributions for the singular values σ_i :

$$\sigma_i = \max(\widehat{\sigma}_i, 10^{-16}), \quad \widehat{\sigma}_i = \begin{cases} 1/i & \text{(linear)} \\ i^{-10} & \text{(power)} \\ e^{-i} & \text{(exponential)} \end{cases} . \tag{2.48}$$

These three distributions are illustrated in Figure 2.2. All random matrices are square and of size 100×100. All random tensors are of fourth order ($d = 4$) and of size $100 \times 100 \times 100 \times 100$. We generate the matrices as $Q_1 \Sigma Q_2$ where Q_1, Q_2 are random orthogonal matrices and Σ is a diagonal matrix with the specified singular values as coefficients. For the tensor experiments involving Tucker and TT decompositions, we generate the low-rank tensor in the Tucker format with $Q^{(1)}$, ..., $Q^{(d)}$ random orthonormal matrices and a dth-order core tensor G whose coefficient $G_{i_1,\ldots i_d}$ is given by $\sigma_{\max(i_1,\ldots,i_d)}$ in Equation (2.48). For the experiments involving the HT format, we generate the low-rank tensor in the HT format whose leaf nodes are matrices with orthogonal columns, and each element $G^{(t)}_{i,j,k}$ of its internal cores is set to $\sigma_{\max(i,j,k)}$.

In addition to these random data sets, we also report results for data sets coming from two real-life applications in Section 2.5.4.

2.5.2 . Experimental results

In the first experiment, we analyze the behavior of Algorithm 2.1 for the matrix case, which is provided in Figure 2.3. We consider three types of matrices based on their singular value distribution (linear, power, and exponen-

tial) and two `lra` kernels (QRCP and randomized SVD). In each case, we plot the relative error

$$\eta_i = \frac{\|X - F_i\|}{\|X\|} \tag{2.49}$$

where F_i is the computed low-rank factor of X after i refinement steps (F_0 is thus a standard LRA of X in precision u_ℓ). The number next to each marker indicates the rank of F_i after recompression.

Figure 2.3 shows that for all three matrices and for both QRCP and randomized SVD, Algorithm 2.1 behaves as expected : the error η_i is roughly equal to $\varepsilon_\ell^{i+1} = (\theta u_\ell)^{i+1}$ after i refinement steps. Thus, using single precision as the low precision u_ℓ, we can achieve an accuracy close to double precision with only one refinement step. Naturally, since for QRCP $\theta = 2^{-3}$ is relatively small, $(u_\ell/\theta)^2$ is noticeably larger than the accuracy achieved by a standard double precision LRA; the refined factors are thus not completely as accurate as if computed directly in double precision. This gap can be filled by performing a second refinement step, although this would likely be unnecessary in most practical scenarios.

Similar results are obtained using half precision as the low precision format (fp16 or bfloat16 arithmetic). An error close to the single precision accuracy can be achieved in just one or a few steps. Moreover, we can even reach double precision accuracy if needed, which illustrates an attractive property of algorithm 2.1 : it can reach an accuracy which is only limited by the high precision, but not by the low precision, despite this low precision being used to perform most of its operations.

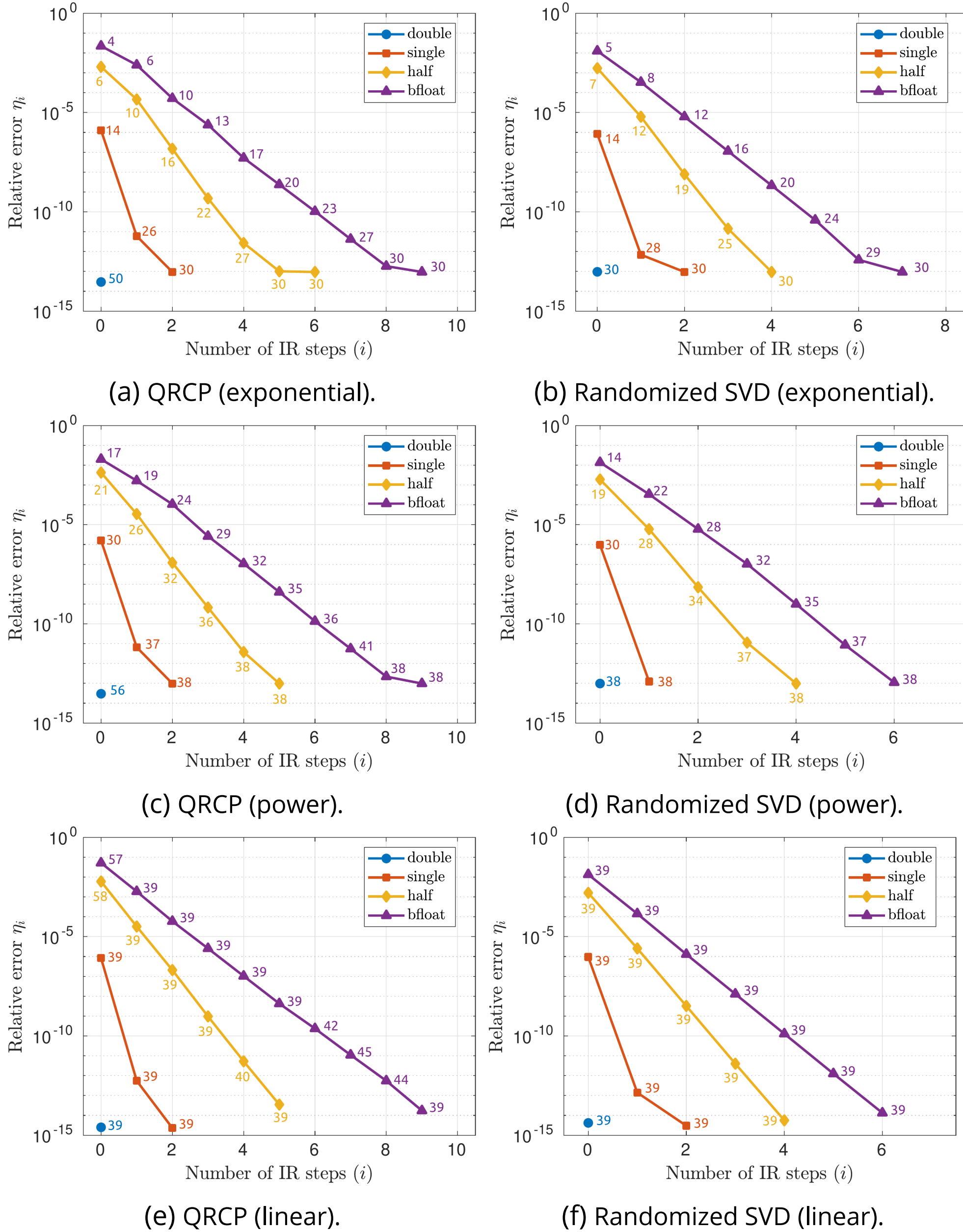
Finally, we discuss the rank behavior of F_i across IR steps. We see that it is roughly equal to $\texttt{rank}(X, \eta_i)$, the numerical rank of X at accuracy η_i. Thus, for matrices with rapidly decaying singular values such as the exponential case in Figure 2.2, these ranks tend to be much smaller at the early steps in low precision. Reflecting on the cost of the algorithm, we can expect the use of low precision plus refinement to be particularly cost-efficient for such matrices.

Figure 2.4 shows similar plots for the tensor decompositions (TTSVD, HOSVD, and HTSVD). The results for tensors follow the same trend, and lead to the same conclusion as for matrices. The main difference is that the rank of F_i (text labels) is now a vector instead.

Overall, our experiments confirm that Algorithm 2.1 is able to rapidly converge to a high accuracy, while using low precision for the LRA kernel. This observation is valid for all types of matrices and tensors in our test set, and all five `lra` kernels that we tested, which shows that Algorithm 2.1 is robust and can work in a wide variety of settings.

2.5.3 . Role of θ_ℓ

There is a trade-off in choosing the scaling factor θ_ℓ : the larger it is, the faster the convergence (since the error is reduced by a factor roughly equal



(a) QRCP (exponential).

(b) Randomized SVD (exponential).

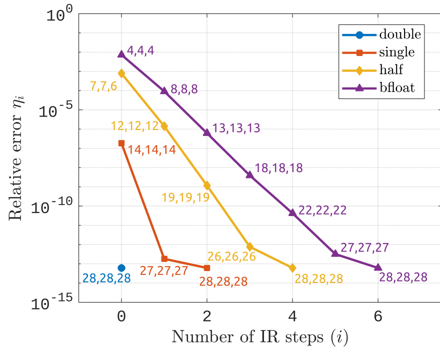
(c) QRCP (power).

(d) Randomized SVD (power).

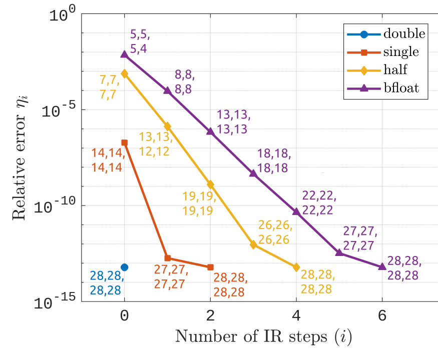
(e) QRCP (linear).

(f) Randomized SVD (linear).

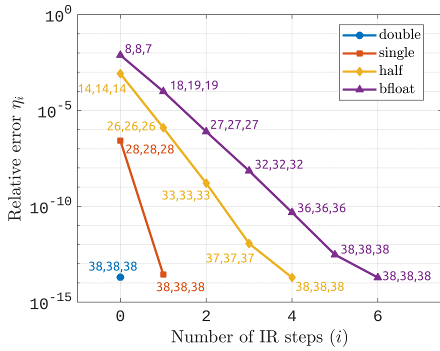
Figure 2.3 – Convergence of Algorithm 2.1 for three types of matrices (with different singular value distributions, see Figure 2.2) and for two different `lra` kernels (QRCP or randomized SVD). The number next to each marker indicates the rank of F_i after recompression.



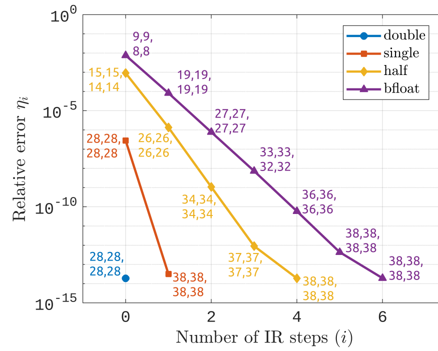
(a) TTSVD (exponential).



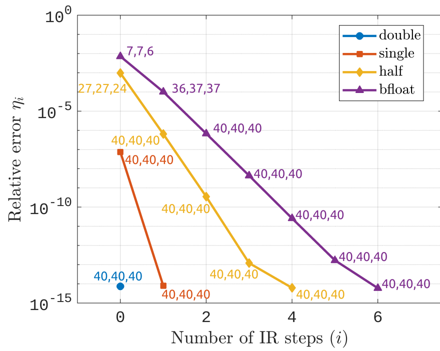
(b) HOSVD (exponential).



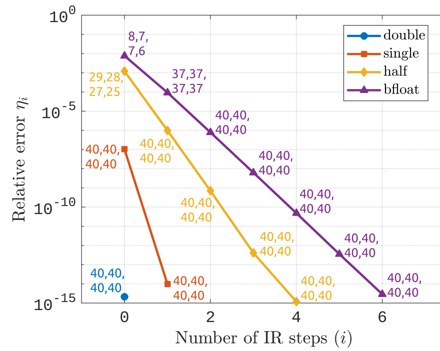
(c) TTSVD (power).



(d) HOSVD (power).



(e) TTSVD (linear).

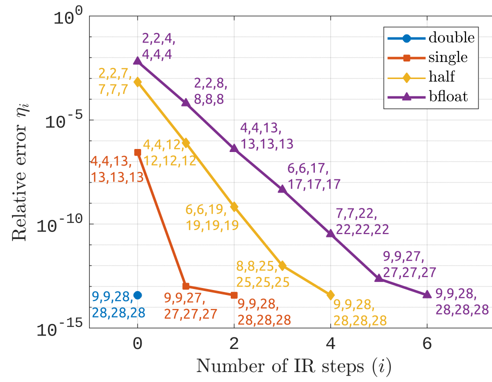


(f) HOSVD (linear).

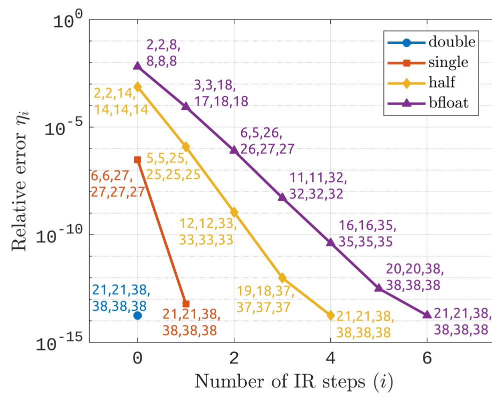
Figure 2.4 – Convergence of Algorithm 2.1 for three types of tensors (depending on the singular value distribution, see Figure 2.2) and with TTSVD or HOSVD as `lra` kernel. The numbers next to each marker indicate the rank vector of F_i after recompression.

to $\theta_\ell u_\ell$ at each refinement step), but if it is too large, the rank will no longer be correctly detected and this will lead to a significant rank growth.

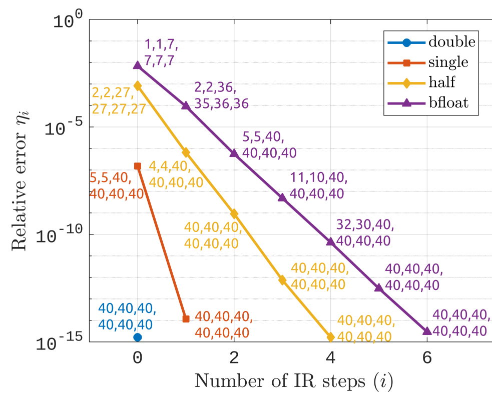
We illustrate this in Table 2.1 and Table 2.2, where we compare the convergence of Algorithm 2.1 for different values of θ_ℓ. Table 2.1 is for the TTSVD



(a) HTSVD (exponential).



(b) HTSVD (power).



(c) HTSVD (linear).

Figure 2.5 – Convergence of Algorithm 2.1 for three types of tensors (depending on the singular value distribution, see Figure 2.2) and with HTSVD as `lra` kernel. The numbers next to each marker indicate the rank vector of F_i after recompression.

kernel and Table 2.2 is for the QRCP kernel; fp16 arithmetic is used as the

Table 2.1 – Relative error η_i and $\texttt{rank}(F_i)$ at different steps i and for different values of θ_ℓ, for TTSVD (using fp16 as u_ℓ and exponential distribution of singular values).

	Relative error η_i					$\texttt{rank}(F_i)$				
$i \mid \theta_\ell$	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	6e-04	8e-04	9e-04	2e-03	5e-03	23,39,16	7, 7, 6	6, 6, 6	5, 5, 5	4, 4, 4
1	3e-07	1e-06	4e-06	1e-05	9e-05	14,14,15	12,12,12	11,11,11	10,10,10	8, 8, 8
2	2e-10	1e-09	8e-09	7e-08	6e-07	21,21,21	19,19,19	17,17,17	15,15,15	13,13,13
3	9e-14	8e-13	2e-11	4e-10	4e-09	29,29,29	26,26,26	23,23,23	20,20,20	18,18,18
4		6e-14	6e-14	1e-12	4e-11		28,28,28	28,28,28	26,26,26	22,22,22
5				6e-14	3e-13				28,28,28	27,27,27
6					6e-14					28,28,28

low precision u_ℓ in both cases. The tables show that the method converges faster as θ_ℓ increases, as expected : the error η_i is smaller for larger values of θ_ℓ. For TTSVD (Table 2.1), this faster convergence is achieved without compromising the correct rank detection, which remains contained throughout the iterations and for all values of $\theta_\ell \leq 1$. Thus, in this case, a large value of θ_ℓ is recommended. The situation is different for QRCP (Table 2.2), for which a too large value of θ_ℓ (here $\theta_\ell \geq 2^{-2}$) prevents the rank from being correctly detected, thus leading to a rank explosion.

We therefore conclude that the optimal choice of θ_ℓ depends on the LRA algorithm, and more specifically, on its sensitivity to rounding errors when detecting the numerical rank. Empirically, we have observed QRCP to be the most sensitive of the LRA kernels, and to a lesser extent the randomized SVD kernel ; the other kernels behaved well even for large θ_ℓ. For this reason, in our experiments we have set $\theta_\ell = 2^{-1}$ for all kernels except QRCP and randomized SVD, for which we have set $\theta_\ell = 2^{-3}$ and $\theta_\ell = 2^{-2}$, respectively. This setting allows to keep the ranks contained except for a few sporadic cases when using half precision. It must be noted that the sensitivity to rounding errors is also matrix (or tensor) dependent ; however, we do not tune θ_ℓ individually for each matrix/tensor, because this would not be representative of a realistic scenario where it is unknown in advance how sensitive the data at hand actually is.

2.5.4 . Results on real-life data

Finally, we also experiment our method on data sets (one matrix and one tensor) coming from real-life applications. The matrix corresponds to a subdomain in the discretization of the Poisson equation. It is of size 253×252 and its singular values decay relatively rapidly (not shown). The tensor comes from a quantum chemistry application and corresponds to an eigenfunction of the Hamiltonian operator for computing the vibrational spectrum of the H2CO molecule [52]. Here, the tensor is 6th-order ($d = 6$), of size

Table 2.2 – Relative error η_i and $\texttt{rank}(F_i)$ at different steps i and for different values of θ, for QRCP decomposition (using fp16 as u_ℓ and exponential distribution of singular values).

	Relative error η_i					$\texttt{rank}(F_i)$				
$i \mid \theta_\ell$	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	2e-03	2e-03	2e-03	2e-03	3e-03	13	9	9	7	6
1	2e-05	1e-05	1e-05	5e-05	1e-04	25	18	13	10	9
2	5e-08	8e-08	7e-08	1e-07	1e-06	72	43	20	16	14
3	2e-10	2e-10	4e-10	8e-10	8e-09	95	86	36	21	19
4	2e-12	2e-12	3e-12	2e-12	4e-11	27	27	29	27	24
5	7e-13	7e-13	7e-13	7e-13	1e-12	28	28	28	28	27
6					7e-13					28

$17 \times 17 \times 13 \times 13 \times 9 \times 9$, and obtained from an eigenvalue computation in which both the operator and vectors are compressed in the TT format throughout (and we obtain the full tensor with `decompress` in high precision in the end), and the precision is set to 10^{-9} to prevent rank explosion.

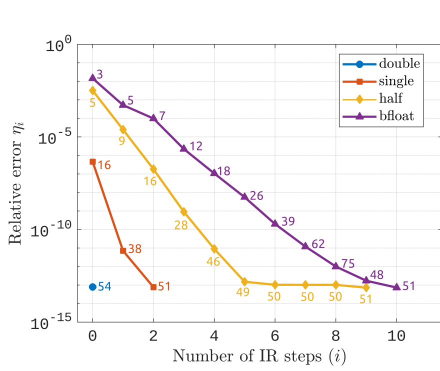
We report the results for these two problems in Figure 2.6-Figure 2.7 with all the corresponding LRA kernels, using the same legend and methodology as for the random data experiments. Overall, the results lead to the same trends and conclusions as the previous experiments ; ranks progressively increase with the IR steps, and our method manages to achieve high precision levels in all cases. In the TT case, ranks stop increasing around the compute precision of the application (10^{-9}), whereas in HOSVD and HTSVD cases we still observe a significant rank increase beyond this threshold. This is mostly due to the fact that the underlying tensor decomposition of this tensor is TT by construction, and this topology mismatch results in higher ranks.

