HAL
open science

Multiplication matricielle efficace et conception logicielle pour la bibliothèque de calcul exact LinBox

Brice Boyer

To cite this version:

Brice Boyer. Multiplication matricielle efficace et conception logicielle pour la bibliothèque de calcul exact LinBox. Mathématiques générales [math.GM]. Université de Grenoble, 2012. Français. NNT : 2012GRENM019 . tel-00767915

HAL Id: tel-00767915

https://theses.hal.science/tel-00767915

Submitted on 20 Dec 2012

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.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques – Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Brice Boyer

Thèse dirigée par **Jean-Guillaume Dumas**

préparée au sein du **Laboratoire Jean-Kuntzman**
et de l'**école doctorale MSTII**

Multiplication matricielle efficace et conception logicielle pour la bibliothèque de calcul exact LinBox

Thèse soutenue publiquement le **21-06-2012**,
devant le jury composé de :

M. Yves DENNEULIN
PR, ENSIMAG, Président
M. David SAUNDERS
PR, University of Delaware, Rapporteur
M. Nicolas THIÉRY
MdC, Université Paris Sud, Rapporteur
Mme. Dominique DUVAL
PR, Université de Grenoble, Examinateur
M. Pascal GIORGI
MdC, Université Montpellier2, Examinateur
M. Jean-Guillaume DUMAS
MdC, Université de Grenoble, Directeur de thèse

À mes parents, à la mémoire de mon grand-père.

Remerciements

Tout d'abord, je remercie Yves Denneulin pour avoir accepté de présider le jury de cette thèse, ainsi que Nicolas Thiéry et B. David Saunders pour avoir rapporté ce manuscrit et apporté de précieuses améliorations.

Le cheminement jusqu'à la rédaction de cette thèse n'a pas toujours été facile et je tiens à remercier ici toutes les personnes qui m'ont aidé, soutenu, encouragé pendant ces quatre années.

C'est ensuite à mon directeur de thèse Jean-Guillaume Dumas que vont mes remerciements, pour ses conseils, son enthousiasme, et tout particulièrement ses sujets intéressants, ses propositions ambitieuses et ses idées productives.

Il y a toute l'équipe du projet LinBox qui m'a donné envie d'apprendre les langages C/C++, de contribuer librement du code, et surtout d'orienter une partie de mes travaux de thèse vers des aspects à la fois pratiques et de conception logicielle. En particulier, il y a eu de nombreuses conversations et échanges avec Clément Pernet, David B. Saunders, Pascal Giorgi qui ont amené un travail très productif. Les multiples séjours à Grenoble de Jean-François Biasse (University of Calgary) ont été très intensifs et productifs. Je remercie aussi Wayne Eberly qui m'a accueilli à l'université de Calgary, Alberta au Canada – ainsi que le chaleureux accueil qui m'a été fait par les canadiens que j'ai rencontrés durant cet hiver 2009–2010.

Au sein du laboratoire, je remercie Roland Denis pour nos nombreuses discussions sur les subtilités de divers langages de programmation qui m'ont permis de résoudre de nombreux problèmes. La sympathique ambiance qui règne entre thésards dans la tour IRMA est aussi un atout certain pour nous aider à traverser sereinement ces années de thèse, merci à tous ! Plus particulièrement, je remercie Souleymane Kadri, Christophe Chabot, Thomas Oberlin ou Guillaume Ollier qui se sont succédés dans notre remarquable bureau 16. Enfin, le travail parfait du personnel administratif du 4e étage est à apprécier et à remercier.

Finalement, je tiens à remercier tout particulièrement les personnes de la vie en dehors de la thèse, qui m'ont soutenues et encouragées, notamment mes parents et grands-parents, mon amie Chloé. Je ne veux pas non plus oublier mes amis d'avant ou mes compagnons de cordée qui se reconnaîtront et qui j'en suis sûr apprécieront mes nouvelles disponibilités...