2.5.5 . Estimation of the time cost and role of ω_ℓ

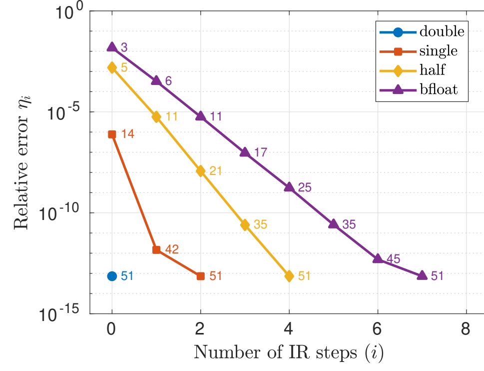
Based on the results obtained in the previous experiments, we can also obtain a rough estimation of the time cost of each method, using the formulas (2.41) (for classical hardware) or (2.44) (for hardware with block FMA units). This estimation depends on the number of iterations needed to converge to the desired accuracy, the ranks at each iteration, and the relative speed of the low precision arithmetic ω_ℓ.

We illustrate how this estimation can be obtained by using the case of randomized SVD with the Poisson matrix (Figure 2.6b) as an example. We consider two possible settings :

- Setting 1 : classical (Central Processing Unit (CPU)) hardware, where the relative speed of each arithmetic is proportional to the number of bits. For this setting we target an accuracy of $\varepsilon = 10^{-12}$, and we use fp64 as



(a) QRCP (Poisson matrix).



(b) Randomized SVD (Poisson matrix).

Figure 2.6 – Convergence of Algorithm 2.1 for a real-life matrix (Poisson) of size 253×252, for different `lra` kernels. The numbers next to each marker indicate the rank of F_i after recompression.

the high precision and fp32 as the low precision (hence $\omega_\ell = 0.5$).

- Setting 2 : GPU hardware with tensor core units (a type of block FMA). We target an accuracy of $\varepsilon = 10^{-5}$, and we use fp32 as the high precision and fp16 as the low precision. On most tensor core architectures (A100 and H100 in particular), fp16 arithmetic is 16 times faster than fp32, hence we take $\omega_\ell = \frac{1}{16} = 0.0625$.

In setting 1, our IR method (red curve in Figure 2.6b) converges in 2 iterations to an accuracy of $\varepsilon = 10^{-12}$. We use formula (2.41) with $\texttt{rank}(X, 10^{-6}) = 14$, $\texttt{rank}(X, 10^{-12}) = 42$, $c_1 = 6$, and $c_2 = 2$ to obtain an estimated cost of

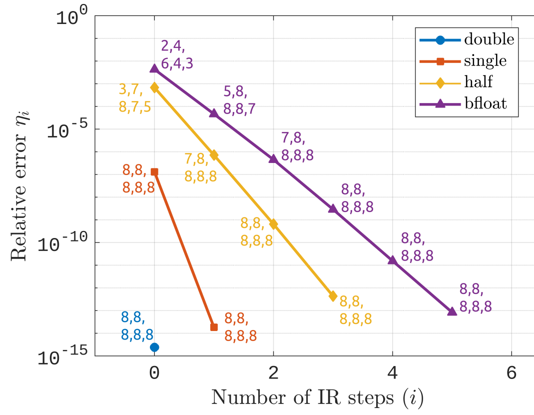
$$p \times \left(0.5 \times 6 \times (14 + 14 + 42) + 2 \times 14 + 3\right) = 241p. \quad (2.50)$$

This is to be compared with simply computing the LRA directly in fp64 arithmetic, for which formula (2.42) gives an estimated time cost of $6 \times 42 \times p = 252p$. Hence, we can expect a time reduction of about 4%. This example illustrates that even with classical hardware where the low precision is “only” twice as fast as the high precision, our IR method can still achieve moderate speedups in the case where the ranks decrease rapidly at low accuracies, such as for this Poisson matrix. This is however not the case for other matrices, including the random synthetic ones used previously.

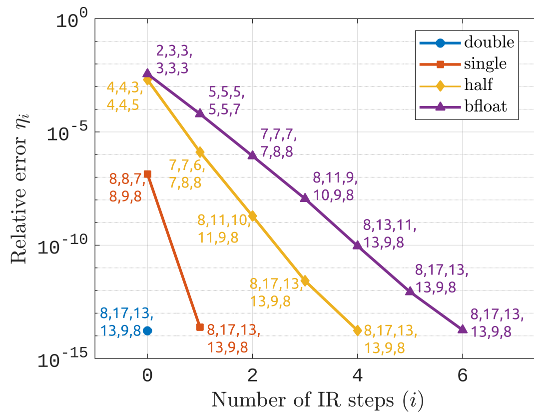
In setting 2, our IR method (yellow curve in Figure 2.6b) converges in 2 iterations to an accuracy of $\varepsilon = 10^{-5}$. We use formula (2.44) with $\texttt{rank}(X, 10^{-3}) = 5$, $\texttt{rank}(X, 10^{-5}) = 11$, $c_1 = 6$, and $c_2 = 2$ to obtain an estimated cost of

$$p \times \Big(0.0625 \times \big(6 \times (5 + 5 + 11) + 2 \times 5\big) + 3\Big) = 11.5p. \quad (2.51)$$

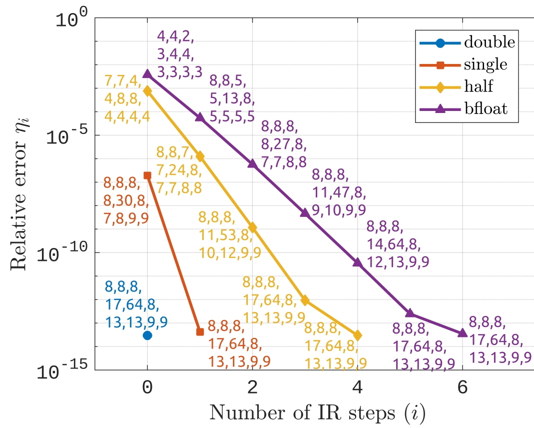
This is to be compared with simply computing the LRA directly in fp32 arithmetic, for which formula (2.42) gives an estimated time cost of $6 \times 11 \times p = 66p$.



(a) TTSVD (H2CO tensor).



(b) HOSVD (H2CO tensor).



(c) HTSVD (H2CO tensor).

Figure 2.7 – Convergence of Algorithm 2.1 for a real-life tensor (H2CO) of size $17 \times 17 \times 13 \times 13 \times 9 \times 9$, for different `lra` kernels. The numbers next to each marker indicate the rank of F_i after recompression.

Hence, we can expect a significant time reduction by a factor of almost $6\times$. This example illustrates that our IR method is particularly attractive on hardware with very fast low precision arithmetic.

The above examples are only meant to illustrate the trends that one might expect on different types of hardware and for different target accuracies and low precisions. This analysis can be similarly applied to the other matrices, LRA algorithms, and to tensors.

2.6 . Conclusion

We have presented a new mixed precision IR algorithm for computing low-rank matrix and tensor approximations. The algorithm first computes a LRA in low precision, and then computes another LRA of the error term, also in low precision, to refine the accuracy of the approximation. The process can be repeated to further refine the accuracy, and we ensure the rank of the approximation remains bounded by using inexpensive recompression operations. We have performed the error analysis of this algorithm, which proves that the low precision determines the convergence speed, whereas the attainable accuracy only depends on the high precision. Therefore, any desired level of accuracy can be attained, even though most of the operations are performed in low precision. This makes the algorithm computationally attractive, and we have performed a complexity analysis to determine specific conditions under which we can expect it to be cheaper than simply computing a LRA directly in high precision. We have applied our algorithm to various matrix and tensor LRAs algorithms, and performed MATLAB experiments that confirm its robustness and convergence in a wide range of settings. Our error analysis is based on a general model that assumes the use of numerically stable implementations for the basic kernels (in particular, LRA and recompression). For tensors, such computations are experimentally observed to behave stably in practice, but formally proving their stability remains an open problem that we will tackle in Chapter 4 of this thesis.

As evidenced by our complexity analysis, the potential of this IR method is especially high on modern architectures with very fast low precision arithmetic, such as GPU accelerators. Developing a high performance implementation of this method on such architectures is the goal of the next chapter (Chapter 3).

3 - Mixed precision randomized low-rank approximation with GPU tensor cores

3.1 . Introduction

Random projection methods are simple and robust techniques for reducing the dimensionality of data while preserving its structure [22]. These methods are widely used in machine learning and signal processing for tasks such as factorization [53]. Moreover, the matrix operations at the heart of these methods make them highly suitable for exploiting accelerators such as Graphics Processing Unit (GPU)s [23]. As explained in Section 1.4, modern GPUs provide extremely fast low precision arithmetics. The goal of this chapter is therefore to investigate to what extent these very fast low precision units can be exploited for accelerating randomized projection methods.

Specifically, in this chapter, we consider the fixed-rank randomized Low-rank approximations (LRA) algorithm from [22, Alg. 4.1] and describe in Algorithm 3.1. The main contribution of this chapter is the design of a new mixed precision version of this randomized LRA method, with a performance and accuracy analysis showing that the proposed method is able to exploit GPU tensor cores reliably and efficiently [54].

Our method is based on three key ideas :

- The first idea consists in *performing the matrix-matrix products (GEneral Matrix Multiplication (GEMM) kernel) in mixed precision arithmetic using the tensor cores*, since these operations represent the asymptotic bottleneck of the method. We compare several GEMM variants depending on how the conversions between fp32 and fp16 are handled, and identify one variant in particular that achieves the best performance-accuracy trade-off.
- Then, having significantly accelerated the GEMM operations, we observe that the orthonormalization step (QR kernel), despite requiring an asymptotically negligible number of flops, becomes the new performance bottleneck. Then the second idea is to *switch the orthonormalization method from the standard Householder QR to a CholeskyQR algorithm*, which mainly relies on GEMM and is therefore much more efficient on GPUs. We mitigate the inherent instability of CholeskyQR by performing it in fp64 rather than fp32 arithmetic.
- This leads to a mixed precision randomized LRA method employing three precisions (fp16, fp32, and fp64). We show that this method can be up to $8\times$ faster than the standard randomized LRA method in fixed precision fp32 arithmetic and achieves an average accuracy of order

10^{-2}, which may be sufficient for some applications. Then the third idea is to *use the iterative refinement method for LRA proposed in the previous chapter, Chapter 2 to improve the accuracy of the method*. We show that with refinement, the accuracy of this method improves significantly to an average of order 10^{-5}, while still being up to $2.2\times$ faster than the standard LRA method in fp32 arithmetic.

This chapter is structured as follows. In Section 3.2, we provide the necessary technical background on randomized LRA methods and discuss related work. In Section 3.3, we describe the proposed mixed precision method and its implementation using GPU tensor cores. In Section 3.4, we perform some experiments to analyze the performance and accuracy of our method. We finally provide our concluding remarks in Section 3.5.

3.2 . Background

3.2.1 . Randomized LRA

Algorithm 3.1 Randomized low-rank approximation fixed-rank variant.

Input : $A \in \mathbb{R}^{m\times n}$, the target rank k.
Output : $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{n\times k}$ such that $A \approx XY^T$.

1 : $\Omega \leftarrow \texttt{randn}(n,k)$
2 : $B \leftarrow A\Omega$
3 : $[X,\sim] \leftarrow \texttt{qr}(B)$
4 : $Y \leftarrow A^T X$

Given a matrix $A \in \mathbb{R}^{m\times n}$, we want to approximate A as a low-rank product XY^T of smaller matrices $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{n\times k}$, where the rank k is (much) smaller than min(m,n).

Among the many possible methods to compute LRA, randomized ones have encountered much success due to their ability to mainly rely on efficient matrix-matrix products. In this chapter, we focus on randomized LRA based on Gaussian sampling [22], as outlined in Algorithm 3.1. Commonly, an *oversampling* parameter is added to the rank k (e.g., $l = k+p$, p small oversampling) to ensure that the matrix B in line 2 is full rank, to do a final LRA after Line 4 to be more accurate. In our case, we set $p = 0$ to optimize the computation performance by avoiding a last LRA and assume the rank to be still small enough for analysis with the state-of-the-art.

We note that many alternative variants of Algorithm 3.1 are possible ; for example the sampling may be performed differently (e.g., via a fast Fourier transform), or we may compute specific types of LRA (e.g., singular value

decomposition (SVD)) by further decomposing $Q^T A$. In addition, while Algorithm 3.1 is a fixed-rank algorithm, fixed-accuracy variants have also been proposed [22, 31] and considered in Section 1.2.3 Algorithm 1.2, in which an accuracy threshold ε is prescribed and the rank k_ε is adaptively discovered by the algorithm. In this chapter, we focus on the fixed-rank case, which is simpler to implement. Extending our high performance implementation to the fixed-accuracy case is left for future work.

Algorithm 3.1 relies on two computational kernels : matrix-matrix products (GEMM kernel) and QR factorization (QR kernel). Importantly, the GEMM kernel performs $4mn\ell$ flops whereas the QR kernel only performs $c_{\text{qr}} m\ell^2$ flops (where c_{qr} is a small constant that depends on the specific QR factorization method that is used). Therefore, the performance of the overall method should be guided by the GEMM kernel, which can be performed very efficiently on modern computer architectures and especially on GPU accelerators.

As mentioned in Section 1.5.1, only Connolly, Higham, and Pranesh [41], Ootomo and Yokota [42], and Buttari, Mary, and Pacteau [43], start from an exact or fixed arithmetic context to propose mixed precision variants of randomized LRA. However, Connolly et al. methods do not exploit the high-speed low precision arithmetic available on GPU tensor cores to accelerate randomized LRA ; these are more MATLAB experiments. Our method proposed in the next section and its GPU implementation should, therefore, be most directly compared to the method of Ootomo and Yokota [42].

As explained in Section 1.5.1, their method does not reduce the accuracy of the LRA thanks to the emulation of fp32 arithmetic in the first GEMM and the use of fp32 arithmetic in the remaining kernels (QR and second GEMM). However, this limits the maximum speedup obtainable by this approach, since a large part of the computations is still executed in fp32 arithmetic ; Ootomo and Yokota [42] thus report a speedup of $1.28\times$ compared with randomized SVD entirely in fp32 arithmetic.

In contrast, our approach is more performance-driven : we use tensor core arithmetic in both GEMM kernels (without emulating fp32 arithmetic), obtaining speedups at the price of a lesser accuracy ; we then implement the iterative refinement method defined on the Chapter 2 on GPUs to improve the accuracy while retaining a good speedup.

To hold the line of this chapter, we recall an iterative refinement scheme Algorithm 3.2, which roughly sketches the work done in the previous chapter.

3.3 . Mixed precision randomized LRA on GPU tensor cores

In this section, we propose a mixed precision variant of `randLRA`, the randomized LRA method outlined in Algorithm 3.1, and describe its GPU imple-

Algorithm 3.2 Randomized LRA with iterative refinement.

Input : $A \in \mathbb{R}^{m\times n}$, the target rank k, the oversampling p.
Output : $X \in \mathbb{R}^{m\times 3k}$ and $Y \in \mathbb{R}^{n\times 3k}$ such that $A \approx XY^T$.

1 : $[X_1, Y_1] = \texttt{randLRA}(A, k, p)$ {in low precision}
2 : $E = A - X_1 Y_1^T$ {in high precision}
3 : $[X_2, Y_2] = \texttt{randLRA}(E, 2k, p)$ {in low precision}
4 : $X = [X_1, X_2]$
5 : $Y = [Y_1, Y_2]$

mentation. As mentioned, `randLRA` relies primarily on two kernels : the matrix-matrix product (GEMM kernel) and the QR factorization (QR kernel).

3.3.1 . GEMM kernel

The GEMM kernel $C = AB$ can be executed up to $16\times$ faster using GPU tensor core units. However, these units require the input matrices A and B to be represented in fp16 which necessitates a conversion when they are originally stored in fp32. We can distinguish several variants depending on which matrices are converted (A and B only, or also C), and on whether these conversions are handled explicitly or implicitly. Indeed, a first option is to handle conversions implicitly by keeping the matrices in the input fp32 format and letting cuBLAS itself perform the conversions to fp16 ; the advantage of this approach lies in the simplicity of the code and the efficiency of the conversion which is handled by the optimized library. In the explicit approach, on the other hand, we convert the input matrices to fp16 ourselves before calling the cuBLAS tensor core GEMM ; even though our own conversion routine might be less efficient, the advantage of this approach is that matrices that were already converted to fp16 can be reused in other calls to tensor core GEMM without the need for further conversions (note that this however requires extra storage to store the explicitly converted matrix).

In summary, we evaluate three variants of the GEMM kernel. We denote these variants as `tgemm` (to indicate the use of tensor cores) and use the subscripts "$32|32$", "$16|16$", "$16|32$" to indicate the precision type of the input (A and B) and output (C). Note that if the input type is fp32, an implicit conversion to fp16 is performed, whereas, if the output type is fp32, no conversion to fp16 is required because tensor cores have the ability to accumulate directly in fp32 [17].

- $\texttt{tgemm}_{32|32}$: A, B, C are all stored in fp32 ; the GEMM implicitly converts A and B to fp16 during the computation but keeps C in fp32.
- $\texttt{tgemm}_{16|16}$: A, B, C are all explicitly converted from fp32 to fp16 before the computation of the GEMM, which does not need any conversions.
- $\texttt{tgemm}_{16|32}$: A and B are explicitly converted to fp16 but C is kept in

fp32 ; the GEMM accumulates the computation in C in fp32 arithmetic and thus requires no further conversions.

We will compare the performance-accuracy trade-off achieved by each of these three variants in our benchmarks in the next section.

3.3.2 . QR kernel

The second main kernel of `randLRA` is the QR factorization kernel. Several methods exist that achieve different trade-offs between efficiency and stability, for example, classical or modified Gram-Schmidt, or Householder QR. The most stable approach is the Householder QR factorization, which is implemented in GPU libraries such as MAGMA and cuSOLVER. Unfortunately, Householder QR is also inefficient on GPUs due to its limited parallelism, and current implementations do not exploit tensor core arithmetic. As a result, in the context of `randLRA` on GPU tensor cores, even though the GEMM kernel requires in theory an asymptotically dominant number of flops, in practice the QR kernel becomes the performance bottleneck. This is because the GEMM kernel strongly benefits from GPUs and especially mixed precision tensor core units, whereas the QR kernel is less efficient on GPUs and cannot exploit tensor core units.

Motivated by these observations, we propose to use instead the Cholesky QR factorization, a much faster variant of QR which mainly relies on matrix-matrix products and can thus exploit GPUs much more efficiently. Cholesky QR orthonormalizes a tall-skinny matrix A by computing the Gram matrix $B = A^T A$, computing its Cholesky factorization $R^T R = B$, and obtaining the orthonormal factor by the triangular solve $Q = AR^{-1}$. Unfortunately, the condition number of the Gram matrix $\kappa(B) = \kappa(A)^2$ is large even for moderately ill-conditioned A, which makes Cholesky QR unstable due to the possible breakdown of the Cholesky factorization of B if B is singular in the working precision.

In fp32 arithmetic such breakdowns are expected to occur when $\kappa(A) \gtrsim 10^4$. In order to address this issue, we switched the Cholesky QR factorization from fp32 to fp64 arithmetic ; in this case, breakdowns can only occur when $\kappa(A) \gtrsim 10^8$. Since in the context of `randLRA` the input matrix is stored in fp32 arithmetic, breakdowns should thus never occur with Cholesky QR in fp64 arithmetic. The resulting algorithm is outlined in Algorithm 3.3. We note that the final triangular solution step (line 4) could be performed in fp32 without affecting stability [55] ; this is an improvement that we will investigate in future work.

3.3.3 . Randomized LRA

Having discussed the implementation of the GEMM and QR kernels on GPUs, we can now present the implementation of `randLRA`, outlined in Algorithm 3.4.

Algorithm 3.3 Cholesky QR kernel implementation on GPU.

Input : $A_{32} \in \mathbb{R}^{m \times n}$.
Output : Orthonormal factor $Q_{32} \in \mathbb{R}^{m \times n}$ of A_{32}.

1 : $A_{64} = \texttt{fp64}(A_{32})$
2 : $B_{64} = A_{64}^T A_{64}$
3 : $R_{64} = \texttt{chol}(B_{64})$
4 : $Q_{64} = A_{64} R_{64}^{-1}$
5 : $Q_{32} = \texttt{fp32}(Q_{64})$

Depending on the GEMM variant used, `randLRA` takes different forms ; in Algorithm 3.4, we describe the case where $\texttt{tgemm}_{16|32}$ is used, which allows for explicitly converting A to fp16 only once (line 3) and reusing it in both GEMM calls (lines 3 and 6). The QR kernel (line 4) can be either standard Householder QR in fp32 arithmetic, or the mixed precision Cholesky QR kernel presented previously (Algorithm 3.3).

Algorithm 3.4 Mixed precision `randLRA` on GPU tensor cores.

Input : $A_{32} \in \mathbb{R}^{m \times n}$, the target rank k.
Output : $X_{16} \in \mathbb{R}^{m \times k}$ and $Y_{16} \in \mathbb{R}^{n \times k}$ such that $A_{32} \approx X_{16} Y_{16}^T$.

1 : $\Omega_{16} = \texttt{randn}(n, k)$ {in fp16}
2 : $A_{16} = \texttt{fp16}(A_{32})$
3 : $B_{32} = \texttt{tgemm}_{16|32}(A_{16}, \Omega_{16})$ {with fp16/fp32 tensor cores}
4 : $Q_{32} = \texttt{qr}(B_{32})$
5 : $X_{16} = \texttt{fp16}(Q_{32})$
6 : $Y_{32} = \texttt{tgemm}_{16|32}(A_{16}^T, X_{16})$ {with fp16/fp32 tensor cores}
7 : $Y_{16} = \texttt{fp16}(Y_{32})$

Finally, we describe in Algorithm 3.5 our implementation of the iterative refinement approach [56] on GPU tensor cores, using Algorithm 3.4 as the low precision `randLRA` method.

Algorithm 3.5 Mixed precision `randLRA` on GPU tensor cores, with iterative refinement.

Input : $A_{32} \in \mathbb{R}^{m \times n}$, the target rank k.
Output : $X_{16} \in \mathbb{R}^{m \times 3k}$ and $Y_{16} \in \mathbb{R}^{n \times 3k}$ such that $A_{32} \approx X_{16} Y_{16}^T$.

1 : $[X_{16}, Y_{16}] = \texttt{randLRA}(A_{32}, k)$
2 : $E_{32} = A_{32} - \texttt{tgemm}_{16|32}(X_{16}, Y_{16}^T)$
3 : $[X'_{16}, Y'_{16}] = \texttt{randLRA}(E_{32}, 2k)$
4 : $X_{16} = [X_{16}, X'_{16}]$
5 : $Y_{16} = [Y_{16}, Y'_{16}]$

3.4 . Experiments

3.4.1 . Experimental setting

All experiments have been carried out on the Jean Zay supercomputer located at IDRIS [1]. Each node is equipped with 2 Intel Cascade Lake 6240R processors and 8 NVIDIA A100 PCIe 40 GB GPUs, for a total memory of 768 GB. Although each node has several GPUs, we use a single GPU for these experiments. On both architectures, we use CUDA 12.0.0. The CUDA package provides access to the libraries cuBLAS, cuSOLVER and cuRAND.

The matrices used in our experiments are randomly generated. Given the specified rank k, we generate two random Gaussian matrices $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{n\times k}$ and define $A = XY^T$. In all experiments we do not use any oversampling ($p = 0$).

We measure the performance of our algorithms in number of Tera floating-point operations per second (TFLOPS), that is,

$$\text{Performance } = \frac{N_{\text{flops}}}{10^{12} \cdot \text{time}}. \tag{3.1}$$

To measure the "effective" performance of the algorithms, we use the same reference number of flops N_{flops} for all of them, regardless of their actual number of flops. Specifically we use

$$N_{\text{flops}} = 4mnk + 2nk^2 - \frac{2}{3}k^3 + O(mn), \tag{3.2}$$

which corresponds to the number of flops of the baseline version, which performs only two GEMMs and one QR factorization.

3.4.2 . Performance and accuracy of kernels

In this section, we evaluate the performance of the building blocks of our `randLRA` algorithm : the GEMM and QR kernels.

GEMM kernel

We begin by comparing in Figure 3.1a the performance of the standard GEMM in fp32 arithmetic (`sgemm`) with the three `tgemm` variants that use the tensor cores described previously. The figure shows the performance for computing $C = AB$ where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times k}$, where $m = n = 35840$ are fixed and where k varies from 8 to 1024. (This shape of matrices corresponds to the two GEMMs performed by `randLRA`).

For the $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ variants, we plot their performance both with and without including the time taken by the explicit conversion of the

1. `http://www.idris.fr/eng/jean-zay/index.html`

matrices to fp16. For the $\texttt{tgemm}_{32|32}$ the implicit conversion is always performed and thus always included. The figure shows that, as expected, the implicit conversion performed by $\texttt{tgemm}_{32|32}$ is more efficient than the explicit one performed by our own implementation, so that this variant is faster than $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ if we include the time for explicit conversion. Interestingly, the relative cost of the conversion decreases as the dimension k increases, so that the performance of $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ including the conversion eventually becomes comparable to that of $\texttt{tgemm}_{32|32}$ for a sufficiently large k. More importantly, if we do not need to perform this conversion (because the input matrix is already stored in fp16), then the $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ variants become significantly faster than the $\texttt{tgemm}_{32|32}$ one.

We will investigate the difference in accuracy of these three variants directly in the context of their use in `randLRA`.

QR kernel

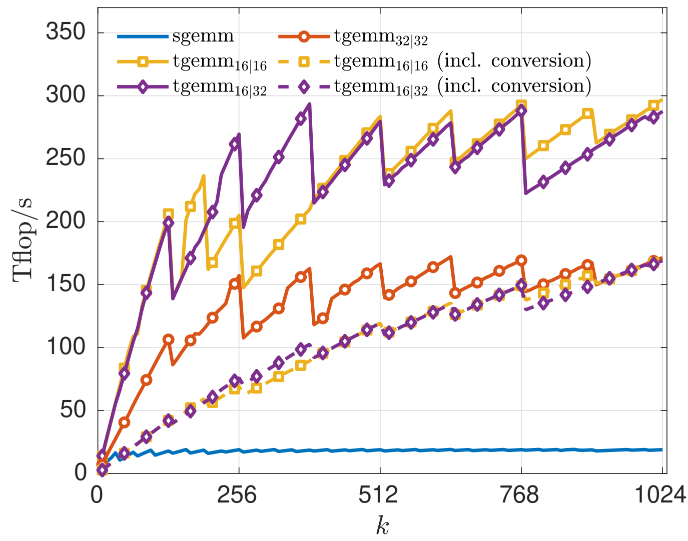
We now turn to the performance of the QR kernel, reported in Figure 3.1b. The figure compares the performance for orthonormalizing a matrix $B \in \mathbb{R}^{n\times k}$ where $n = 35840$ is fixed and where k varies from 8 to 1024. (Again, this corresponds to the shape of matrix arising in the QR kernel in `randLRA`).

We compare the classical Householder QR algorithm implemented in cuSOLVER using fp32 arithmetic (`sgeqrf`) with the Cholesky QR algorithm, using either fp32 or fp64 arithmetic (`scholQR` and `dcholQR`, the latter corresponding to Algorithm 3.3). At the time of these experiments, the only GPU implementation of Cholesky QR that we found is the one available in the MAGMA library. However, we found MAGMA's implementation not to be efficient for the target sizes in our context, and therefore we made our own implementation.

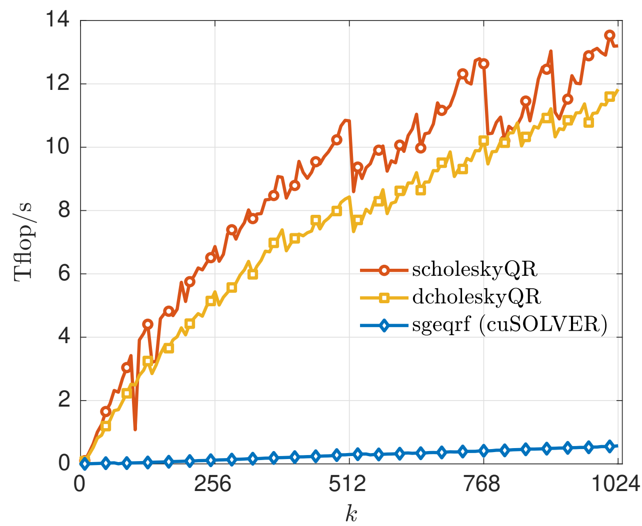
Regarding the risk of Cholesky breaking down, in experiments with `randLRA` using Cholesky QR in fp32 (not shown), we found a significant fraction (about 14%) of breakdowns, which disappeared by using fp64 arithmetic instead. In comparison, Householder QR is robust (even in fp32 arithmetic), but extremely slow, as expected. Overall, our implementation of Cholesky QR in fp64 can be more than $20\times$ faster than Householder QR in fp32.

3.4.3 . Mixed precision randomized LRA

We now evaluate the performance and accuracy of randomized LRA, without iterative refinement to begin. We compare eight different variants. Two of them correspond to Algorithm 3.1 with the GEMM in standard fp32 arithmetic (`sgemm`), and with either fp32 Householder QR (`sgeqrf`) or fp64 Cholesky QR (`dcholQR`). The other six variants correspond to Algorithm 3.4, using one of the three `tgemm` variants for GEMM and again either fp32 Householder QR or fp64 Cholesky QR. Figure 3.2a plots the performance of these eight variants,



(a) GEMM



(b) QR

Figure 3.1 – Performance of the GEMM and QR kernels.

and Figure 3.2b plots their relative accuracy, measured as $\|A - XY^T\|/\|A\|$, where $\|\cdot\|$ denotes the Frobenius norm.

As a general preliminary observation, note that the use of Householder or Cholesky QR does not impact the accuracy for any of the variants (and so the latter is preferable since it is faster).

Let us first analyze the baseline variant using `sgemm` (fp32 arithmetic without tensor cores). This is the most accurate variant, with an average error of order 10^{-4}. However, this is also the slowest variant since it does not benefit from tensor cores : its performance is almost constant as soon as $k \geq 256$ and is limited to only 14 TFLOPS with Householder QR. The use of Cholesky QR slightly improves its performance but still remains limited to only 17 TFLOPS.

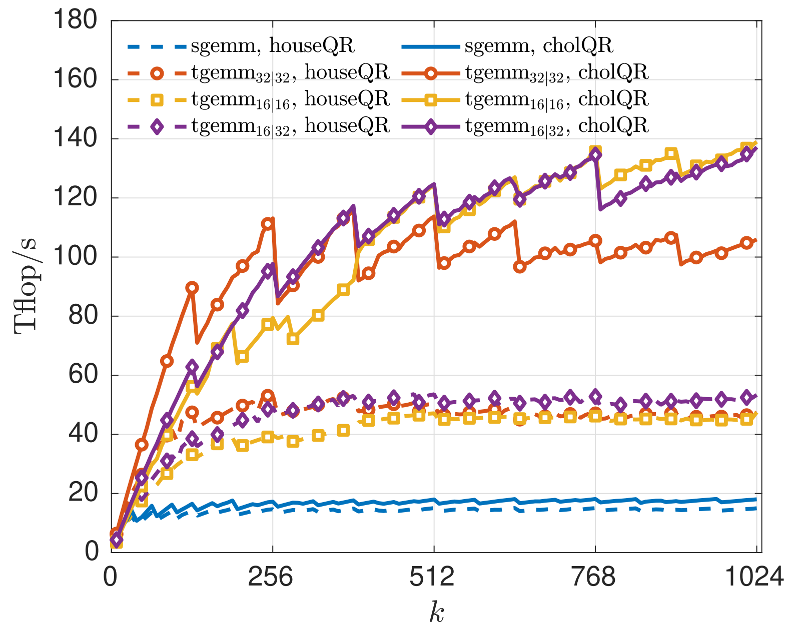
Then, let us now analyze the variants using `tgemm` (mixed precision GEMMs with tensor cores). Regardless of the choice of `tgemm` variant, using Householder QR limits the attainable performance to at best 50 TFLOPS. In this case, the GEMM performs well, but performance is limited by the performance of Householder QR, which is extremely slow as previously analyzed, and thus becomes the bottleneck, even though it requires an asymptotically negligible number of flops compared with GEMM. Therefore, for these `tgemm` variants, using Cholesky QR significantly improves performance by moving the bottleneck back to the GEMM.

Let us finally compare the three different `tgemm` variants to determine which is preferable. We can see in Figure 3.2b that $\texttt{tgemm}_{16|16}$ is much less accurate than both $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$, with an average error of order 10^{-1} instead of 10^{-2}. This comes from C being stored in fp16 : indeed, even though the tensor cores accumulate the operations in fp32 arithmetic, writing them in a matrix C stored in fp16 generates fp16 rounding errors which lead to a significant loss of accuracy. This effect has been well characterized in the literature, see in particular Blanchard et al. [17] for an error analysis and Lopez and Mary [57] for the consequence of this observation on LU factorization.

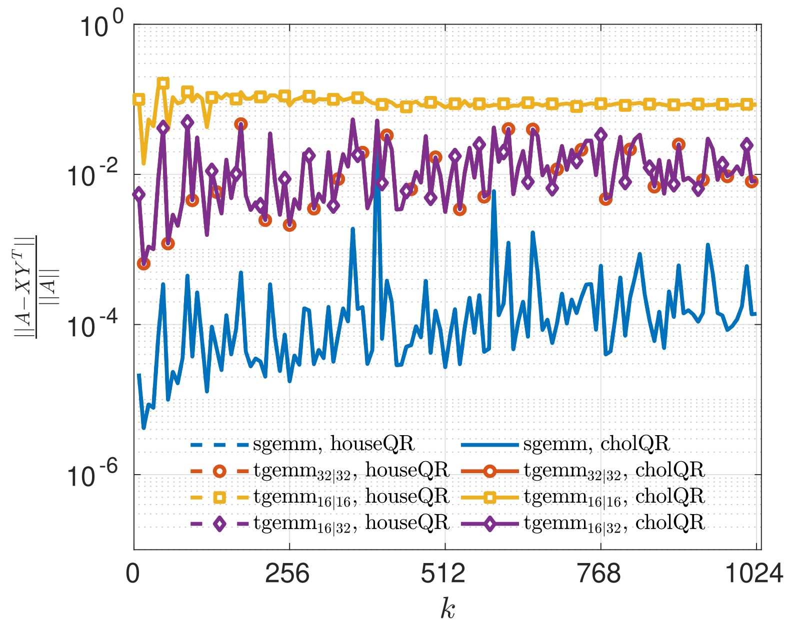
The choice of GEMM variant therefore comes down to which of $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ is faster. As Figure 3.2a shows, this depends on the rank k. When k is small the relative cost of the conversion with respect to the computation is large and so $\texttt{tgemm}_{32|32}$ is faster than $\texttt{tgemm}_{16|32}$. As k increases the conversion becomes less and less costly with respect to the computation so eventually (here for $k \gtrsim 256$) $\texttt{tgemm}_{16|32}$ becomes faster, reaching up to 140 TFLOPS for large values of k. Therefore, `randLRA` and $\texttt{tgemm}_{16|32}$ is up to $8\times$ faster than `randLRA` with `sgemm`, at the price of a lesser accuracy in this case without iterative refinement.

3.4.4 . Iterative refinement

Finally, we conclude by evaluating the performance and accuracy of `randLRA` using Iterative Refinement (IR). We compare the same eight variants



(a) Performance



(b) Accuracy

Figure 3.2 – Performance and accuracy of `randLRA` *without* refinement. (In (b), `tgemm`$_{32|32}$ and `tgemm`$_{16|32}$ completely overlap both for Householder QR and Cholesky QR).

as in the previous section, except that the variants that exploit mixed precision tensor cores (tgemm) now use IR (Algorithm 3.5); the two variants using sgemm *do not* use IR, since their accuracy is already satisfactory; we keep them as a reference point. Figure 3.3a plots the performance and Figure 3.3b plots the relative accuracy.

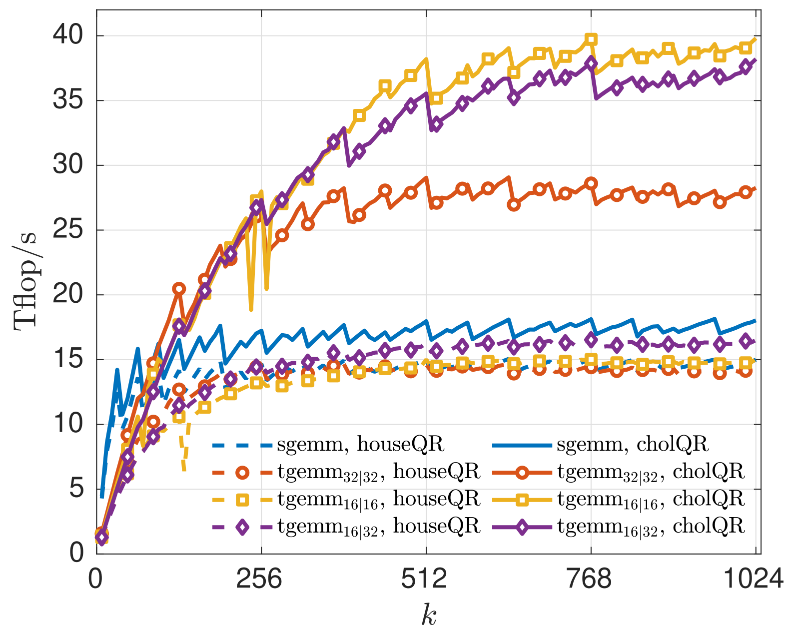
In terms of accuracy, Figure 3.3b confirms that the use of IR significantly improves the accuracy of all mixed precision variants. Specifically, the variants using $\texttt{tgemm}_{16|16}$ achieve an average error of order 10^{-2} instead of 10^{-1} and, more interestingly, the variants using $\texttt{tgemm}_{32|32}$ or $\texttt{tgemm}_{16|32}$ achieve an average error of order 10^{-5} instead of 10^{-2}. Thus, IR makes randLRA with these variants of tgemm at least as accurate, and in many cases even more accurate, than the standard randLRA with sgemm.

It remains to evaluate the impact of IR on performance. Note that the use of IR increases the number of flops by about a factor 4 (two GEMMs with rank k and three GEMMs with rank $2k$, instead of two GEMMs of rank k). For large values of k (for which the maximum performance is attained), the use of IR makes randLRA with $\texttt{tgemm}_{32|32}$ about $3.8\times$ slower and randLRA with either $\texttt{tgemm}_{16|16}$ or $\texttt{tgemm}_{16|32}$ about $3.5\times$ slower. The fact that the relative performance of $\texttt{tgemm}_{32|32}$ compared with $\texttt{tgemm}_{16|16}$ and $\texttt{tgemm}_{16|32}$ decreases when IR is used is explained by the fact that the relative weight of the conversions decreases with IR. In any case, the important conclusion is that the $\texttt{tgemm}_{16|32}$ variant with IR remains much faster than the variant with sgemm and no IR, with a speedup of up to a factor $2.2\times$. We therefore have obtained a method that is both faster *and* more accurate than the fp32 randLRA baseline.

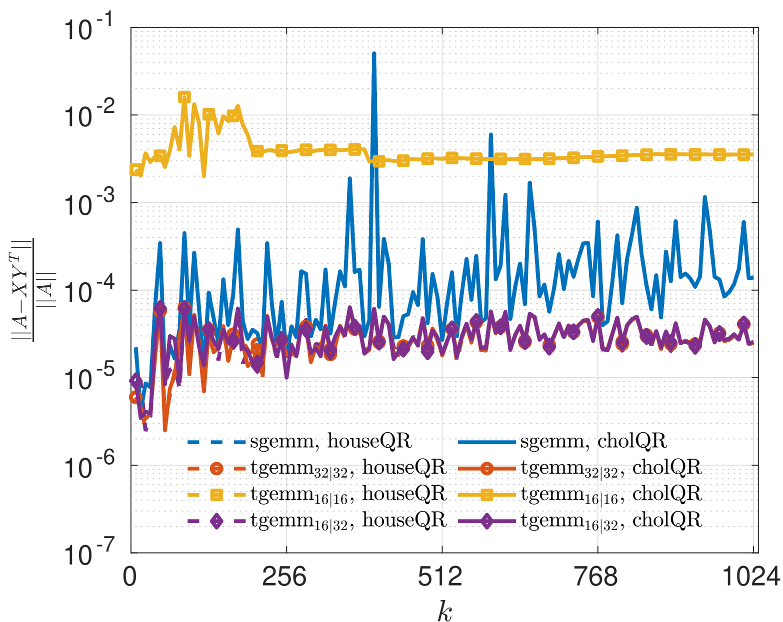
3.5 . Conclusion

We have proposed a new randomized LRA method that efficiently and reliably exploits mixed precision GPUs. Our method, outlined in Algorithm 3.5, combines three key ideas. First, we use the GPU tensor core units to accelerate the matrix-matrix products (GEMM kernel) while minimizing the accuracy loss by using mixed precision arithmetic (matrices are converted to fp16, but computations are accumulated in fp32). Second, we replace the standard Householder QR by Cholesky QR, which is much more efficient on GPU, and we mitigate its inherent instability by performing it in fp64 arithmetic. Third and lastly, we implement the iterative refinement approach proposed in Chapter 2 to recover full fp32 accuracy. Overall, our method achieves an accuracy that is at least as good and in many cases even better than a standard randomized LRA in fp32 arithmetic, while being up to $2.2\times$ faster.