Table des matières

Remerciements iii

Table des matières v

Table des figures ix

Liste des tableaux xi

Liste des algorithmes xiii

Liste des extraits de code xv

Introduction. 1

Contexte. 3

Quelques problématiques du calcul exact. 3

Environnement logiciel. 4

Plan du mémoire. 5

Notations. 7

I Multiplication efficace de matrices denses. 9

Introduction. 11

L'algorithme de Strassen et ses variantes. 11

Les algorithmes pour le produit $C \leftarrow AB$. 12

Produit avec accumulation $C \leftarrow C + AB$. 13

Implantations dans les logiciels de calcul exact. 13

Plan de la première partie. 14

1 Amélioration du placement en mémoire pour la multiplication de matrices. 15

1.1 Meilleur placement en mémoire : un jeu de galet. 16
1.1.1 Présentation générale du problème. 16
1.1.2 Le jeu de galet. 17
1.1.3 Implantation du jeu de galet. 18
1.1.4 Conclusion et prolongements. 21
1.2 Ordonnancements efficaces en mémoire. 21
1.2.1 Ordonnancements avec entrées constantes : une première approche. 22
1.2.2 Ordonnancements avec écrasement des opérandes : cas carré. 27
1.2.3 Ordonnancements avec entrées constantes : cas carré. 32
1.2.4 Cas général : ordonnancements hybrides. 32
1.2.5 Synthèse. 36
1.2.6 Implantation des nouveaux ordonnancements. 36
1.3 Prolongements. 36

2 Recherche de nouvelles formules de multiplication. 39
2.1 Recherche automatique de formules de multiplication matricielle... 39
2.1.1 Présentation et formalisation du problème. 39
2.1.2 Préparation de l'implantation de la recherche de formes bilinéaires. 40
2.1.3 Recherche automatique de nouvelles formules. 42
2.1.4 Utilisation de formules approchées sur les corps finis ? 45
2.2 Prolongements. 46

II Multiplication matrice creuse et vecteur dense. 47

Introduction. 49
Pourquoi l'opération SpMV ? 49
Généralités sur SpMV. 50
Formats de stockage. 50
Outils logiciels. 50
L'évolution du matériel. 51
Plan du chapitre. 52

3 Présentation de ffspmvgpu. 53
3.1 Formats traditionnels. 53
3.2 Implantation de base. 57
3.2.1 Représentation des corps finis. 57
3.2.2 Adapter les bibliothèques numériques. 59
3.2.3 Utilisation d'OpenMP. 59
3.3 Nouveaux formats. 59
3.3.1 Premier essai : compilation à la volée (*JIT*). 61
3.3.2 Prise en compte des ± 1. 62
3.3.3 Retour sur les formats de base. 63
3.3.4 Formats hybrides. 64
3.3.5 Algorithme heuristique pour le choix des formats. 64
3.4 Version itératives et par blocs de vecteurs. 64
3.4.1 Utiliser les multi-vecteurs. 65

3.4.2 Performance et problèmes. 65
3.5 Prolongements. 67

4 Application au calcul du rang. 69
4.1 Parallélisation de SpMV sur CPU et GPU. 69
4.1.1 Parallélisation du calcul de la suite matricielle. 70
4.1.2 Parallélisation du calcul des σ-bases. 70
4.1.3 Calcul parallèle du co-degré du déterminant. 72
4.1.4 Performances de la version parallèle de l'algorithme de Wiedemann. 73
4.2 Prolongements. 73

III Conception logicielle pour une bibliothèque de calcul. 75

Introduction. 77
Quelques problèmes et défis pour une bibliothèque de calcul. 77
Le choix du langage C++ modèle la bibliothèque. 78
Configuration et installation simplifiées, préalable à une bonne conception. 80
Plan de la troisième partie. 82

5 Conception d'une multiplication matricielle efficace et générique : la solution `mul`. 83
5.1 Matrices et Vecteurs dans LinBox : simplification et homogénéisation. 83
5.1.1 Nouvelles matrices et vecteurs dans LinBox. 84
5.1.2 Opérateur `rebind` et modèle d'allocation. 86
5.1.3 Description des matrices BLAS et creuses. 86
5.1.4 Permutations. 89
5.1.5 Prolongements. 90
5.2 La solution mul. 90
5.2.1 Multiplication générique : système contrôleur–modules. 91
5.2.2 Nouvelles routines BLAS dans FFLAS. 92
5.2.3 Multiplication de matrices entières. 95
5.2.4 Conclusion, prolongements. 97

6 Optimisation des performances. 101
6.1 Garantir des performances par leur mesure (*benchmarking*). 102
6.1.1 Des *benchmarks* pour mesurer, tester, paramétrer. 102
6.1.2 Une *framework* de *benchmarks* partagée par les utilisateurs et l'optimiseur. . . 103
6.1.3 Prolongements. 106
6.2 Amélioration des performances : conception par briques de base. 107
6.2.1 Optimisations, micro-optimisations : défi ou temps perdu ? 108
6.2.2 Conception générique par *building blocks* (briques modulables). 109
6.2.3 Conclusion. 112

7 Amélioration de la qualité du code. 113
7.1 Comment documenter une bibliothèque générique ? 114
7.1.1 Annoter proprement et simplement le code. 114
7.1.2 Documenter simultanément et efficacement la bibliothèque pour l'utilisateur et le développeur. 115

7.1.3 Prolongements. 116
7.2 Rendre le code plus sûr. 117
7.2.1 Tester et vérifier une bibliothèque générique. 117
7.2.2 Normalisation du code : robustesse et pérennisation. 119
7.3 Conclusion. 125

Conclusion. 127

Conclusion et perspectives. 129

Index 131

Bibliographie 133

Table des figures

0.1 LinBox : un intergiciel 5

1.1 Arbre des dépendances dans l'algorithme de Winograd. 16
1.2 Arbre des dépendances pour $C \leftarrow \alpha AB + \beta C$ avec 6 accumulations. 17
1.3 Arbre des dépendances pour $C \leftarrow \alpha AB + \beta C$ avec 5 accumulations. 18
1.4 Importance du nombre d'appels récursifs dans la complexité de `wacc`. 26

2.1 Un exemple de prompt. 44
2.2 Une capture d'écran de l'interface Qt. 44

3.1 Format `COO` 54
3.2 Format `CSR` 54
3.3 Format `CSC` 54
3.4 Format `ELL` 55
3.5 Format `ELL_R` 55
3.6 Format `DIA` 56
3.7 Format `JDS` 56
3.8 Format `BCSR` 56
3.9 Retarder `fmod` 57
3.10 Performances de ffspmvgpu sur CPU/GPU et `float/double` 58
3.11 Format `CSR` parallélisé par OpenMP avec N cœurs, sur Palo-Alto@ujf et sur Joran@imag 61
3.12 Prise en compte des ± 1 lors de SpMV 63
3.13 Comparaison des performances des divers formats de stockage sur CPU/GPU 63
3.14 Format `COO_S` 64
3.15 Accélération du format auto-généré par rapport à `CSR` sur CPU et GPU. 65
3.16 Multivecteurs et accélération de ffspmvgpu sur CPU/GPU 65
3.17 Accélération sur GPU de ffspmvgpu par réutilisation de données 66

4.1 ffspmvgpu *vs.* LinBox lors de la génération de la suite matricielle 70
4.2 Multiplication matricielle polynomiale parallèle avec OpenMP. 71
4.3 Calcul en parallèle des σ-bases avec OpenMP 72

5.1 Format de fichier CSR. 89

5.2 Patron de conception contrôleur–modules 91
5.3 Conception d'un algorithme récursif contrôlé 92
5.4 Complexité algorithmique de la cascade Winograd-BLAS 93
5.5 Complexité spatiale de la cascade Winograd-BLAS 93
5.6 Nombre d'allocations total pour la cascade Winograd-BLAS 93

6.1 Exemple de sortie de `benchmark` sur Perso@home : `fgemm`. 104

Liste des tableaux

0.1 Comparaison des temps des produits avec ou sans accumulation 13

∗ Ordonnancements pour la multiplication matricielle ∗

1.1 wino 22
1.2 wacc 23
1.3 accw 25
1.4 accw2 26
1.5 IP 27
1.6 IPbis 27
1.7 OvL 28
1.8 OvR 28
1.9 OvLbis 29
1.10 OvRbis 29
1.11 AcLR 30
1.12 AccR 30
1.13 AccL 31
1.14 Acc 32

∗ ∗ ∗

1.16 Multiplication de matrices rectangulaires : temps de calculs en secondes sur Joran@imag. 36
1.15 Tableau général des complexités 37

2.1 L'algorithme de Winograd bis. 40
2.2 Résultats (prémisses) 43
2.3 Formule approchée (3, 2, 2) de Bini. 45
2.4 Algorithme approché de Bini *vs.* Winograd (fgemm) sur $\mathbb{Z}/101\mathbb{Z}$ avec seuil de 1 000. . . 46