Our work illustrates the convergence of approximate computing techniques by combining LRAs, randomization, mixed precision arithmetic, and GPU acceleration. Our results not only highlight the effectiveness of using



(a) Performance



(b) Accuracy

Figure 3.3 – Performance and accuracy of `randLRA` *with* refinement. (In (b), $\texttt{tgemm}_{32|32}$ and $\texttt{tgemm}_{16|32}$ completely overlap both for Householder QR and Cholesky QR).

mixed precision GPUs for accelerating randomized LRA while preserving a satisfying accuracy, but also pave the way toward exploring other similar GPU-based approximate methods in linear algebra and beyond.

4 - Numerical stability of tree tensor network operations, and a stable rounding algorithm

4.1 . Introduction

This chapter is concerned with tensor computations in finite precision arithmetic and aims to answer the following questions :

1. What are the key properties that guarantee the stability of tensor computations in finite precision arithmetic ?
2. How can we design tensor algorithms that are guaranteed to be stable in low precision ?

Our contribution towards tackling these questions is two-fold. First, we propose a general framework based on a tree tensor network format that encompasses the most important tensor formats (tensor-train (TT), Tucker, and hierarchical Tucker (HT)) and we define the essential operations that are needed to express some of the most common computations of interest. We then carry out an error analysis of an abstract computation in this framework and identify conditions to guarantee its stability. In particular, a key condition is that the norm of the tensor should be tightly concentrated around a single node of the network, a property that we will formalize in definition 4.2 of our framework. This property is notably satisfied when all nodes of the tensor except one are semi-orthogonal. Our analysis shows that if this property is maintained throughout the computation then the error introduced by each operation is controlled in terms of the norm of the global tensor. This first contribution can thus be used to establish the stability of a wide range of tensor computations that can be expressed in our framework.

Our second contribution is to use our framework to propose a general rounding algorithm : tensor rounding [21, p. 8] is key operation which consists in finding the optimal ranks for a prescribed tensor structure and a prescribed accuracy. In view of the conclusions of our error analysis, the algorithm is careful to maintain the semi-orthogonality of all nodes except one throughout the computation, and is thus guaranteed to be stable. The algorithm works for any tree topology of tensor and can thus be applied to a wide range of tensor formats. We compare this rounding algorithm with the existing Gram SVD-based rounding for hierarchical Tucker tensors proposed in [14], which is unstable [25]. We show that our algorithm can significantly improve the accuracy of the rounding in finite precision arithmetic, and can thus more reliably exploit the low precision arithmetics available on modern hardware.

The rest of this chapter is structured as follows. In Section 4.2 we present our general framework for tensor computations. Then, in Section 4.3, we carry out the error analysis of tensor computations based on our framework and identify conditions to guarantee their stability. We then exploit our framework to establish the stability of some common tensor computations in Section 4.4. We propose a tensor rounding algorithm in Section 4.5 that is guaranteed to be stable. In Section 4.6, we perform some numerical experiments that illustrate the stability of our rounding algorithm and compare it to the unstable Gram SVD-based one. Finally, we provide our concluding remarks in Section 4.7.

4.2 . The framework

In this section we present the framework that we will use to develop our analysis and algorithms. First, in Section 4.2.1, we introduce some basic definitions and notations. Then, in Section 4.2.2, we define the four key kernels that operate locally on a tree tensor network. Finally, in Section 4.2.3, we define a key property of the network that our analysis will require.

4.2.1 . Definitions and notations

Our framework is based on tree tensor networks [58, 59] already defined in Section 1.3.1, and illustrated in Figure 1.9. Given two tensor networks $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, we define the network $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\boldsymbol{\mathcal{B}}$ as the contraction of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ along one or multiple specified dimensions (which are left implicit as they will be apparent from the context).

In the following, to clearly distinguish the objects under consideration, we will use the following notation :

- scalars are denoted with lower case letters (in particular, we will denote the inner edges as r_i and the outer edges as n_i);
- matrices are denoted with upper case letters (e.g., X, U);
- tensors (nodes of a tensor network) are denoted with calligraphic upper case letters (e.g., $\mathcal{X}, \mathcal{U}$);
- and tensor networks are denoted with bold calligraphic upper case letters (e.g., $\boldsymbol{\mathcal{X}}$).

We define $\|\boldsymbol{\mathcal{A}}\| = \|\mathcal{A}\|$, that is, the norm of the network is the norm of the full tensor it represents. As a result the usual submultiplicativity of the Frobenius norm is preserved for tensor networks :

$$\|\boldsymbol{\mathcal{A}}\boldsymbol{\mathcal{B}}\| = \|\mathcal{A}\mathcal{B}\| \leq \|\mathcal{A}\|\|\mathcal{B}\| = \|\boldsymbol{\mathcal{A}}\|\|\boldsymbol{\mathcal{B}}\|. \tag{4.1}$$

Moreover, we extend the definition of semi-orthogonality to tensors as follows : given two adjacent nodes $\mathcal{A}$ and $\mathcal{B}$, we call $\mathcal{A}$ semi-orthogonal *in the*

direction of $\mathcal{B}$ if the matricization A of $\mathcal{A}$ in the direction of $\mathcal{B}$ is a matrix with orthonormal columns (when considering the contraction $\mathcal{AB}$) or with orthonormal rows (when considering the contraction $\mathcal{BA}$). The invariance of the Frobenius norm is preserved with this extended definition : if $\mathcal{A}$ is semi-orthogonal in the direction of $\mathcal{B}$ then

$$\|\mathcal{AB}\| = \|\mathcal{B}\|. \tag{4.2}$$

Finally, we define $\boldsymbol{\mathcal{A}}+\boldsymbol{\mathcal{B}}$ as the network that represents the tensor $\mathcal{A}+\mathcal{B}$; note that its nodes are *not* the sum of the nodes of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, but rather their concatenation. An important case where this simplifies is when $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ only differ by one node : if $\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{X}}_1\mathcal{C}\boldsymbol{\mathcal{X}}_2$ and $\boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}}_1\mathcal{D}\boldsymbol{\mathcal{X}}_2$, then

$$\boldsymbol{\mathcal{A}}+\boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}}_1\mathcal{C}\boldsymbol{\mathcal{X}}_2 + \boldsymbol{\mathcal{X}}_1\mathcal{D}\boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1(\mathcal{C}+\mathcal{D})\boldsymbol{\mathcal{X}}_2. \tag{4.3}$$

4.2.2 . Operations on tree tensor networks

In order to express various computations of interest in our framework, we define four key kernels that operate locally on a tree tensor network. These four kernels are `matricize`, `tensorize`, `split`, and `merge`.

- `matricize`$(\mathcal{U}, r)$ matricizes the tensor $\mathcal{U}$ along the dimension r (which may be an inner or outer edge). This operation is illustrated in Figure 4.1, where $\mathcal{U}_3$ is a tensor of dimensions $r_1 \times r_2 \times r_3$ that is matricized along the dimension r_2. This results in matrix U_3 of dimensions $r_2 \times r_1 r_3$. To leave the representation of the entire network unchanged, we add a "phantom node" representing the identity tensor $\mathcal{I}$; note that this node is purely conceptual and never actually stored. In the case where r is an inner edge, we will also write `matricize`$(\mathcal{U}, \mathcal{V})$, where $\mathcal{V}$ is the node connected to $\mathcal{U}$ by edge r.
- `tensorize`(U) is the inverse operation of `matricize` : it reshapes the matrix U into its tensor form $\mathcal{U}$ by contracting it with the phantom core $\mathcal{I}$ (note that this contraction is conceptual and does not require any operation). This operation is also illustrated in Figure 4.1 (going from right to left).
- `split`(A) decomposes a matrix node A into the product of two matrices B and C. This operation is illustrated in Figure 4.2, where the node $U_2 \in \mathbb{R}^{n_1 \times r_1}$ is split into the product BC of $B \in \mathbb{R}^{n_1 \times r_0}$ and $C \in \mathbb{R}^{r_0 \times r_1}$.
- `merge`$(\mathcal{B}, \mathcal{C})$ is the inverse operation of `split` : given two tensors $\mathcal{B}$ and $\mathcal{C}$, it merges them into a single node $\mathcal{A}$ representing their contraction $\mathcal{BC}$ along the dimension that connects them. This operation is also illustrated in Figure 4.2 (going from right to left).

Note that since these kernels modify the network, formally we should pass $\boldsymbol{\mathcal{X}}$ as input/output, that is, we should write $\boldsymbol{\mathcal{X}} =$ `matricize`$(\mathcal{U}, r, \boldsymbol{\mathcal{X}})$ and so

on. To keep the notation light, we instead consider these kernels to be member functions of the network, so that it is implicit that they modify the network.

The `merge` kernel allows us to express contractions and, more specifically, matrix–matrix products when its input are matrix nodes. The `split` kernel allows us to express various types of matrix decompositions, such as QR or truncated singular value decomposition (SVD) decompositions. Together with `matricize` and `tensorize`, these four kernels are sufficient to express a wide range of tensor computations of interest. We will provide some examples in Section 4.4.

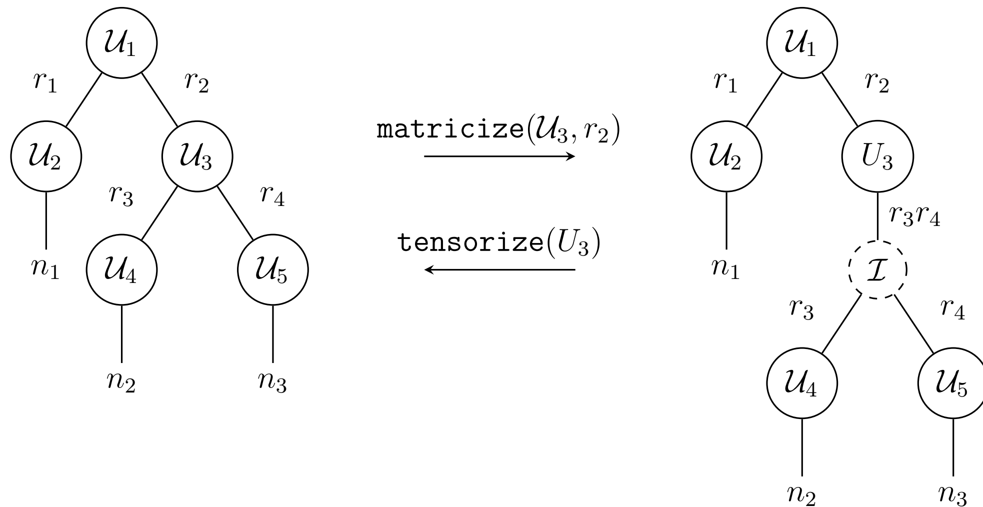


Figure 4.1 – Illustration of the `matricize` and `tensorize` kernels.

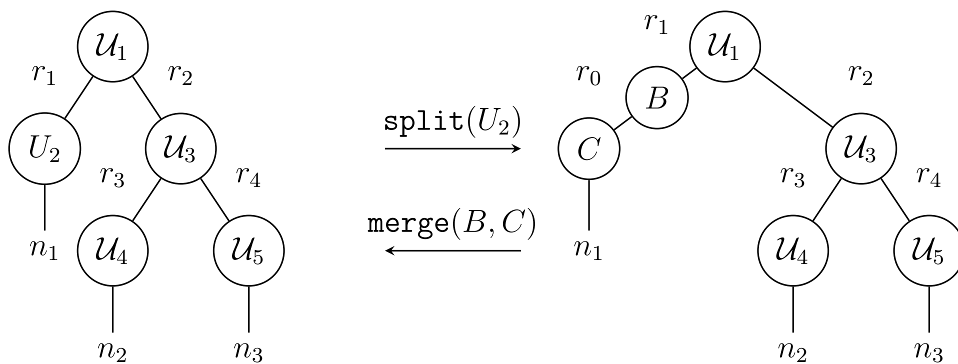


Figure 4.2 – Illustration of the `split` and `merge` kernels.

4.2.3 . α-normalized networks

Bounding the error introduced by each of the kernels in terms of the local node to which they are applied is not sufficient to obtain an error bound in terms of the global network.

Definition 4.1. *Given two nodes $\mathcal{U}_i$, $\mathcal{U}_j$ of a network $\boldsymbol{\mathcal{X}}$, $\mathcal{U}_i$ is said to be α_i-normalized in the direction of $\mathcal{U}_j$ if there exists a constant $\alpha_i > 0$ such that the matricization U_i of $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$ satisfies, for any matrix M of compatible dimensions, $\|U_i M\| \le \alpha_i \|M\|$.*

Definition 4.2. *Consider a network $\boldsymbol{\mathcal{X}}$ composed of n nodes $\mathcal{U}_1, \dots, \mathcal{U}_n$. $\boldsymbol{\mathcal{X}}$ is said to be α-normalized with respect to a node $\mathcal{U}_j$ if there exists a vector of constants $\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{R}^n$ such that $\|\mathcal{U}_j\| \le \alpha_j \|\boldsymbol{\mathcal{X}}\|$ and all the other nodes $\mathcal{U}_i$, $i \neq j$, are α_i-normalized in the direction of $\mathcal{U}_j$.*

This abstract definition reflects the fact that in several contexts the norm of the network $\boldsymbol{\mathcal{X}}$ is tightly concentrated in a single node $\mathcal{U}_j$, while the remaining nodes satisfy some Lipschitz-like condition meaning that their contraction approximately preserves the norm. In particular, there are two important cases where the network $\boldsymbol{\mathcal{X}}$ is α-normalized with small α_i values.

The first case is when the matricization of any node $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$ is semi-orthogonal. Indeed, in this case, $\|\boldsymbol{\mathcal{X}}\| = \|\mathcal{U}_j\|$ and the network is e-normalized where e is the vector of all ones, that is, $\alpha_i = 1$ for $i = 1\colon n$. Note that in this case, e-normality is equivalent to other definitions discussed in the literature, under the name of r-orthogonality [60, Definition 24] or t-frame [14, Definition 3.5].

The second case is when the network is obtained as the sum of two e-normalized networks. Indeed, in this case, the nodes with at least two inner dimensions remain semi-orthogonal. The other nodes are matrices since they are of degree two : one inner dimension and one outer dimension (since any node can be reshaped to only have one outer dimension). These nodes become the concatenation of two semi-orthogonal matrices. Let $U = [U_1 \; U_2]$ be such a node, where U_1 and U_2 are semi-orthogonal. Then we can prove that $\|UM\| \le \sqrt{2}\|M\|$ for any matrix M of compatible dimensions. Indeed, denoting $M = \begin{bmatrix} M_1 \\ M_2 \end{bmatrix}$ the partition of M corresponding to the partition of U, we have

$$\|UM\| = \|U_1 M_1 + U_2 M_2\| \le \|U_1 M_1\| + \|U_2 M_2\| = \|M_1\| + \|M_2\|. \tag{4.4}$$

Thus

$$\|UM\|^2 \le (\|M_1\| + \|M_2\|)^2 = \|M_1\|^2 + \|M_2\|^2 + 2\|M_1\|\|M_2\| = \|M\|^2 + 2\|M_1\|\|M_2\|. \tag{4.5}$$

The term $\|M_1\|\|M_2\| = \|M_1\|\sqrt{\|M\|^2 - \|M_1\|^2}$ is maximized when $\|M_1\|^2 = \|M\|^2/2$, in which case it attains its maximum value

$$\frac{\|M\|}{\sqrt{2}} \sqrt{\frac{\|M\|^2}{2}} = \frac{\|M\|^2}{2}. \tag{4.6}$$

Hence, overall

$$\|UM\|^2 \leq \|M\|^2 + 2\|M_1\|\|M_2\| \leq \|M\|^2 + 2\frac{\|M\|^2}{2} = 2\|M\|^2 \tag{4.7}$$

which proves that $\|UM\| \leq \sqrt{2}\|M\|$. In conclusion the sum of two e-normalized networks is an α-normalized network where $\alpha_i = \sqrt{2}$ for the matrix nodes and $\alpha_i = 1$ for the nodes of higher order.

We can derive the following useful properties of normalized networks.

Lemma 4.1. *If a network $\boldsymbol{\mathcal{X}}$ with n nodes is α-normalized with respect to a node $\mathcal{U}_j$, then*

$$\|\boldsymbol{\mathcal{X}}\| \leq \prod_{i=1,\, i\neq j}^{n} \alpha_i \|\mathcal{U}_j\|. \tag{4.8}$$

Proof. The proof is by induction on the number of nodes n of $\boldsymbol{\mathcal{X}}$. If $n = 1$ then $\mathcal{U}_j$ is the only node of $\boldsymbol{\mathcal{X}}$, so that $\|\boldsymbol{\mathcal{X}}\| = \|\mathcal{U}_j\|$ and thus (4.8) holds (with the convention that empty products are equal to 1). Assume (4.8) holds for any network with $n-1$ nodes. Then let $\mathcal{U}_i$ be any node adjacent to $\mathcal{U}_j$ and define $\boldsymbol{\mathcal{Y}}$ the network which is obtained from $\boldsymbol{\mathcal{X}}$ by replacing the nodes $\mathcal{U}_i$ and $\mathcal{U}_j$ by their contraction $\mathcal{U}_i\mathcal{U}_j$. This network has $n-1$ nodes and it satisfies $\|\boldsymbol{\mathcal{Y}}\| = \|\boldsymbol{\mathcal{X}}\|$. By induction, we therefore have

$$\|\boldsymbol{\mathcal{X}}\| = \|\boldsymbol{\mathcal{Y}}\| \leq \prod_{k=1,\, k\neq i,j}^{n} \alpha_k \|\mathcal{U}_i\mathcal{U}_j\|. \tag{4.9}$$

Since $\mathcal{U}_i$ is α_i-normalized in the direction of $\mathcal{U}_j$ we have $\|\mathcal{U}_i\mathcal{U}_j\| \leq \alpha_i\|\mathcal{U}_j\|$, which concludes the proof. □

Lemma 4.2. *If a network $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to a node $\mathcal{U}_j$, then for any node $\mathcal{U}_i$, $i \neq j$, we have*

$$\|\mathcal{U}_i\| \leq \alpha_i\sqrt{r_i}, \tag{4.10}$$

where r_i is the dimension of $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$.

Proof. Let U_i be the matricization of $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$; U_i is a matrix with r_i columns. We have

$$\|\mathcal{U}_i\| = \|U_i\| = \|U_i I_{r_i}\| \leq \alpha_i\|I_{r_i}\| = \alpha_i\sqrt{r_i}, \tag{4.11}$$

where I_{r_i} is the $r_i \times r_i$ identity matrix. □

4.3 . Stability analysis

In this section, we are interested in analyzing the effect of a network $\boldsymbol{\mathcal{X}}$ undergoing a sequence of inexact operations in finite precision arithmetic.