3.1 Profil des matrices creuses 60

4.1 Calcul du rang dans $\mathbb{Z}/65521\mathbb{Z}$ avec OpenMP et l'algo. par blocs de Wiedemann 73
4.2 Évolution du nombre de lignes de code dans LinBox 78
4.3 Temps de 'make check' avec LinBox-1.3.0, sur Joran@imag, avec gcc-4.7 82
4.4 Temps de 'make check -j2' avec LinBox-1.3.0, sur Joran@imag, avec gcc-4.7 82

5.1 Comparaison de divers algorithmes pour la multiplication matricielle entière 96

6.1 Comparaison des opérations axpy et .+=.*. (µs) sur Joran@imag 110
6.2 Optimisations de applyP : calcul de $P\left(P^{-1}AP\right)P^{-1}$ (secondes). 111
6.3 Optimisations de fgemm : calcul de $C = 0AB + \beta C$ (secondes). 111

7.1 Évolution de la prise en charge des compilateurs dans LinBox 120

Liste des algorithmes

0.1 L'algorithme de Strassen. 12
0.2 L'algorithme de Winograd. 12
0.3 L'algorithme de Bodrato. 13
0.4 L'algorithme $C \leftarrow AB + C$ de [HLJJ+96b]. 14

1.1 `Explore : recherche d'ordonnancements avec le jeu de galets` 19
1.2 Un nouvel algorithme de Winograd pour $C \leftarrow AB + C$. 24
1.3 `IPOvMM` : multiplication de matrice en place avec écrasement 33
1.4 `IPMM` : multiplication matricielle rapide en place 34

4.1 Multiplication rapide de matrices polynomiales 71

5.1 `AlgoSeuil` : contrôleur 92
5.2 `RecursiveCase` : module récursif 92
5.3 `ftrtr`(`left`, `istrans`, L, U) 94
5.4 `ftrtr`(`left`, `istrans`, U, L) 94
5.5 `ftrtr`(`right`, `istrans`, $L^{(1)}$, $L^{(2)}$) 94

Liste des extraits de code

1.1 Exemple d'arbre en entrée de galet. 20

3.1 Utilisation de routines numériques. 59
3.2 Compilation d'une matrice creuse 61
3.3 Compilation de matrice en librairie dynamique. 62
3.4 Utilisation de la matrice dans la librairie dynamique. 62
3.5 Compilation d'une matrice sur GPU. 62
3.6 Réutilisation des données sur GPU 66

4.1 Exemple de solution `rank.h`. 79
4.2 Style de programmation de LinBox. 79
4.3 Configuration et installation automatique de LinBox. 81

5.1 Exemple de formats de matrices denses. 84
5.2 Classes de matrices dans LinBox. 85
5.3 Interface d'une BoîteNoire 85
5.4 Vecteurs unifiés dans LinBox. 86
5.5 Exemple d'opérateur `rebind`. 86
5.6 Ancienne interface d'une matrice 87
5.7 Matrice dense (format BLAS) 87
5.8 Sous-matrice dense (format BLAS) 87
5.9 Matrice creuse dans LinBox-1.1.7. 88
5.10 Matrice creuse sous formats standards. 88
5.11 Matrices de permutations dans LinBox.. 90
5.12 Implantation d'`apply` sur un vecteur par solution `mul`. 91
5.13 `ftrtrm` 92
5.14 Sélection des algorithmes par traits. 97
5.15 Implantation de l'algorithme Hermite. 98
5.16 Implantation de la solution Hermite 99
5.17 IML wrapping 99

6.1 Structure de données des benchmarks. 104
6.2 Paramètre par défaut (multiplication). 105
6.3 Paramètre par défaut (systèmes entiers). 105
6.4 Fonction par défaut (fmod). 106
6.5 Seuils et fonctions par défaut. 106

6.6 Utilisation d'instructions SSE pour additionner deux vecteurs. 109
6.7 Opérateur réduction modulaire % ou fmod. 109
6.8 Utilisation des blocs génériques *vs.* les opérateurs. 110
6.9 Optimisation de fscal : retours rapides. 112

7.1 Désactivation de code par pré-processeur. 114
7.2 static_assert : assertion lors de la compilation. 118
7.3 Spécialisation du code pour certains compilateurs. 120
7.4 Mention longue de la licence de LinBox. 121
7.5 Licence raccourcie. 122
7.6 Modèle d'un fichier source .h. 122
7.7 Styles de codage pour Vim et Emacs. 123
7.8 Style de codage : fonctions 124
7.9 Style de codage : classe 124
7.10 Style de codage : if-else 124
7.11 Style de codage : switch 124
7.12 Exemple d'automatisation de respect des règles. 125

Introduction.