Denoting $\widehat{\boldsymbol{\mathcal{X}}}$ the resulting network after such a sequence of operations, we ask how different is $\widehat{\boldsymbol{\mathcal{X}}}$ from $\boldsymbol{\mathcal{X}}$? To answer this question, we must first define a measure of distance between tree tensor networks. The natural definition is to measure the norm of the difference of the tensors that these networks represent, that is :

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}.\mathtt{full}() - \widehat{\boldsymbol{\mathcal{X}}}.\mathtt{full}()\| = \|\mathcal{X} - \widehat{\mathcal{X}}\|. \tag{4.12}$$

In order to upper bound $\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\|$, we divide our analysis into three parts. First, in Section 4.3.1, we make some assumptions on the accuracy of the four kernels. Then, in Section 4.3.2, we analyze how a single local operation affects the accuracy of the global network. Finally, in Section 4.3.3, we show that the global error resulting from a sequence of such operations is bounded by the sum of the local errors and conclude our analysis.

4.3.1 . Model

First, since the `matricize` and `tensorize` kernels only reshape data without performing any numerical computations, it is natural to assume that they do not incur any error. In other words, we have

$$\mathtt{tensorize}(\mathtt{matricize}(\mathcal{A}, r)) = \mathcal{A}. \tag{4.13}$$

Consider now three matrices A, B, and C such that $A = BC$. Let $\widehat{B}\widehat{C}$ be the decomposition computed by the $\mathtt{split}(A)$ kernel. We assume that it satisfies,

$$\widehat{B}\widehat{C} = \mathtt{split}(A) = A + E, \quad \|E\| \le c\varepsilon\|A\|, \tag{4.14}$$

where c is a constant depending only on the dimensions of the matrices. This amounts to assuming that the decomposition computed by `split` is backward stable with respect to some precision parameter ε. This assumption is certainly satisfied for standard matrix decompositions (QR, SVD, ...) [26] that are typically used in the computations of interest.

Finally, consider three tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ such that $\mathcal{A} = \mathcal{B}\mathcal{C}$ where the product $\mathcal{B}\mathcal{C}$ corresponds to their contraction along a common dimension, and let $\widehat{\mathcal{A}}$ be the contraction computed by the $\mathtt{merge}(\mathcal{B}, \mathcal{C})$ kernel. We assume that it satisfies

$$\widehat{\mathcal{A}} = \mathtt{merge}(\mathcal{B}, \mathcal{C}) = \mathcal{B}\mathcal{C} + \mathcal{E}, \quad \|\mathcal{E}\| \le c\varepsilon\|\mathcal{B}\|\|\mathcal{C}\|. \tag{4.15}$$

This assumption is certainly satisfied if A, B, and C are matrices and A is computed via a standard matrix-matrix product [26, (3.13)]. It is not hard to see that the assumption is also satisfied for tensors if the contraction is performed by matricizing $\mathcal{B}$ and $\mathcal{C}$ along their common dimension and computing their product via a standard matrix-matrix product.

4.3.2 . Local errors

Let us now consider two networks $\boldsymbol{\mathcal{X}}$ and $\widehat{\boldsymbol{\mathcal{X}}}$, where $\widehat{\boldsymbol{\mathcal{X}}}$ is obtained by applying a single instance of one of the four kernels of our framework. Our goal is to bound $\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\|$. In fact, we only need to focus on the `split` and `merge` kernels since, as mentioned in the previous section, the `matricize` and `tensorize` kernels do not incur any error and thus $\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = 0$.

Let us first consider the case where $\widehat{\boldsymbol{\mathcal{X}}}$ is obtained by applying `split` on a given node $\mathcal{U}_k \equiv A$ of $\boldsymbol{\mathcal{X}}$. We define $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ the partial networks corresponding to all nodes along either dimension of A, so that $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}_1 A \boldsymbol{\mathcal{X}}_2$. Then by (4.14) we have

$$\widehat{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{X}}_1 \widehat{B}\widehat{C} \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1 (A+E) \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2 \tag{4.16}$$

and so

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2\| \le \|\boldsymbol{\mathcal{X}}_1\|\|E\|\|\boldsymbol{\mathcal{X}}_2\| \le c\varepsilon \|\boldsymbol{\mathcal{X}}_1\|\|A\|\|\boldsymbol{\mathcal{X}}_2\|. \tag{4.17}$$

This shows that even a single operation can lead to instability if $\|\boldsymbol{\mathcal{X}}_1\|\|A\|\|\boldsymbol{\mathcal{X}}_2\| \gg \|\boldsymbol{\mathcal{X}}\|$.

Stability can fortunately be guaranteed when the network is α-normalized as defined in Definition 4.2. Indeed, assume that $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to some node $\mathcal{U}_j$, and let n be the number of nodes of $\boldsymbol{\mathcal{X}}$.

If $j = k$ (that is, $\mathcal{U}_j = A$), then the network $\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2$ is β-normalized with respect to node E where $\beta_i = \alpha_i$ for all $i \ne j$ (the value of β_j is irrelevant). Therefore, by Lemma 4.1 we obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2\| \le \prod_{i=1,\, i\ne j}^{n} \alpha_i \|E\| \le \prod_{i=1}^{n} \alpha_i c\varepsilon \|\boldsymbol{\mathcal{X}}\| \tag{4.18}$$

since $\|E\| \le c\varepsilon\|A\| \le c\varepsilon\alpha_j\|\boldsymbol{\mathcal{X}}\|$.

If $k \ne j$ (that is, $\mathcal{U}_j \ne A$), assume that $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_2$ (the case $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_1$ is analogous). Then the network $\boldsymbol{\mathcal{X}}_1 E$ is β-normalized with respect to node E with $\beta_i = \alpha_i$ for all nodes $i \in \boldsymbol{\mathcal{X}}_1$ (the value of β_k is irrelevant). Therefore, by Lemma 4.1 we have

$$\|\boldsymbol{\mathcal{X}}_1 E\| \le \prod_{i\in\boldsymbol{\mathcal{X}}_1} \alpha_i \|E\| \le \prod_{i\in\boldsymbol{\mathcal{X}}_1} \alpha_i c\varepsilon \|A\| \le \prod_{i\in\boldsymbol{\mathcal{X}}_1} \alpha_i \alpha_k \sqrt{r_k} c\varepsilon \tag{4.19}$$

where r_k is the dimension of $A = \mathcal{U}_k$ in the direction of $\mathcal{U}_j$ by Lemma 4.2. Moreover, the network $\boldsymbol{\mathcal{X}}_2$ is $\bar{\alpha}$-normalized where $\bar{\alpha}$ is a subset of α restricted to the nodes in $\boldsymbol{\mathcal{X}}_2$, and therefore by Lemma 4.1

$$\|\boldsymbol{\mathcal{X}}_2\| \le \prod_{i\in\boldsymbol{\mathcal{X}}_2,\, i\ne j} \alpha_i \|\mathcal{U}_j\| \le \prod_{i\in\boldsymbol{\mathcal{X}}_2} \alpha_i \|\boldsymbol{\mathcal{X}}\|. \tag{4.20}$$

By combining (4.19) and (4.20) we finally obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2\| \leq \|\boldsymbol{\mathcal{X}}_1 E\| \|\boldsymbol{\mathcal{X}}_2\| \leq \prod_{i=1}^{n} \alpha_i \sqrt{r_k} c\varepsilon \|\boldsymbol{\mathcal{X}}\|. \tag{4.21}$$

For the `merge` kernel, the analysis is very similar. Let $\texttt{merge}(\mathcal{B}, \mathcal{C})$ be applied to $\boldsymbol{\mathcal{X}}$ where $\mathcal{B} \equiv \mathcal{U}_k$ and $\mathcal{C} \equiv \mathcal{U}_\ell$. Defining $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ such that $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}_1 \mathcal{B}\mathcal{C} \boldsymbol{\mathcal{X}}_2$, by (4.15) we have

$$\widehat{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{X}}_1 \widehat{\mathcal{A}} \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1 (\mathcal{B}\mathcal{C} + \mathcal{E}) \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2. \tag{4.22}$$

The goal is thus to bound $\|\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2\|$ assuming that $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to some node $\mathcal{U}_j$.

If $j = \ell$ (that is, $\mathcal{U}_j = \mathcal{C}$) then $\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2$ is β-normalized with $\beta_i = \alpha_i$ for all $i \neq j, k$. By Lemma 4.1 we have

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2\| \tag{4.23}$$

$$\leq \prod_{i=1,\, i\neq j,k}^{n} \alpha_i \|\mathcal{E}\| \tag{4.24}$$

$$\leq \prod_{i=1,\, i\neq j,k}^{n} \alpha_i c\varepsilon \|\mathcal{U}_k\| \|\mathcal{U}_j\| \tag{4.25}$$

$$\leq \prod_{i=1,\, i\neq k}^{n} \alpha_i c\varepsilon \|\mathcal{U}_k\| \|\boldsymbol{\mathcal{X}}\| \tag{4.26}$$

$$\leq \prod_{i=1}^{n} \alpha_i \sqrt{r_k} c\varepsilon \|\boldsymbol{\mathcal{X}}\| \tag{4.27}$$

since $\|\mathcal{U}_k\| \leq \alpha_k \sqrt{r_k}$ by Lemma 4.2.

If $j = k$ (that is, $\mathcal{U}_j = \mathcal{B}$) then we similarly have

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq \prod_{i=1}^{n} \alpha_i \sqrt{r_\ell} c\varepsilon \|\boldsymbol{\mathcal{X}}\|. \tag{4.28}$$

Finally, if $j \neq k, \ell$ (that is, $\mathcal{U}_j$ is neither $\mathcal{B}$ nor $\mathcal{C}$), assume that $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_2$ (the case $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_1$ is analogous). Then $\boldsymbol{\mathcal{X}}_1 \mathcal{E}$ is β-normalized with respect to node $\mathcal{E}$ with $\beta_i = \alpha_i$ for all nodes $i \in \boldsymbol{\mathcal{X}}_1$. Therefore, by Lemma 4.1 we have

$$\|\boldsymbol{\mathcal{X}}_1 \mathcal{E}\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_1} \alpha_i \|\mathcal{E}\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_1} \alpha_i c\varepsilon \|\mathcal{U}_k\| \|\mathcal{U}_\ell\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_1} \alpha_i \alpha_k \alpha_\ell \sqrt{r_k r_\ell} c\varepsilon \tag{4.29}$$

by Lemma 4.2. Moreover, by the same argument as before we also have

$$\|\boldsymbol{\mathcal{X}}_2\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_2} \alpha_i \|\boldsymbol{\mathcal{X}}\|. \tag{4.30}$$

By combining (4.29) and (4.30) we finally obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2\| \leq \prod_{i=1}^{n} \alpha_i \sqrt{r_k r_\ell} c\varepsilon \|\boldsymbol{\mathcal{X}}\|. \tag{4.31}$$

We have thus bounded the error introduced by $\texttt{split}(A)$ and $\texttt{merge}(\mathcal{B}, \mathcal{C})$ assuming that $\boldsymbol{\mathcal{X}}$ is α-normalized. The bounds (4.21) and (4.31) are proportional to $\prod_{i=1}^{n} \alpha_i$, which must therefore be small to guarantee stability. They also depend on at most $\sqrt{r_k r_\ell}$, which is certainly bounded by the largest inner dimension r of the network.

We summarize the conclusions of this local error analysis in the following theorem.

Theorem 4.1. *Let $\widehat{\boldsymbol{\mathcal{X}}}$ be obtained by applying either $\texttt{split}(A)$ or $\texttt{merge}(\mathcal{B}, \mathcal{C})$ to $\boldsymbol{\mathcal{X}}$. Under the assumptions* (4.14) *and* (4.15)*, and assuming that $\boldsymbol{\mathcal{X}}$ is α-normalized as defined in Definition 4.2, we have*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq rc \prod_{i=1}^{n} \alpha_i \varepsilon \|\boldsymbol{\mathcal{X}}\|, \tag{4.32}$$

where r is the largest inner dimension of $\boldsymbol{\mathcal{X}}$ and c is the constant in (4.14) *and* (4.15)*.*

4.3.3 . Global error

We now consider a network $\widehat{\boldsymbol{\mathcal{X}}}$ which is the result of a sequence of p local operations ($\texttt{split}$ or $\texttt{merge}$) applied to $\boldsymbol{\mathcal{X}}$. Let $\boldsymbol{\mathcal{X}}_k$ be the network after having performed k operations, so that $\boldsymbol{\mathcal{X}}_0 = \boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{X}}_p = \widehat{\boldsymbol{\mathcal{X}}}$.

One technical difficulty is that the local errors incurred by each operation lead to second-order error terms : the error incurred by the kth operation depends on the errors incurred by the previous $k-1$ operations. This effect, however, only introduces an $O(\varepsilon^2)$ error term which is not particularly significant but quite tedious to compute precisely. Therefore, in the following we will not track explicitly these $O(\varepsilon^2)$ terms.

To bound the global error $\widehat{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{X}}$, we observe that it is bounded by the sum of the local errors. Indeed, we have the telescopic sum

$$\begin{aligned}
\|\widehat{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{X}}\| &= \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_p\| && (4.33)\\
&= \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_1 + \boldsymbol{\mathcal{X}}_1 - \boldsymbol{\mathcal{X}}_p\| && (4.34)\\
&= \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_1 + \boldsymbol{\mathcal{X}}_1 - \cdots + \boldsymbol{\mathcal{X}}_{p-1} - \boldsymbol{\mathcal{X}}_p\| && (4.35)\\
&\leq \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_1\| + \cdots + \|\boldsymbol{\mathcal{X}}_{p-1} - \boldsymbol{\mathcal{X}}_p\| && (4.36)\\
&= \sum_{k=0}^{p-1} \|\boldsymbol{\mathcal{X}}_k - \boldsymbol{\mathcal{X}}_{k+1}\|. && (4.37)
\end{aligned}$$

Let n_k be the number of nodes of $\boldsymbol{\mathcal{X}}_k$. Assume that at each step k the network $\boldsymbol{\mathcal{X}}_k$ is α_k-normalized, where $\alpha^{(k)} \in \mathbb{R}^{n_k}$ is a vector with elements $\alpha_i^{(k)}$, $i = 1\colon n_k$. Define

$$\zeta_k = r_k c \prod_{i=1}^{n_k} \alpha_i^{(k)}, \tag{4.38}$$

where r_k is the largest inner dimension of $\boldsymbol{\mathcal{X}}_k$. Then by Theorem 4.1 we have

$$\|\boldsymbol{\mathcal{X}}_k - \boldsymbol{\mathcal{X}}_{k+1}\| \leq \zeta_k \varepsilon \|\boldsymbol{\mathcal{X}}_k\| = \zeta_k \varepsilon \|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2). \tag{4.39}$$

We therefore obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \sum_{k=0}^{p-1} \|\boldsymbol{\mathcal{X}}_k - \boldsymbol{\mathcal{X}}_{k+1}\| \tag{4.40}$$

$$\leq \sum_{k=0}^{p-1} \zeta_k \varepsilon \|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{4.41}$$

$$\tag{4.42}$$

We summarize our conclusions in the following theorem, which determines sufficient conditions for an abstract computation on $\boldsymbol{\mathcal{X}}$ to be stable.

Theorem 4.2. *Let $\widehat{\boldsymbol{\mathcal{X}}}$ be obtained by applying a sequence of p operations of the form $\mathtt{split}(A)$ or $\mathtt{merge}(\mathcal{B}, \mathcal{C})$ to $\boldsymbol{\mathcal{X}}$. Under the assumptions* (4.14) *and* (4.15)*, and assuming that at each step k of the computation, the intermediate network $\boldsymbol{\mathcal{X}}_k$ is $\alpha^{(k)}$-normalized as defined in Definition 4.2 with $\alpha^{(k)} \in \mathbb{R}^{n_k}$, we have*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon \|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{4.43}$$

with

$$s = \sum_{k=0}^{p-1} c r_k \prod_{i=1}^{n_k} \alpha_i^{(k)}, \tag{4.44}$$

where r_k is the largest inner dimension of $\boldsymbol{\mathcal{X}}_k$ and c is the constant in (4.14) *and* (4.15)*.*

The bound (4.43) can be simplified to

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \lesssim p r \alpha^n c \varepsilon \|\boldsymbol{\mathcal{X}}\|, \tag{4.45}$$

where $r = \max_k r_k$, $n = \max_k n_k$, and $\alpha = \max_{i,k} \alpha_i^{(k)}$. In other words, Theorem 4.2 shows that the error grows linearly with the kernel error $c\varepsilon$, the largest inner dimension r of the network, the number of operations p, and the term α^n, which is exponential in the number of nodes n. Therefore, if the kernels are stable (small c), the main condition for the overall network computation to also be stable is that α is small.

As discussed in Section 4.3.1, there are important practical cases where we know the network to be α-normalized for small values of α_i. In these cases, stability is thus guaranteed. We discuss concrete examples in the next section (Section 4.4). We also develop in the section after that (Section 4.5) a rounding algorithm that exploits the conclusion of our analysis to guarantee stability.

4.4 . Examples of stable tensor computations

Our analysis in the previous section has identified conditions for an abstract computation in our framework to be stable. Here, we aim to provide some concrete examples of tensor computations that are proven stable by our analysis. To do so, we need to achieve two things : 1) show that the computation of interest can be expressed in our framework ; and 2) show that throughout the computation the network remains α-normalized (ideally with small values of α).

4.4.1 . `full`

We begin with the simple example of the `full` operator, which contracts a tree tensor network into the full tensor it represents as illustrated in Figure 4.3 (going from left to right).

This operation can simply be expressed as a sequence of $\texttt{merge}(\mathcal{U}, \mathcal{V})$ of adjacent nodes $\mathcal{U}$ and $\mathcal{V}$, until only one node $\mathcal{X}$ remains, as shown in Algorithm 4.1.

Algorithm 4.1 $\texttt{full}()$

1 : Choose a node $\mathcal{U}$.
2 : **while** $\mathcal{U}$ is not the only node left **do**
3 :   Let $\mathcal{V}$ be a node adjacent to $\mathcal{U}$.
4 : $\mathcal{U} = \texttt{merge}(\mathcal{U}, \mathcal{V})$
5 : **end while**

Mathematically, the contractions may be performed in any order since all orders lead to the same result $\mathcal{X}$. However, some orders may computationally more efficient depending on the dimensions. Moreover, in finite precision arithmetic, different orders are not numerically equivalent. The following result provides an error bound for α-normalized networks.

Corollary 4.1. *Let Algorithm 4.1 be applied to a network $\boldsymbol{\mathcal{X}}$ α-normalized with respect to node $\mathcal{U}_j$. If $\mathcal{U}_j$ is chosen as node $\mathcal{U}$ on line 1, then the computed $\widehat{\mathcal{X}}$ satisfies*

$$\|\mathcal{X} - \widehat{\mathcal{X}}\| \leq s\varepsilon \|\boldsymbol{\mathcal{X}}\| \tag{4.46}$$

with $s = (n-1)cr \prod_{i=1}^{n} \alpha_i$, where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, r is its largest inner dimension, and c is the constant in (4.14) *and* (4.15)*.*

Proof. Algorithm 4.1 consists of $n-1$ `merge` operations where the intermediate networks $\boldsymbol{\mathcal{X}}_k$ at each step k are progressively more and more contracted : $\boldsymbol{\mathcal{X}}_k$ has $n_k = n-k$ nodes and its largest inner dimension r_k is certainly bounded by the largest inner dimension r of the original network $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}_0$. Assume (without loss of generality since the nodes can be relabelled) that $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to $\mathcal{U}_n$ and that the other nodes are merged with $\mathcal{U}_n$ in the order $\mathcal{U}_{n-1}, \dots \mathcal{U}_1$. Then at step k $\boldsymbol{\mathcal{X}}_k$ is $\beta^{(k)}$-normalized with respect to $\mathcal{U}_n$ where $\beta^{(k)} \in \mathbb{R}^{n_k}$ satisfies $\beta_i^{(k)} = \alpha_i$ for $i < n_k$ and $\beta_{n_k}^{(k)} = \prod_{\ell=n_k}^{n} \alpha_\ell + O(\varepsilon)$. The $O(\varepsilon)$ term accounts for the inexactness of the operations which may slightly alter the α constants, and which will only contribute to the $O(\varepsilon^2)$ term in the final error bound. As a result at each step k we have the relation $\prod_{i=1}^{n_k} \beta_i^{(k)} = \prod_{i=1}^{n} \alpha_i + O(\varepsilon)$. Therefore, by Theorem 4.2 we obtain the bound (4.43) with

$$s = \sum_{k=0}^{n-2} c r_k \prod_{i=1}^{n_k} \beta_i^{(k)} + O(\varepsilon^2) \leq \sum_{k=0}^{n-2} c r \prod_{i=1}^{n} \alpha_i + O(\varepsilon^2). \tag{4.47}$$

□

Corollary 4.1 shows that if a network $\boldsymbol{\mathcal{X}}$ is α-normalized with small values of α_i, it can be stably contracted into a full tensor $\mathcal{X}$. Note that the assumption that $\mathcal{U}_j$ is chosen as $\mathcal{U}$ on line 1 is required. Indeed, given two nodes $\mathcal{U}_i$ and $\mathcal{U}_k$ that are respectively α_i- and α_k-normalized in the direction of $\mathcal{U}_j$, we have not assumed that their contraction $\mathcal{U}_i\mathcal{U}_k$ is $\alpha_i\alpha_k$-normalized. If this were the case, then the error bound (4.46) would hold for any order of contractions. Importantly, this is the case when the nodes are semi-orthogonal ($\alpha_i = \alpha_k = 1$), because the product of two semi-orthogonal matrices in the same direction is also semi-orthogonal.

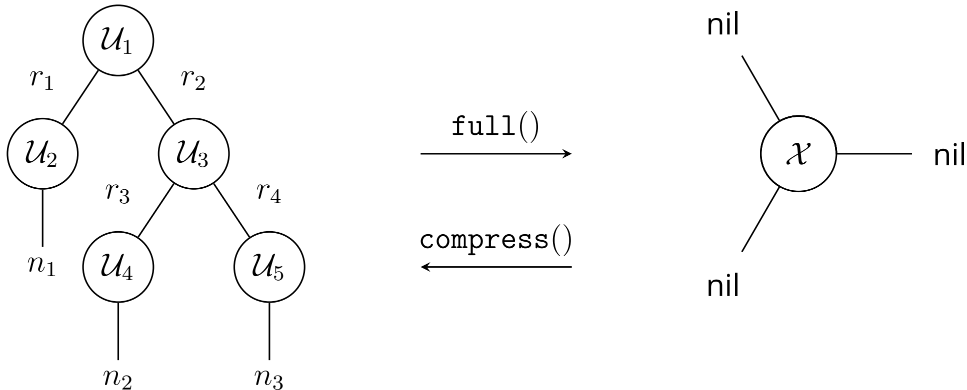


Figure 4.3 – Illustration of `full` and `compress`.

4.4.2 . `compress`

We can also express the inverse operation of the `full` operation, denoted as `compress` : given a full tensor $\mathcal{X}$, we seek to compress it into a tree tensor network $\boldsymbol{\mathcal{X}}$ with a prescribed structure, as illustrated in Figure 4.3 (going from right to left).

Algorithm 4.2 $\texttt{compress}()$

1 : **for** each node $\mathcal{U}_i = \mathcal{U}_1, \ldots \mathcal{U}_{n-1}$ of the prescribed structure **do**
2 : $\quad X = \texttt{matricize}(\mathcal{X}, r)$ for a suitable dimension r
3 : $\quad UY = \texttt{split}(X)$ {Truncated SVD ; U is semi-orthogonal}
4 : $\quad \mathcal{U}_i = \texttt{tensorize}(U)$
5 : $\quad \mathcal{X} = \texttt{tensorize}(Y)$
6 : **end for**
7 : $\mathcal{U}_n = \mathcal{X}$ {The remaining $\mathcal{X}$ is the root node.}

To minimize the inner dimensions of the network, we perform successive Low-rank approximations (LRA)s, which are computed via truncated SVDs. Thus, we can express `compress` as a sequence of `matricize`, `split`, and `tensorize` kernels, as described in Algorithm 4.2. Note that Algorithm 4.2 computes the nodes of the network in the order $\mathcal{U}_1$, ..., $\mathcal{U}_n$; this is done without loss of generality since the nodes can be relabelled. Note also that the "suitable dimension r" on line 2 strongly depends on the network structure. For example, for a Tucker tensor, the $n-1=d$ matricizations will be done along the d outer dimensions of the original full tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$. For more general tree networks, the leaf nodes will similarly lead to matricizations along the outer dimensions ; the nodes in the upper levels will lead to matricizations along the inner dimensions computed in the previous levels.

Importantly, the truncated SVD yields a semi-orthogonal factor U and a non-orthogonal factor Y that we recursively compress. Therefore, Algorithm 4.2 yields a network $\boldsymbol{\mathcal{X}}$ where all nodes except the root are semi-orthogonal in the direction of the root. Therefore, $\boldsymbol{\mathcal{X}}$ is e-normalized where e is the vector of all ones. This readily yields the following error bound.

Corollary 4.2. *Let Algorithm 4.2 be applied to a tensor $\mathcal{X}$ to compress it into a prescribed network structure $\boldsymbol{\mathcal{X}}$. The computed $\widehat{\boldsymbol{\mathcal{X}}}$ satisfies*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq (n-1)cr\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2), \tag{4.48}$$

where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, r is its largest inner dimension, and c is the constant in (4.14) *and* (4.15)*.*

Proof. Algorithm 4.2 consists of $n-1$ truncated SVD (`split`) operations where the networks $\boldsymbol{\mathcal{X}}_k$ at each step k are progressively more and more compressed : $\boldsymbol{\mathcal{X}}_k$ has $k+1$ nodes and its largest inner dimension r_k is certainly bounded by the largest inner dimension r of the final network $\boldsymbol{\mathcal{X}}_{n-1} = \boldsymbol{\mathcal{X}}$. Since

each node $\mathcal{U}_k$ computed at step k is semi-orthogonal in the direction of the root node $\mathcal{X}$, the network $\boldsymbol{\mathcal{X}}_k$ at each step k is β-normalized with respect to $\mathcal{X}$ (the only non-orthogonal node), where $\beta = e_{k+1} + O(\varepsilon)$ is equal to the vector of all ones of length $k+1$, up to $O(\varepsilon)$ inexactness due to the previous operations. Therefore, by Theorem 4.2 we have the bound (4.43) with $s = (n-1)cr$. □

4.4.3. `orthog`

Finally, we consider the `orthog` operation, which consists in orthogonalizing all nodes of a network except its root. As described in Algorithm 4.3, this can be achieved by performing a QR factorization of each node (line 7), matricized in the direction of its parent (line 6); the node is then replaced by the tensorized semi-orthogonal factor Q (line 8), while the non-orthogonal factor R is merged with the parent node (line 9). At the end of this process, all the nodes will be semi-orthogonal in the direction of the root $\mathcal{R}$, which will be the only node that is not semi-orthogonal and which satisfies $\|\mathcal{R}\| = \|\boldsymbol{\mathcal{X}}\|$.

Algorithm 4.3 $\texttt{orthog}()$

1 : **for** each node $\mathcal{U}$ from leaves to root **do**
2 :   **if** $\mathcal{U}$ is the root **then**
3 :   **return**
4 :   **else**
5 : $\mathcal{P} = \texttt{parent}(\mathcal{U})$
6 : $U = \texttt{matricize}(\mathcal{U}, \mathcal{P})$ {Matricize in the direction of the parent}
7 : $QR = \texttt{split}(U)$ {QR factorization ; Q is semi-orthogonal}
8 : $\texttt{tensorize}(Q)$
9 : $\texttt{merge}(R, \mathcal{P})$
10 :   **end if**
11 : **end for**

The following result provides an error bound for orthogonalizing α-normalized networks.

Corollary 4.3. *Let Algorithm 4.3 be applied to a network $\boldsymbol{\mathcal{X}}$ α-normalized with respect to its root. Then the computed $\widehat{\boldsymbol{\mathcal{X}}}$ satisfies*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{4.49}$$

with $s = 2(n-1)cr\prod_{i=1}^{n}\alpha_i$, where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, r is its largest inner dimension, and c is the constant in (4.14) *and* (4.15)*.*

Proof. Algorithm 4.3 consists of $2(n-1)$ operations : $n-1$ QR factorizations (`split`) and $n-1$ contractions (`merge`). Let us denote $\boldsymbol{\mathcal{X}}_{2k}$ the network after k

`split` and k `merge`, and $\boldsymbol{\mathcal{X}}_{2k+1}$ the network after $k+1$ `split` and k `merge`, for $k = 0\colon n-1$. Then $\boldsymbol{\mathcal{X}}_{2k}$ has n nodes and $\boldsymbol{\mathcal{X}}_{2k+1}$ has $n+1$ nodes; the largest inner dimension of either of them is bounded by the largest inner dimension r of the original network. $\boldsymbol{\mathcal{X}}_{2k}$ corresponds to the original network $\boldsymbol{\mathcal{X}}$ where the first k nodes have been orthogonalized; therefore it is β-normalized where $\beta \in \mathbb{R}^n = [e_k \; \alpha_{k+1\colon n}] + O(\varepsilon)$, that is, the first k coefficients of β are 1 and the rest are equal to the remaining coefficients of α, up to $O(\varepsilon)$ inexactness from the previous operations. Similarly, $\boldsymbol{\mathcal{X}}_{2k+1}$ is β-normalized where $\beta \in \mathbb{R}^{n+1} = [e_k \; \alpha_{k\colon n}] + O(\varepsilon)$. Thus, in either case $\prod_i \beta_i \leq \prod_i \alpha_i + O(\varepsilon)$. By Theorem 4.2 we thus obtain (4.49). □

`orthog` is particularly of interest when $\boldsymbol{\mathcal{X}}$ is obtained as the sum of two e-normalized networks. As explained in Section 4.2.3, in this case $\boldsymbol{\mathcal{X}}$ is α-normalized with $\alpha_i \leq \sqrt{2}$; we may use `orthog` to make the network e-normalized again. Corollary 4.3 shows that this can be done stably since $\alpha_i \leq \sqrt{2}$ is a small constant.

4.5 . A general stable rounding algorithm

Tensor rounding is a fundamental task that consists in finding the optimal dimensions for each edge of the network while satisfying a prescribed accuracy ε. In this section, we exploit the conclusions of our error analysis in Section 4.3 to design a tensor rounding algorithm, that is both general and stable. It is *general*, because it can handle any tree tensor network and only relies on the four kernels defined by our framework; it is *stable*, because it takes care of preserving the semi-orthogonality of all nodes except one throughout the computation.

We first describe the proposed algorithm in Section 4.5.1, and then discuss in Section 4.5.2 how it relates to existing tensor rounding algorithms.

4.5.1 . The proposed algorithm

Our rounding algorithm is outlined in Algorithm 4.4. It proceeds in two phases. The first phase (`orthog`) orthogonalizes all nodes of the network except its root as described and analyzed in Section 4.4.3. At the end of this first phase, the root of the network is therefore the only node that is not semi-orthogonal.

Algorithm 4.4 `round`

1: `orthog`$()$ (Algorithm 4.3)
2: Let $\mathcal{U}$ be the root (which is the only node not semi-orthogonal).
3: `truncate`$(\mathcal{U})$ (Algorithm 4.5)

The second phase (`truncate`) then proceeds from the root down to the leaves and is implemented through the recursive function `truncate` outlined in Algorithm 4.5. It is initially called on the root node $\mathcal{U}$, which is the only node that is not semi-orthogonal. Then it loops on each of its children $\mathcal{C}$ (line 2), and we tighten the dimension connecting $\mathcal{U}$ and $\mathcal{C}$ by computing a truncated SVD YV of the matricized $\mathcal{U}$ (line 5). The semi-orthogonal factor V is tensorized to replace $\mathcal{U}$ (line 6), while the non-orthogonal factor Y is merged with the children $\mathcal{C}$ (line 7), which newly becomes the only node of the network that is not semi-orthogonal. We can therefore recursively call `truncate` on $\mathcal{C}$ (line 8), which will tighten all the dimensions in the subtree rooted at $\mathcal{C}$. The recursion terminates when the child $\mathcal{C}$ is a leaf node (lines 14 to 17) : in this case, we compute a truncated SVD VY (line 15) where the semi-orthogonal factor V is on the left and is merged with $\mathcal{C}$ (line 16), whereas Y is tensorized (line 17) to replace $\mathcal{U}$ which therefore remains non-orthogonal. Once the recursion on $\mathcal{C}$ has terminated, all that remains to do is to transfer the non-orthogonality back to the parent node $\mathcal{U}$. This is achieved by performing a QR factorization (line 10) of $\mathcal{C}$ matricized in the direction of its parent $\mathcal{U}$ (line 9). The semi-orthogonal factor Q is tensorized (line 11) to replace $\mathcal{C}$ while the non-orthogonal factor R is merged (line 12) with the parent $\mathcal{U}$, which is therefore once again the only node that is not semi-orthogonal.

The key idea of this rounding algorithm is to concentrate the non-orthogonality in a single node of the network, which therefore carries the entire norm of the network, and this information is transferred through the network as the computation advances while keeping all the other nodes semi-orthogonal. In other words, the orthogonalization phase makes the network e-normalized, and then the network is maintained e-normalized throughout the entire truncation phase. We therefore have the following key result.

Corollary 4.4. *Let Algorithm 4.4 be applied to a network $\boldsymbol{\mathcal{X}}$ α-normalized with respect to its root. Then the computed $\widehat{\boldsymbol{\mathcal{X}}}$ satisfies*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{4.50}$$

with

$$s = cr\left(2(n-1)\prod_{i=1}^{n}\alpha_i + 4(n-1) - 2\ell\right),$$

where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, ℓ is its number of leaves, r is its largest inner dimension, and c is the constant in (4.14) *and* (4.15).

Proof. The result is a direct consequence of Corollary 4.3, which bounds the error for the orthogonalization phase, and of the fact that the network remains e-normalized throughout the truncation phase. The $2(n-1)\prod_{i=1}^{n}\alpha_i$ accounts for the orthogonalization phase and comes from (4.49). The truncation phase leads to an error pcr where p is the number of operations.

Algorithm 4.5 $\texttt{truncate}(\mathcal{U})$

1: {On input, $\mathcal{U}$ is the only node not semi-orthogonal}
2: **for** all children $\mathcal{C}$ of $\mathcal{U}$ **do**
3: $U = \texttt{matricize}(\mathcal{U},\mathcal{C})$ {Matricize $\mathcal{U}$ in the direction of its child}
4: **if** $\mathcal{C}$ is not a leaf **then**
5: $YV = \texttt{split}(U)$ {Truncated SVD; V is semi-orthogonal}
6: $\mathcal{U} = \texttt{tensorize}(V)$
7: $\mathcal{C} = \texttt{merge}(Y,\mathcal{C})$ {$\mathcal{C}$ is now the only node not semi-orthogonal}

8: $\texttt{truncate}(\mathcal{C})$ {Recursive call}
9: $C = \texttt{matricize}(\mathcal{C},\mathcal{U})$ {Matricize $\mathcal{C}$ in the direction of its parent}
10: $QR = \texttt{split}(C)$ {QR factorization}
11: $\mathcal{C} = \texttt{tensorize}(Q)$
12: $\mathcal{U} = \texttt{merge}(R,\mathcal{U})$
13: {$\mathcal{U}$ is back to being the only node not semi-orthogonal}
14: **else**
15: $VY = \texttt{split}(U)$ {Truncated SVD; V is semi-orthogonal}
16: $\mathcal{C} = \texttt{merge}(V,\mathcal{C})$
17: $\mathcal{U} = \texttt{tensorize}(Y)$ {$\mathcal{U}$ still is the only node not semi-orthogonal}
18: **end if**
19: **end for**

`truncate` requires four operations (two `split` and two `merge`) for each internal node (excluding the leaves and the root), and two operations (one `split` and one `merge`) for the leaves. Therefore, we have $p = 4(n-1-\ell) + 2\ell = 4(n-1) - 2\ell$. □

In particular, Corollary 4.4 shows that the sum of two e-normalized tensors can be stably rounded with Algorithm 4.4, since in this case $\alpha_i \leq \sqrt{2}$ (see Section 4.2.3).

4.5.2 . Comparison with existing tensor rounding algorithms

Various tensor rounding algorithms have been proposed in the literature, depending on the network topology.

For the TT format (which corresponds to a chain of nodes), a standard implementation of the algorithm is provided in [9, Alg .2]. The algorithm is divided in two steps. First, all the nodes are orthogonalized from right to left except for the leftmost one. The second step starts from this leftmost non-orthogonal node, tightens its inner dimension to the prescribed accuracy ε via a truncated SVD, and merges the non-orthogonal factor with the node to its right. The process is then recursively repeated on this second node until all inner dimensions have been tightened. It is not hard to see that our Algorithm 4.4, when applied to a tensor train topology, reduces precisely to this very algorithm. In particular, this proves that the tensor train rounding algorithm from [9] is stable in finite precision arithmetic.

For the Tucker format, to the best of our knowledge, no rounding algorithm is described in the literature. However, the high-order singular value decomposition (HOSVD) algorithm described in [33], which implements the `compress` kernel (computation of the Tucker tensor directly from the full tensor), can be used to round a given Tucker tensor as follows. First, we make all nodes semi-orthonormal except for the root (`orthog`); then, we use HOSVD to compute a Tucker approximation of the root (`compress`); finally, we contract the leaves of this Tucker root tensor with the leaves of the original tensor (`merge`) to obtain its tightened Tucker approximation. This method performs exactly the same operations as our `round` algorithm, but in a different order : our algorithm would interlace the truncations (`split`) and contractions (`merge`) along each successive dimension of the root, instead of first performing all the truncations in `compress` with HOSVD and then all the contractions. However, since these operations are independent the two orders produce numerically equivalent results, and so we conclude that using this HOSVD-based method is also stable.

A widely used algorithm is the Gram SVD, which was originally proposed for the HT format [14] (which corresponds to a complete binary tree topology) and notably implemented in the MATLAB Htucker toolbox [51]. Gram SVD has also been recently extended to the tensor train format [21]. For each

node, Gram SVD computes its Gramian, a small square matrix whose eigenvalue decomposition can be used to recover the desired truncated SVD of the node in exact arithmetic. This algorithm is quite efficient and is in particular very suitable for parallelization. However, in finite precision arithmetic, this algorithm is unstable due to the ill-conditioning of the Gramians : the error analysis in [25] proves that an accuracy of the order of the square root of the machine precision can be expected, which is quite limiting when considering the use of low precision arithmetics. We will perform a detailed experimental comparison between this algorithm and our Algorithm 4.4 in the next section.

Finally, we also mention the PhD thesis [61], which contains a rounding algorithm for general tree topologies [61, Alg. 6]. This algorithm shares some similarities with our Algorithm 4.4, but also some differences. In particular, it performs the truncation from leaves to root instead of from root to leaves. Nevertheless, this algorithm also has the property of preserving the semi-orthogonality of all nodes except one throughout the computation, and so based on our error analysis, we expect it to be numerically stable.

4.6 . Numerical experiments

We now present some numerical experiments to validate the numerical stability of our rounding algorithm, Algorithm 4.4, and compare its accuracy to the Gram SVD approach.

4.6.1 . Experimental setting

All experiments were performed with MATLAB R2019a.

We use a tensor $\boldsymbol{\mathcal{Z}}$ obtained as the sum of two fourth-order tensors $\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{100\times100\times100\times100}$, where the nodes of $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ are randomly generated with exponentially decaying elements. For these experiments, we use the HT format (binary tree network), although we also tested other network topologies and obtained similar results.

We evaluate the accuracy of the algorithms with the normwise relative error

$$\eta = \|\boldsymbol{\mathcal{Z}} - \widehat{\boldsymbol{\mathcal{Z}}}\|/\|\boldsymbol{\mathcal{Z}}\|, \tag{4.51}$$

where $\widehat{\boldsymbol{\mathcal{Z}}}$ is the computed rounded tensor.

We test the use of various floating-point arithmetics : double, single, and half precisions, which correspond to a machine precision of $u = 2^{-53} \approx 1 \times 10^{-16}$, $u = 2^{-24} = 6 \times 10^{-8}$, and $u = 2^{-11} = 5 \times 10^{-4}$, respectively. Furthermore, we simulate the use of lower precisions using the chop library [38].