Contexte.

Sommaire

Quelques problématiques du calcul exact. 3
Environnement logiciel. 4
Plan du mémoire. 5

Nous pouvons qualifier le contexte général dans lequel nous travaillons en les termes *algèbre linéaire efficace*. La première partie de cette description — *algèbre linéaire* — nous emmène à la recherche de nouveaux algorithmes, de nouvelles techniques pour résoudre des problèmes d'algèbre linéaire, tandis que la seconde — *efficace* — nous entraîne dans des considérations logicielles ou matérielles. En essence donc, l'art de l'*algèbre linéaire efficace* consiste à développer de pair ces deux aspects. À la différence des numériciens, nous développons de l'algèbre linéaire sur des corps dont la représentation et les opérations sont exactes, p.ex. les entiers, les corps finis. Nous disons alors que nous faisons du *calcul exact*.

Dans ce mémoire donc nous allons introduire et développer de nouveaux algorithmes, adapter des techniques, implanter efficacement des routines, mais aussi réfléchir à la conception (*design*) d'un logiciel à la fois générique et efficace.

Quelques problématiques du calcul exact.

L'algèbre linéaire est une brique élémentaire du calcul exact et beaucoup de problèmes algorithmiques en découlent. Ces problèmes peuvent se situer à un niveau de base : multiplier des matrices, faire des opérations sur les vecteurs (**B**acic **L**inear **A**lgebra **S**ubroutines). Nous pouvons aussi chercher les meilleures complexités pour certains problèmes majeurs (rang, résolution de système, noyau, déterminant, formes normales de matrices, polynômes caractéristiques ou minimaux...). À un niveau intermédiaire, de nombreux algorithmes ont un rôle de pierre angulaire (restes chinois, relèvements...). Savoir créer logiciellement une bonne alchimie entre ces trois niveaux n'est pas tâche aisée : c'est un enjeu que nous devons constamment garder à l'esprit.

Dans cette idée directrice de rendre efficace nos problèmes phares d'algèbre linéaire, il n'est évidemment pas suffisant, même si cela est nécessaire, de ne chercher qu'à réduire leur complexité asymptotique, soient-elles praticables. Il faut justement pouvoir en créer une implantation efficace, pour toute taille de paramètres. Savoir rendre une implantation performante passe par plusieurs étapes. Cela peut commencer par réduire théoriquement les constantes et les termes cachés dans nos complexités théo-

riques en $\mathcal{O}(\cdot)$ ou en $\mathcal{O}^{\sim}(\cdot)$. Cela passe aussi par une utilisation adéquate et aisée d'autres algorithmes de plus bas niveau, très optimisés. Il y a dans cette dernière phrase une grande partie de la problématique de nos travaux de recherche. D'abord, rendre une *utilisation simple*, c'est avoir à disposition une interface claire et simple vers les routines requises. Cependant, devant la multitude d'algorithmes proposés, choisir le bon algorithme relève aussi de la *compréhension* de ces algorithmes dans leur contexte : cela passe par une organisation pertinente du code et de sa documentation.

Se rapprocher de niveaux les plus bas n'est cependant pas sans risques, c'est pourquoi il faut créer un certain niveau d'abstraction dans l'écriture de nos algorithmes. En effet, le matériel évolue rapidement, les paradigmes associés évoluent aussi (modèles de mémoire, modèles de calculs), les interfaces logicielles sont souvent propriétaires, non complètes, ou non pérennes (p.ex. la technologie Cuda de ATI/AMD qui sans cesse évolue). Il faut donc toujours savoir intercaler une couche d'abstraction afin de mieux anticiper les évolutions. En particulier, après la montée en puissance des cartes mères — plus de jeux d'instructions sur les processeurs, des caches plus importants, une mémoire plus grosse, plus rapide, des bus plus rapides, et surtout plus d'instructions par secondes — , arrive une généralisation du parallélisme (multi-cœurs, multi-processeurs, cartes graphiques en mode *compute*) et des ressources hybrides (typiquement multi-cœur/GPU sur les ordinateurs personnels). Des questions se posent alors : comment adapter les algorithmes existants aux nouveaux paradigmes parallèles ? À quel niveau mettre le parallélisme ?