For the orthonormalization operations (`split` without truncation), we use MATLAB's `qr` function, which implements Householder QR factorization. Since this is a stable operation, it satisfies assumption (4.14) with $\varepsilon = u$. For the truncated SVD operations (`split` with truncation), we use MATLAB's `svd` to

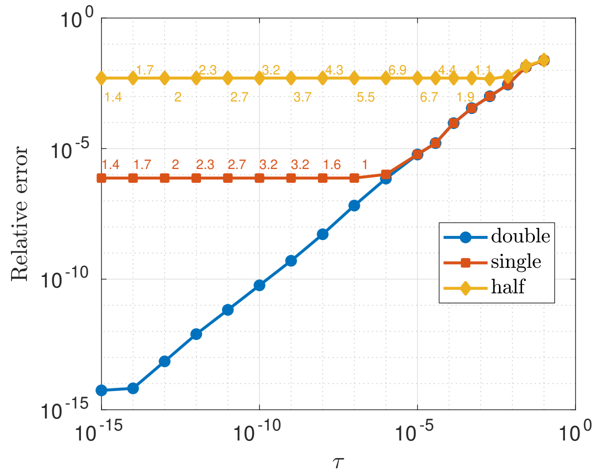


Figure 4.4 – Accuracy of Algorithm 4.4 depending on the truncation threshold τ and the floating-point precision u. The text labels next to the markers indicate the compression ratio between a given variant in lower precision and the reference compression obtained in double precision, for the same value of τ (if this ratio is equal to 1 we omit the label).

compute the full SVD and then truncate the decomposition based on the specified truncation threshold τ. Since the SVD is also stable, this operation satisfies assumption (4.14) with $\varepsilon = u + \tau$. Finally, for the contraction operations (`merge`), we simply use standard matrix-matrix products, and so assumption (4.15) is satisfied with $\varepsilon = u$. Overall, we therefore expect our `round` algorithm to achieve an accuracy of order $\varepsilon = u + \tau$, which we will validate in Section 4.6.2.

We will also compare with Gram SVD in Section 4.6.3. For these experiments, we use the MATLAB Htucker toolbox [51], with some modifications of our own for the purpose of the experiments (such as plugging the `chop` function in order to simulate low precision arithmetic).

4.6.2 . Validating the stability of Algorithm 4.4

In Figure 4.4 we evaluate the accuracy of Algorithm 4.4 for various values of the truncation threshold τ and for various precisions u. The numbered labels next to the markers indicate the ratio between the size of the tensor rounded at accuracy τ using double precision and the size of the tensor rounded for the same τ but using a lower precision (single or half); when this ratio is equal to one, we omit it for the sake of readability. This ratio measures the potential loss of compression incurred when using low precision : indeed, the numerical noise introduced by the use of low precision may negatively affect the ability of the algorithm to tighten the dimensions to their optimal value if the requested accuracy τ is too close to the machine precision u.

Figure 4.4 confirms the numerical stability of Algorithm 4.4 : the relative error (4.51) is of order $\varepsilon = \max(\tau, u)$, which means that if an accuracy of order ε is requested, using a truncation threshold $\tau = \varepsilon$ and any precision $u \lesssim \tau$ will deliver an error of order ε. Using a precision u that is too close to (or larger than) τ is not desirable since it leads to a significant loss of compression ; however, taking u safely smaller than τ leads to exactly the same compression as if using double precision. To be precise, the figure shows that single precision arithmetic can be used without any loss of compression for $\tau \geq 10^{-7}$, whereas half precision arithmetic can be used for $\tau \geq 10^{-3}$. In conclusion, Algorithm 4.4 can be safely run in lower precision arithmetic, without degrading neither the accuracy nor the compression. This is a very attractive property that suggests that our algorithm should be able to take advantage of all the performance benefits of lower precision arithmetics on modern hardware.

4.6.3 . Comparison with Gram SVD

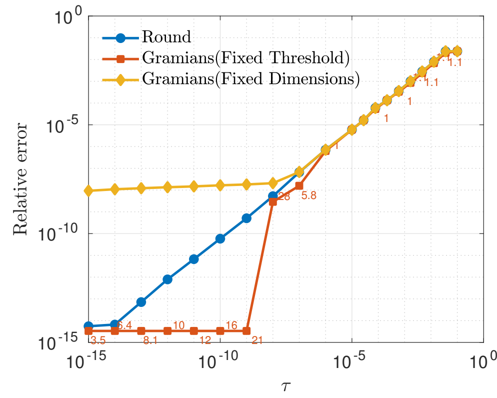
Next, we compare the numerical behavior of Algorithm 4.4 with that of the widely used Gram SVD.

As mentioned in Section 4.5.2, Gram SVD is computationally efficient and in particular very suitable for parallelization. However, it is also numerically unstable : the error analysis in [25] proves an error bound of order $\sqrt{u}$ where u is the machine precision.

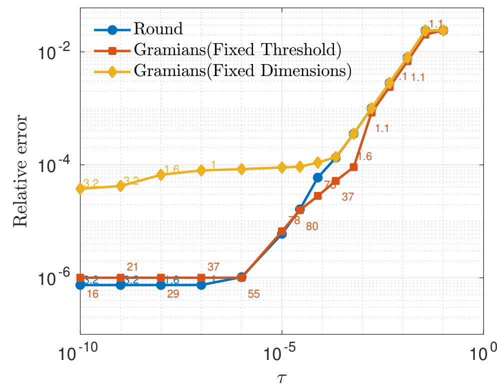
Figure 4.5 compares the accuracy of our Algorithm 4.4 and of Gram SVD when using double precision (Figure 4.5a), single precision (Figure 4.5b), or half precision (Figure 4.5c). We compare two variants of Gram SVD : one that uses a prescribed truncation threshold τ, and one that tightens the network to prescribed dimensions that correspond to the dimensions obtained via our Algorithm 4.4 with the corresponding τ.

The first Gram SVD variant with prescribed τ is unfortunately unable to correctly tighten the dimensions of the tensor except for very large values of τ, as indicated by the numbered labels that are greater than 1. This is due to the numerical noise of size $\sqrt{u}$ introduced by the instability of the algorithm, which therefore requires τ to be safely smaller than $\sqrt{u}$. For example, even in double precision, Gram SVD achieves a correct compression only when $\tau \geq 10^{-6}$. This effect is even worse when using single of half precisions, for which Gram SVD is effective only when $\tau \geq 10^{-3}$ and $\tau \geq 10^{-1}$, respectively.

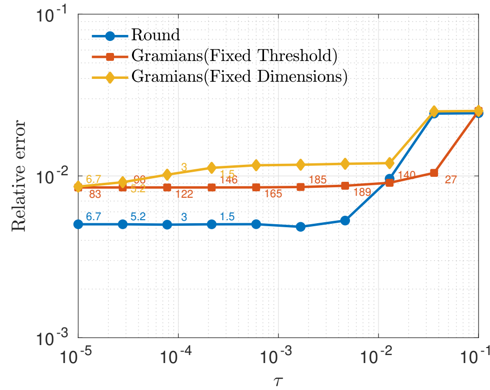
The second Gram SVD variant circumvents this issue by enforcing a truncation to some predetermined dimensions ; this is not a realistic scenario in general since the tightened dimensions may not be known in advance (in fact, here, we are using our Algorithm 4.4 to determine the correct dimensions). However, in some cases, it may be acceptable to heuristically truncate to some prescribed dimensions. In any case, Figure 4.5 shows that this variant of Gram SVD achieves an accuracy of order $\sqrt{u}$ as expected ; therefore, it achieves a



(a) Double precision



(b) Single precision



(c) Half precision

Figure 4.5 – Comparison between Algorithm 4.4 and Gram SVD depending on the truncation threshold τ and the floating-point precision u. The text labels next to the markers indicate the compression ratio between a given variant and the reference compression obtained with Algorithm 4.4 in double precision, for the same value of τ (if this ratio is equal to 1 we omit the label).

comparable accuracy to that of Algorithm 4.4 only when $\tau \geq \sqrt{u}$ and is much less accurate when $\tau \ll \sqrt{u}$.

These results confirm that our Algorithm 4.4 is much more stable than Gram SVD and is in particular more resilient to the use of low precisions ; therefore, it is better suited to exploit low precision arithmetics available on modern hardware.

4.7 . Conclusion

We have proposed a general framework for analyzing the stability of tensor computations in finite precision arithmetic. Our framework is based on tree tensor networks, which encompass a wide range of tensor formats. It defines four key kernel operations that, when combined, allow for expressing a wide range of tensor operations, such as compression, contractions, orthogonalization, and rounding. Our framework formalizes a key property, defined in Definition 4.2 and called α-normalization, that we have identified as fundamental for stability. An example of an α-normalized network is when all the nodes except one are semi-orthogonal (in this case $\alpha = e$, the vector of all ones).

We have performed an error analysis which leads to Theorem 4.2, our main theoretical result. It proves that an arbitrary computation composed of a sequence of local operations is stable provided that each local operation is stable, and that the network remains α-normalized throughout the computation (with small values for α). We use this key result to establish the stability of some common tensor computations in Corollaries 4.1 to 4.3.

Then, in Section 4.5, we have turned to the fundamental problem of tensor rounding. One of the most widely used tensor rounding algorithms is based on the Gram SVD, which is unstable in finite precision arithmetic. In view of the conclusions of our error analysis, we have defined a rounding algorithm that is careful to maintain the semi-orthogonality of all nodes except one throughout the computation. As a result, it is guaranteed stable (Corollary 4.4). Our numerical experiments have confirmed the stability of the algorithm in practice, and shown that it is significantly more resilient to the use of low precisions than Gram SVD. Thus, our algorithm is better suited to take advantage of all the performance benefits of lower precision arithmetics on modern hardware.

5 - Conclusion

5.1 . Contributions

In this last chapter, we summarize all our contributions. Each chapter exploits new methods to improve the efficiency of low-rank approximations methods. Because low-rank approximations becomes increasingly important to prevent the *curse of dimensionality*, it needs to be computationally efficient and not being a bottleneck of application.

Since low precision accelerates computation on modern hardware, a first idea was to generate a low precision version of low-rank approximation. The algorithm first computes a low-rank approximation in low precision. Then, it computes another low-rank approximation of the error term, also in low precision, to refine the accuracy of the approximation ; this process can be repeated to refine the accuracy further. We ensure the rank of the approximation remains bounded by using inexpensive recompression operations. The interest of the method is that it can be applied to any low-rank approximations method for either matrices or tensors. We have performed the error analysis of this algorithm, which proves that low precision determines the convergence speed, whereas attainable accuracy depends only on high precision. Thus, the algorithm can reach the high precision target accuracy, while most operations are done in low precision. We first validated the behavior of this method in Chapter 2 with MATLAB experiments. These experiments suggest that our method has an especially high potential when using hardware with very fast low precision arithmetic, such as GPUs with tensor core units.

To confirm this potential, we have developed in Chapter 3 a new implementation of randomized low-rank approximation for matrices using GPU tensor cores. We exploit mixed precision arithmetic to accelerate the computation of the matrix-matrix multiplication. Then, we accelerate the QR decomposition with the Cholesky QR kernel with double precision to avoid breakdown. We compare with a standard randomized LRA entirely in fp32 arithmetic, which achieves an average accuracy of order 10^{-4}. Our results show that our approach without refinement is up to $8\times$ faster, with an average accuracy of order 10^{-2}, which may be acceptable for some applications. Otherwise, we show that using refinement significantly improves the accuracy to an average of order 10^{-5}, while remaining up to $2.2\times$ faster than the standard fp32 randomized LRA.

The last contribution in Chapter 4 studies the numerical stability of tree tensor network operations. We develop an error analysis based on a general framework which allows us to prove the stability of a wide range of operations. In particular, we investigate how to perform tensor rounding in a stable

way, since the commonly used Gram SVD method is unstable. We propose a rounding algorithm that is guaranteed to be stable. We test the algorithm experimentally and confirm its better stability compared to the Gram SVD method. Our algorithm is better suited to take advantage of all the performance benefits of lower precision arithmetic on modern hardware.

5.2 . Perspective

All the above contributions presented open several perspectives for future research.

For matrices, the Chapter 3 highlights the speedup performance provided by recent hardware on low-rank approximations, as studied in Chapter 2. However, it also opens the perspective to apply the mixed precision scheme on the Cholesky QR [55] kernel used in Algorithm 3.4, which may exploit the tensor core units for the triangular solve while preserving the numerical stability.

Moreover, Chapter 2 highlights many hardware fields to explore with this iterative refinement scheme. For example, we can investigate an implementation on AMD GPU or Intel GPU rather than NVIDIA GPU. We can also consider less common hardware like Field-Programmable Gate Array (FPGA) to see if the method can be applied efficiently.

For tensors, Chapter 4 opens the perspective to apply our rounding method in a tree tensor network framework with parallelization to speed up the computation and compare the practical performance with the Gramian method on various computer architectures. A parallelization can be performed in the orthogonalization kernel `orthog`, where each factor at the same tree level can process its orthogonalization simultaneously.

Chapter 3 provides good results on matrices for mixed precision low-rank approximations, so we can similarly investigate a high performance implementation on modern hardware for tensors to experimentally validate the potential speedup.

During this thesis, we use *direct methods* to compute the low-rank approximation for tensors. However, we can also investigate the use of *iterative methods*, such as alternating least squares, to compute the low-rank approximation with iterative refinement. Our preliminary results show that the impact of using low precision in these iterative methods is harder to control and predict due to risk of being stuck at a local minimum.

Finally, we would like to focus on how the emergence of hardware with fast low precision arithmetic can benefit our applications across various domains, such as artificial intelligence or quantum computing, where large tensors are commonly encountered. Since our method encompasses many tensor formats, it will be applicable in many practical applications as a general way to tackle the *curse of dimensionality*.

5.3 . Acknowledgements

This thesis was financed by the grants from ANR (JCJC SELESTE, ANR-20-CE46-0008-01) and Paris Ile-de-France Region (DIM RFSI RC-TENSOR No 2021-05) and performed using the HPC resources at GENCI-IDRIS, Plafrim at Inria Bordeaux, and Mesocentre Paris-Saclay.

6 - Publications

Chapter 2 is based on the article [56] submitted on 02 June 2023 to SIAM *Journal on Scientific Computing* and currently under review. Chapter 3 is based on [62] published in Euro-Par24. The work in Chapter 4 will be submitted soon.

6.1 . Submitted articles

[1] Baboulin, Marc and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2023). "Mixed precision iterative refinement for low-rank matrix and tensor approximations". Submitted in : *SIAM Journal on Scientific Computing*.

[2] Baboulin, Marc and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2024). "Numerical stability of tree tensor network operations, and a stable rounding algorithm". Submitted in : *SIAM Journal on Scientific Computing*.

6.2 . Published articles

[1] Baboulin, Marc and Donfack, Simplice and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2024). "Mixed precision randomized low-rank approximation with GPU tensor cores". Published in : *Euro-Par24*

6.3 . Conference communications

[1] Baboulin, Marc and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2023). "Mixed precision iterative refinement for low-rank matrix and tensor approximations". Presented at : *RAIM2022* (2022) in Nantes, France.

[2] Baboulin, Marc and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2023). "Mixed precision iterative refinement for low-rank matrix and tensor approximations". Presented at : *SIAM CSE23* (2023) in Amsterdan, Netherlands.

[3] Baboulin, Marc and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2024). "A general framework for the stable rounding of low-rank tensors with error analysis". Presented at : *SIAM LA24* (2024) in Paris, France.

[4] Baboulin, Marc and Donfack, Simplice and Kaya, Oguz and Mary, Théo and Robeyns, Matthieu (2023). "Mixed precision randomized low-rank

approximation with GPU tensor cores”. Presented at : *Euro-Par24* (2024) in Madrid, Spain.

7 - Appendix

7.1 . Résumé en français

L'algèbre linéaire est un outil fondamental dans de nombreux domaines scientifiques tels que la physique, la chimie, la biologie et l'ingénierie [1, Part IV]. En informatique, son importance est encore plus prononcée avec les matrices, car elle est à la base de nombreux algorithmes et de structures de données. Ces dernières années, la quantité de données à traiter à augmenter de manière exponentielle, conduisant à de nouvelles structures de données telles que les tenseurs, qui sont la généralisation des matrices à des ordres supérieurs à deux. Des domaines de recherche récents tels que le traitement du signal, le traitement d'images, la chimie quantique et l'apprentissage automatique utilisent fréquemment les tenseurs [2, 3, 4, 5, 6, 7]. Cette quantité croissante de données a conduit à un défi important en informatique : la *curse of dimensionality* qui peut être abordé via une méthode appelée approximation de rang faible (ARF). Notre travail se concentrera sur le développement de nouveaux algorithmes pour calculer une ARF en arithmétique à faible précision, qui est une nouvelle direction prometteuse pour accélérer les calculs d'algèbre linéaire sur le matériel moderne.

L'ARF est un outil puissant utilisé dans de nombreuses applications scientifiques pour réduire la dimension de données à grande échelle [7, 8, 9, 10]. Par exemple, une matrice $X \in \mathbb{R}^{n \times n}$ peut être approximée par un produit de rang faible UV^T de matrices U et V de taille $n \times r$, réduisant le coût de stockage initial de $O(n^2)$ à $O(nr)$. Pour les tenseurs, ce coût de stockage est encore plus critique [7, 6, 8]—$O(n^d)$ pour un tenseur d'ordre d. Les méthodes d'ARF de tenseurs décomposent le tenseur plein en un produit de tenseurs de plus faible ordre et de plus faible rang ; plusieurs formats de tenseurs de faible rang ont été proposés comme le format Tucker [11, 12], le format Tensor-Train [9] et le format Tucker hiérarchique [13, 14].

Cependant, calculer les décompositions de matrices ou de tenseurs de faible rang est une tâche intensive en calcul ; elle représente le goulot d'étranglement de nombreuses applications basées sur l'ARF. Par conséquent, développer des algorithmes efficaces pour calculer une ARF est un problème crucial qui a fait l'objet de nombreuses études [15, 7, 8, 6].

Une nouvelle possibilité pour accélérer le calcul de l'ARF est d'utiliser les *faible précision arithmétiques*, qui offrent des avantages de performance significatifs sur le matériel moderne [16]. En particulier, les arithmétiques en *half precision* telles que les formats Institute of Electrical and Electronics Engineers (IEEE) fp16 et bfloat16 atteignent une vitesse très élevée sur les accélérateurs Graphics Processing Unit (GPU) avec un gain de vitesse allant jusqu'à $4\times$ par

rapport à la *double précision* (fp64) [17]. Cependant, la faible précision dégrade la précision des calculs ; par exemple, l'arithmétique en *half precision* fournit, au mieux, entre 3 et 4 chiffres significatifs, selon le format. De nombreuses applications nécessitent de calculer une ARF avec une précision plus élevée [15, 8].

Cela motive le besoin d'algorithmes de *précision mixte*, qui combinent plusieurs formats de précision dans le but d'atteindre la haute performance de la faible précision tout en préservant la haute précision des formats de haute précision [18, 19]. Contrairement à d'autres routines d'algèbre linéaire telles que la résolution de systèmes linéaires, il y a eu relativement peu de travaux sur la conception d'algorithmes de précision mixte (ou même faible précision) pour l'ARF.

Ainsi, l'objectif principal de cette thèse est de développer des algorithmes de précision mixte pour l'ARF qui soient à la fois précis et efficaces. Cet objectif a été atteint par trois contributions principales, qui sont expliquées ci-dessous.