Les instructions de type SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata, instruction unique, données multiples) ne sont certes pas toutes récentes (SSE, altivec), ni l'utilisation des cartes graphiques pour faire du calcul numérique (OpenGL, GLSL), mais à l'heure des débuts du GPGPU (**G**eneral **P**urpose GPU), il semble que leur visibilité se soit fortement accrue et surtout que leur puissance ait gagné des ordres de grandeur. Il semble aussi que les opérations vectorielles redeviennent à la mode chez les constructeurs de processeurs — instructions vectorielles AVX sur les processeurs récents (Intel Sandy Bridge, AMD Bulldozer...). Une preuve de cette généralisation se trouve dans la naissance d'un nouveau langage et d'un standard, OpenCL, conçu pour les systèmes parallèles hétérogènes, supporté et développé par une majorité des grands constructeurs. Tout comme la création du langage C a permis d'utiliser très efficacement l'architecture des processeurs alors à disposition, ce langage permettra-t-il de produire des codes efficaces sur des environnements hétérogènes et parallèles ?

Environnement logiciel.

Nous confrontons ces problèmes à la réalité des implantations dans divers logiciels de calcul exact que nous développons et utilisons. Nous les présentons maintenant brièvement :

- La bibliothèque Givaro [1] [GVR+05] permet de faire de l'arithmétique rapide sur les corps finis, les entiers et rationnels, les polynômes. Il est en concurrence avec de nombreux autres projets (NTL, FLINT...). Avec les développements et les besoins en théorie des nombres et en cryptographie, cette concurrence ne peut que s'accroître.
- La bibliothèque FFLAS-FFPACK [2] [Per01, DGP08] permet de faire de l'algèbre linéaire rapide sur les corps premiers de taille machine. Elle mène une analogie avec les BLAS sur lesquelles elle repose fondamentalement en fournissant un sous ensemble intéressant des routines de type BLAS sur ces corps. La seconde analogie, avec LAPACK, concerne des routines d'algèbre linéaire de plus haut niveau qui se basent sur les FFLAS, p.ex. `LUdivine`, `Rank` ou `NullSpace`.
- Finalement et non des moindres, la bibliothèque LinBox [3] [DEG+05], permet de faire de l'algèbre linéaire exacte sur les entiers, rationnels et les corps finis. Cette bibliothèque est développée internationalement (Canada, États-Unis, France) depuis maintenant bientôt une quinzaine d'années. La bibliothèque LinBox, ainsi que Givaro, est disponible sur de nombreuses distributions Linux (Debian, Fedora,

1 — Disponible à l'adresse `https://forge.imag.fr/projects/givaro/`.
2 — Disponible à l'adresse `http://linalg.org/projects/fflas-ffpack`.
3 — Disponible à l'adresse `http://linalg.org/`.

Gentoo...). Elle est bâtie avec un double souci de généricité et de hautes performances. Nous allons dans les paragraphes suivants la décrire plus précisément.

LinBox : un intergiciel.

Comme l'illustre la figure 0.1, la bibliothèque LinBox se situe comme un intergiciel (*middleware*). Elle propose une interface vers d'autres logiciels de plus haut niveau en leur proposant ses *solutions* (rang, déterminant, résolution de système...) mais aussi elle *interface* de nombreux autres logiciels et bibliothèques spécialisés. De cette manière, LinBox permet par exemple d'abstraire pour Sage une partie de son algèbre linéaire dense. Mais aussi, en reposant sur une sous-couche assez stable et modulaire, LinBox gagne en puissance et en fonctionnalités et s'ancre dans un environnement logiciel foisonnant.

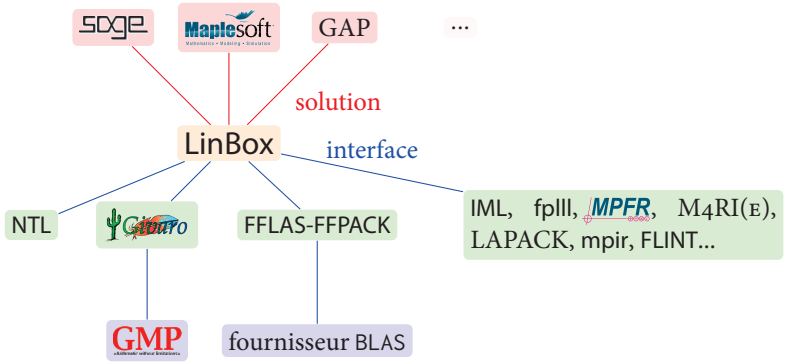

Fig.. 0.1 — LinBox : un intergiciel

LinBox : une bibliothèque générique et performante.