ARF avec raffinement itératif En tant que première contribution, nous proposons une nouvelle méthode pour calculer une ARF en précision mixte définie dans Chapter 2. Notre approche est applicable à pratiquement n'importe quel algorithme d'ARF, impliquant des matrices ou des tenseurs. Elle rappelle le cadre de raffinement itératif utilisé pour résoudre des systèmes linéaires [20] : l'idée est de d'abord calculer une ARF en faible précision, puis évaluer l'erreur (ou le résidu) de cette première ARF, et réappliquer le même noyau d'ARF à ce terme d'erreur pour obtenir un terme de correction qui est utilisé pour affiner la précision de l'ARF. Cela peut être répété de manière itérative pour atteindre n'importe quel niveau de précision souhaité. L'ARF affinée est obtenue comme la somme de l'ARF en faible précision d'origine et du terme de correction, et est donc de rang plus grand, mais toujours de faible rang. Afin de contenir la croissance du rang et de maintenir le rang optimal tout au long des itérations, notre méthode utilise une stratégie de "recompression" [14, 21] qui est effectuée en haute précision, mais dont le coût reste asymptotiquement plus petit que celui de l'ARF et devient également un sujet de notre travail comme dernière contribution dans Chapter 4. Nous effectuons une analyse d'erreur de notre méthode basée sur un modèle d'erreur paramétré général qui suppose seulement que nous avons des implémentations numériquement stables des noyaux de base utilisés dans notre algorithme (ARF, multiplication de matrices et recompression). Nous montrons que la précision utilisée pour le noyau d'ARF—qui est le goulot d'étranglement de l'ensemble de la méthode—n'affecte que la vitesse de convergence du processus, mais pas sa précision atteignable. Afin d'évaluer dans quelles conditions nous pouvons nous attendre à ce que notre méthode soit bénéfique, nous effectuons une analyse de complexité qui mesure le coût de la

méthode en fonction du rang numérique de l'entrée ainsi que du rapport de vitesse entre l'arithmétique de faible et haute précision. Nous identifions deux situations dans lesquelles notre méthode à un fort potentiel. La première est lorsque le rang numérique de l'entrée est petit, en faible précision, ce qui signifie que les valeurs singulières de la matrice ou du tenseur décroissent rapidement ; dans ce cas, les premières itérations de notre méthode deviennent peu coûteuses. La seconde est lorsque le matériel fournit des unités de multiplication–accumulation de matrices en faible précision très rapides [17], ce qui permet de calculer l'ARF en faible précision à une vitesse très élevée.

ARF aléatoire en précision mixte sur GPU Les méthodes de projection aléatoire sont des techniques simples et robustes pour réduire la dimensionnalité des données tout en préservant la structure des données [22]. De plus, les opérations matricielles au cœur de ces méthodes les rendent très adaptées pour exploiter des accélérateurs tels que les GPUs [23]. Ainsi, nous étudions dans quelle mesure ces unités de calcul très rapides en faible précision peuvent être exploitées pour accélérer les méthodes de projection aléatoire, en particulier dans le cas des matrices. La nouvelle contribution est la conception d'une nouvelle méthode d'ARF aléatoire en précision mixte, avec une analyse de performance et de précision montrant que la méthode proposée est capable d'exploiter les unités *tensor cores* des GPU de manière fiable et efficace. Notre méthode repose sur trois idées clés : La première idée consiste à *effectuer les produits de matrices–matrices (noyau GEneral Matrix Multiplication (GEMM)) en précision mixte en utilisant les unités tensor cores*, car ces opérations représentent le goulot d'étranglement asymptotique de la méthode. Nous comparons plusieurs variantes de GEMM en fonction de la manière dont les conversions entre fp32 et fp16 sont gérées, et identifient une variante en particulier qui offre le meilleur compromis performance–précision. Ensuite, ayant considérablement accéléré les opérations de GEMM, nous observons que l'étape d'orthonormalisation (noyau QR), bien qu'exigeant un nombre négligeable de flops asymptotiques, devient le nouveau goulot de performance. Ensuite, la deuxième idée est de *passer la méthode d'orthonormalisation Householder QR standard à un algorithme CholeskyQR* [24], qui repose principalement sur les GEMM et est donc beaucoup plus efficace sur les GPUs. Nous atténuons l'instabilité inhérente de CholeskyQR en le réalisant en arithmétique fp64 plutôt qu'en fp32. Cela conduit à une méthode d'ARF aléatoire en précision mixte utilisant trois précisions (fp16, fp32 et fp64). Nous montrons que cette méthode peut être jusqu'à $8\times$ plus rapide que la méthode d'ARF aléatoire standard [22] en arithmétique à précision fixe fp32 et atteint une précision moyenne de l'ordre de 10^{-2}, ce qui peut être suffisant pour certaines applications. Ensuite, la troisième idée est d'*utiliser notre méthode de raffinement itératif pour l'ARF proposée dans la contribution précédente ci-dessus pour*

améliorer la précision de la méthode. Nous montrons qu'avec le raffinement, la précision de cette méthode s'améliore significativement pour atteindre une précision moyenne de l'ordre de 10^{-5}, tout en restant jusqu'à $2.2\times$ plus rapide que la méthode d'ARF standard en arithmétique fp32.

Stabilité des opérations sur les décompositions tensorielles de faible rang L'ARF avec raffinement itératif proposée dans la première contribution nécessite des noyaux stables, y compris un noyau de recompression. Cette hypothèse nous motive comme dernière contribution à étudier la stabilité du calcul des noyaux de tenseurs, qui n'a pas été étudiée. Ce manque d'analyse peut s'expliquer principalement par deux raisons : d'une part, les erreurs de troncature ont tendance à dominer les erreurs d'arrondi lorsque la haute précision est utilisée, de sorte que ces dernières ont été traditionnellement négligées ; et d'autre part, il existe une grande variété de formats de tenseurs différents, et leurs algorithmes associés peuvent être très complexes à analyser. Il est important de développer une analyse d'erreur précise pour identifier quels algorithmes de tenseurs sont stables et lesquels deviendront problématiques en faible précision. Notre contribution pour aborder ces questions est double. Premièrement, nous proposons un *cadre général basé sur un format de réseau tensoriel arborescent qui englobe les formats de tenseurs les plus importants* (Tensor-Train, Tucker et Tucker hiérarchique) et nous définissons les opérations essentielles nécessaires pour exprimer les algorithmes qui nous intéressent. Nous effectuons ensuite une analyse d'erreur d'un algorithme abstrait dans ce cadre et identifions des conditions pour garantir sa stabilité. En particulier, une condition clé est que la norme du tenseur doit être étroitement concentrée autour d'un seul nœud du réseau, une propriété que nous formaliserons dans Chapter 4. Cette propriété est notamment satisfaite lorsque tous les nœuds du réseau, sauf un, sont semi-orthogonaux. Notre analyse montre que si cette propriété est maintenue tout au long du calcul, alors l'erreur introduite par chaque opération est contrôlée en termes de la norme du tenseur global. Ce premier travail peut donc être utilisé pour établir la stabilité d'une large gamme de calculs de tenseurs qui peuvent être exprimés dans notre cadre. Notre deuxième travail est d'*utiliser notre cadre pour proposer un algorithme de recompression général*. Compte tenu des conclusions de notre analyse d'erreur, l'algorithme est attentif à maintenir la semi-orthogonalité de tous les nœuds, sauf un tout au long du calcul, et est donc garanti d'être stable. L'algorithme fonctionne pour n'importe quelle topologie d'arbre de tenseur et peut donc être appliqué à une large gamme de formats de tenseurs. Nous comparons ce nouvel algorithme de recompression avec l'algorithme de recompression basé sur le Gram SVD pour les tenseurs Tucker hiérarchiques proposé dans [14], qui est instable [25]. Nous montrons que notre nouvel algorithme peut améliorer significativement la précision de la recompression en arithmétique à précision finie, et peut ainsi exploiter de

manière plus fiable les arithmétiques à faible précision disponibles sur le matériel moderne.

Bibliography

[1] Nicholas J. Higham. “Functions of Matrices”. In : *Handbook of Linear Algebra*. Sous la dir. de Leslie Hogben. Boca Raton, FL, USA : Chapman et Hall/CRC, 2006, p. 11.1-11.13.

[2] Nicholas D Sidiropoulos et al. “Tensor decomposition for signal processing and machine learning”. In : *IEEE Transactions on signal processing* 65.13 (2017), p. 3551-3582.

[3] Carl J Appellof et Ernest R Davidson. “Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents”. In : *Analytical Chemistry* 53.13 (1981), p. 2053-2056.

[4] M Alex O Vasilescu et Demetri Terzopoulos. “Multilinear (tensor) image synthesis, analysis, and recognition [exploratory dsp]”. In : *IEEE Signal Processing Magazine* 24.6 (2007), p. 118-123.

[5] Pierre Comon et al. “Tensor decompostions : state of the art and applications”. In : *Institute of Mathematics and its Applications conference series*. T. 71. Oxford ; Clarendon ; 1999. 2002, p. 1-24.

[6] Pierre Comon. “Tensors : a brief introduction”. In : *IEEE Signal Processing Magazine* 31.3 (2014), p. 44-53.

[7] Tamara G. Kolda et Brett W. Bader. “Tensor Decompositions and Applications”. In : *SIAM Rev.* 51.3 (2009), p. 455-500. doi : 10.1137/07070111X.

[8] Lars Grasedyck, Daniel Kressner et Christine Tobler. “A literature survey of low-rank tensor approximation techniques”. In : *GAMM-Mitteilungen* 36.1 (2013), p. 53-78.

[9] Ivan V Oseledets. “Tensor-train decomposition”. In : *SIAM Journal on Scientific Computing* 33.5 (2011), p. 2295-2317.

[10] Lars Grasedyck et Christian Löbbert. “Distributed hierarchical SVD in the hierarchical Tucker format”. In : *Numerical Linear Algebra with Applications* 25.6 (2018), e2174.

[11] Ledyard R Tucker. “Some mathematical notes on three-mode factor analysis”. In : *Psychometrika* 31.3 (1966), p. 279-311. url : https://doi.org/10.1007/BF02289464.

[12] Lieven De Lathauwer, Bart De Moor et Joos Vandewalle. “A multilinear singular value decomposition”. In : *SIAM journal on Matrix Analysis and Applications* 21.4 (2000), p. 1253-1278.

[13] Wolfgang Hackbusch et Stefan Kühn. “A new scheme for the tensor representation”. In : *Journal of Fourier analysis and applications* 15.5 (2009), p. 706-722. url : https://doi.org/10.1007/s00041-009-9094-9.

[14] Lars Grasedyck. “Hierarchical singular value decomposition of tensors”. In : *SIAM journal on matrix analysis and applications* 31.4 (2010), p. 2029-2054.

[15] N Kishore Kumar et Jan Schneider. “Literature survey on low rank approximation of matrices”. In : *Linear and Multilinear Algebra* 65.11 (2017), p. 2212-2244.

[16] Guangli Li et al. “Unleashing the low-precision computation potential of tensor cores on gpus”. In : *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, p. 90-102.

[17] Pierre Blanchard et al. “Mixed Precision Block Fused Multiply-Add : Error Analysis and Application to GPU Tensor Cores”. In : *SIAM J. Sci. Comput.* 42.3 (2020), p. C124-C141. doi : 10.1137/19M1289546.

[18] Nicholas J. Higham et Theo Mary. “Mixed Precision Algorithms in Numerical Linear Algebra”. In : *Acta Numerica* 31 (mai 2022), p. 347-414. doi : 10.1017/s0962492922000022.

[19] Nicholas J. Higham. “Mixed Precision Computations”. In : *New Directions in Numerical Computation*. Sous la dir. de Tobin A. Driscoll, Endre Süli et Alex Townsend. T. 63. 4. Notices Amer. Math. Soc., 2016, p. 398-400. doi : 10.1090/noti1363.

[20] Erin Carson et Nicholas J. Higham. “Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions”. In : *SIAM J. Sci. Comput.* 40.2 (2018), A817-A847. doi : 10.1137/17M1140819.

[21] Hussam Al Daas, Grey Ballard et Lawton Manning. “Parallel tensor train rounding using gram svd”. In : *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2022, p. 930-940.

[22] Nathan Halko, Per-Gunnar Martinsson et Joel A Tropp. “Finding structure with randomness : Probabilistic algorithms for constructing approximate matrix decompositions”. In : *SIAM review* 53.2 (2011), p. 217-288.

[23] T. Mary et al. "Performance of Random Sampling for Computing Low-rank Approximations of a Dense Matrix on GPUs". In : *SC'15 - International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin, USA, nov. 2015. doi : 10.1145/2807591.2807613.

[24] Gene H Golub et Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[25] Théo Mary. "Error analysis of the Gram low-rank approximation (and why it is not as unstable as one may think)". working paper or preprint. Avr. 2024. url : https://hal.science/hal-04554516.

[26] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Philadelphia, PA, USA : Society for Industrial et Applied Mathematics, 2002, p. xxx+680. isbn : 0-89871-521-0. doi : 10.1137/1.9780898718027.

[27] Carl Eckart et Gale Young. "The approximation of one matrix by another of lower rank". In : *Psychometrika* 1.3 (1936), p. 211-218.

[28] James Demmel et al. "Computing the singular value decomposition with high relative accuracy". In : *Linear Algebra and its Applications* 299.1-3 (1999), p. 21-80.

[29] Nicholas J. Higham. "QR Factorization with Complete Pivoting and Accurate Computation of the SVD". In : *Linear Algebra Appl.* 309 (2000), p. 153-174. doi : 10.1016/S0024-3795(99)00230-X.

[30] Zlatko Drmac et Zvonimir Bujanovic. "On the failure of rank revealing QR factorization software–a case study LAPACK Working Note 176". In : (2006).

[31] Per-Gunnar Martinsson et Sergey Voronin. "A Randomized Blocked Algorithm for Efficiently Computing Rank-revealing Factorizations of Matrices". In : *SIAM J. Sci. Comput.* 38.5 (2016), S485-S507. doi : 10.1137/15M1026080.

[32] Andrzej Cichocki et al. "Tensor networks for dimensionality reduction and large-scale optimization : Part 1 low-rank tensor decompositions". In : *Foundations and Trends® in Machine Learning* 9.4-5 (2016), p. 249-429. doi : http://dx.doi.org/10.1561/2200000059.

[33] Nick Vannieuwenhoven, Raf Vandebril et Karl Meerbergen. “A New Truncation Strategy for the Higher-Order Singular Value Decomposition”. In : *SIAM Journal on Scientific Computing* 34.2 (2012), A1027-A1052. doi : 10.1137/110836067. eprint : https://doi.org/10.1137/110836067. url : https://doi.org/10.1137/110836067.

[34] Lars Grasedyck et Wolfgang Hackbusch. “An Introduction to Hierarchical (H-) Rank and TT-Rank of Tensors with Examples”. In : *Computational Methods in Applied Mathematics* 11.3 (2011), p. 291-304. doi : doi:10.2478/cmam-2011-0016. url : https://doi.org/10.2478/cmam-2011-0016.

[35] William Kahan. “IEEE standard 754 for binary floating-point arithmetic”. In : *Lecture Notes on the Status of IEEE* 754.94720-1776 (1996), p. 11.

[36] Nicola Peserico, Bhavin J Shastri et Volker J Sorger. “Integrated photonic tensor processing unit for a matrix multiply : a review”. In : *Journal of Lightwave Technology* 41.12 (2023), p. 3704-3716.

[37] Sparsh Mittal. “A survey of FPGA-based accelerators for convolutional neural networks”. In : *Neural computing and applications* 32.4 (2020), p. 1109-1139.

[38] Nicholas J. Higham et Srikara Pranesh. “Simulating Low Precision Floating-Point Arithmetic”. In : *SIAM J. Sci. Comput.* 41.5 (2019), p. C585-C602. doi : 10.1137/19M1251308.

[39] Patrick Amestoy et al. “Mixed Precision Low Rank Approximations and their Application to Block Low Rank LU Factorization”. In : *IMA Journal of Numerical Analysis* 43.4 (août 2023), p. 2198-2227. doi : 10.1093/imanum/drac037. url : https://hal.science/hal-03251738.

[40] Rise Ooi et al. “Effect of mixed precision computing on H-matrix vector multiplication in BEM analysis”. In : (2020), p. 92-101.

[41] Michael P. Connolly, Nicholas J. Higham et Srikara Pranesh. *Randomized Low Rank Matrix Approximation : Rounding Error Analysis and a Mixed Precision Algorithm*. MIMS EPrint 2022.5. Revised March 2023. UK : Manchester Institute for Mathematical Sciences, The University of Manchester, juill. 2022, p. 18. url : http://eprints.maths.manchester.ac.uk/2884/.

[42] Hiroyuki Ootomo et Rio Yokota. “Mixed-Precision Random Projection for RandNLA on Tensor Cores”. In : *Proceedings of the Platform for Advanced Scientific Computing Conference*. Davos, Switzerland : Association for Computing Machinery, 2023. doi : 10.1145/3592979.3593413.

[43] Alfredo Buttari, Théo Mary et André Pacteau. “Truncated QR factorization with pivoting in mixed precision”. In : (2024).

[44] Massimiliano Fasi et al. “Matrix Multiplication in Multiword Arithmetic : Error Analysis and Application to GPU Tensor Cores”. In : *SIAM J. Sci. Comput.* 45.1 (fév. 2023), p. C1-C19. doi : 10.1137/21m1465032.

[45] Zi Yang, Junnan Shan et Zheng Zhang. “Hardware-Efficient Mixed-Precision CP Tensor Decomposition”. In : *arXiv preprint arXiv:2209.04003* (2022).

[46] Emmanuel Agullo et al. “The backward stable variants of GMRES in variable accuracy”. In : (2022).

[47] Olivier Coulaud, Luc Giraud et Martina Iannacito. “A robust GMRES algorithm in Tensor Train format”. In : *arXiv preprint arXiv:2210.14533* (2022).

[48] Carlos Beltrán, Vanni Noferini et Nick Vannieuwenhoven. “When can forward stable algorithms be composed stably?” In : *arXiv preprint arXiv:2109.10610* (2021). url : https://doi.org/10.48550/arXiv.2109.10610.

[49] Brett W Bader et Tamara G Kolda. “Algorithm 862 : MATLAB tensor classes for fast algorithm prototyping”. In : *ACM Transactions on Mathematical Software (TOMS)* 32.4 (2006), p. 635-653.

[50] Ivan Oseledets. *TT-Toolbox v2.2.2*. 2023. url : https://github.com/oseledets/TT-Toolbox.

[51] Daniel Kressner et Christine Tobler. “htucker—A MATLAB toolbox for tensors in hierarchical Tucker format”. In : *Mathicse, EPF Lausanne* (2012).

[52] Maxim Rakhuba et Ivan Oseledets. “Calculating vibrational spectra of molecules using tensor train decomposition”. In : *The Journal of Chemical Physics* 145.12 (sept. 2016), p. 124101. doi : 10.1063/1.4962420. eprint : https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4962420/15517781/124101_1_online.pdf. url : https://doi.org/10.1063/1.4962420.

[53] Farouk Yahaya et al. “Random Projection Streams for (Weighted) Nonnegative Matrix Factorization”. In : *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2021, Toronto, ON, Canada, June 6-11, 2021*. IEEE, 2021, p. 3280-3284. doi : 10.1109/ICASSP39728.2021.9413496. url : https://doi.org/10.1109/ICASSP39728.2021.9413496.

[54] Riley Murray et al. “Randomized numerical linear algebra : A perspective on the field with an eye to software”. In : *arXiv preprint arXiv:2302.11474* (2023).

[55] Ichitaro Yamazaki, Stanimire Tomov et Jack Dongarra. “Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs”. In : *SIAM J. Sci. Comput.* 37.3 (2015), p. C307-C330.

[56] Marc Baboulin et al. “Mixed precision iterative refinement for low-rank matrix and tensor approximations”. working paper or preprint. Juin 2023. url : https://inria.hal.science/hal-04115337.

[57] Florent Lopez et Theo Mary. “Mixed precision LU factorization on GPU tensor cores : reducing data movement and memory footprint”. In : *The International Journal of High Performance Computing Applications* 37.2 (2023), p. 165-179. doi : 10.1177/10943420221136848. eprint : https://doi.org/10.1177/10943420221136848. url : https://doi.org/10.1177/10943420221136848.

[58] Román Orús. “A practical introduction to tensor networks : Matrix product states and projected entangled pair states”. In : *Annals of physics* 349 (2014), p. 117-158.

[59] Glen Evenbly et Guifré Vidal. “Tensor network states and geometry”. In : *Journal of Statistical Physics* 145 (2011), p. 891-918.

[60] Simon Etter. “Parallel ALS algorithm for solving linear systems in the hierarchical Tucker representation”. In : *SIAM Journal on Scientific Computing* 38.4 (2016), A2585-A2609.

[61] Sebastian Krämer. “Tree tensor networks, associated singular values and high-dimensional approximation”. Thèse de doct. 2020.

[62] Marc Baboulin et al. “Mixed precision randomized low-rank approximation with GPU tensor cores”. working paper or preprint. Juin 2024. url : https://hal.science/hal-04520893.