Un but de LinBox est d'allier la généricité à la performance. Elle est écrite en langage C++, facilitant en partie cette double tâche grâce au mécanisme de `template`/spécialisation qui est fortement utilisé.

Être *générique*, c'est savoir résoudre de manière satisfaisante le même problème pour des types de données variés (par exemple calculer un déterminant sur des matrices creuses ou denses, sur des corps finis ou non). Être *performant*, c'est, sur un problème donné, offrir une solution par rapport à laquelle d'autres logiciels et ou travaux de recherches pourront se comparer. Ces deux problèmes ne sont pas étrangers et ils entraînent des problèmes de conceptions ardus.

LinBox : une bibliothèque libre.

Pour finir, la plus grande partie de l'environnement logiciel dans lequel nous évoluons est libre, avec une licence GPLv2 ou similaire. Les projets logiciels que nous développons partagent aussi cette philosophie. Car il est important que les techniques logicielles que nous développons tout comme nos algorithmes soient disponibles librement pour la communauté académique, les étudiants, mais aussi bien au-delà. Il est aussi important que les chercheurs de domaines connexes et autres utilisateurs puissent trouver un outil performant dans lequel coder et tester *aisément* leurs nouveaux algorithmes et en faire à leur tour bénéficier toute la communauté. La facilité d'utilisation et d'adaptation à LinBox est donc aussi probablement une des lignes directrices qui doit orienter la conception de notre bibliothèque.

Plan du mémoire.

Dans cette thèse, nous allons nous pencher sur divers sujets, tous liés à la conception et à la distribution d'une bibliothèque mathématique d'algèbre linéaire exacte, générique et efficace. Nous commencerons dans la partie I par réfléchir à divers algorithmes de multiplication efficace de matrices denses en mettant l'accent sur la recherche d'ordonnancements performants, notamment du point de vue de l'utilisation de la mémoire. Dans une seconde partie (II), nous nous attellerons à rendre efficace la multiplication entre une matrice creuse et une matrice ou un vecteur dense. Pour cela, nous utiliserons et

créerons d'abord des algorithmes et des formats de stockage de données, mais aussi nous explorerons les capacités de nouveaux horizons de parallélisation (calculs numériques sur GPU). Finalement, dans la partie III, nous nous intéressons à la conception d'une bibliothèque mathématique — en particulier LinBox — . Nous montrerons comment nous pouvons rendre, grâce à des choix de conception pertinents, une bibliothèque mathématique et générique performante.

Notations.

Nous essaierons tout au long de ce manuscrit de conserver une unité dans les notations. Nous notons en capitales grasses les ensembles, tels les entiers $\mathbf{Z}$. Les scalaires seront généralement des minuscules grecques α, β... Les matrices sont représentées en lettres capitales (A) tandis que les vecteurs seront représentées des minuscules grasses, $\mathbf{v}$. En général p représente un nombre premier et les autres minuscules (m, q...) des entiers.

Les noms de logiciels ou de programmes sont écrits avec une police sans empattements (LinBox) et tout ce qui se rapporte à du code machine est écrit dans une police à chasse fixe (`forall`). Parfois, même si nous essayons de les éviter, les mots issus de l'anglais sont en italique (*wrap*).

Sauf mention explicite, les ordinateurs utilisés pour les mesures sont nommés :

— Joran@imag : processeur Intel® Core2™ Duo E8400 2× 3GHz, carte graphique NVidia Quadro NVS 290, architecture [4] Cuda1.1, 4Go (giga-octets) de mémoire RAM ;

— Palo-Alto@ujf : processeur Intel® Xeon® X5482 8× 3,2GHz, carte graphique NVidia GTX 280, architecture Cuda1.3, 32 Go de RAM ;

— Montpellier@lirm : processeur Intel® Xeon® E5345, 8× 2,33GHz, 16 Go de RAM ;

— Perso@home : processeur AMD 2,2GHz sans carte graphique utilisable par ATI Stream SDK, 1Go de RAM.

4 — Pour plus de détails : http://developer.nvidia.com/cuda-gpus.

I

Multiplication efficace de matrices denses.

Introduction.

Sommaire

L'algorithme de Strassen et ses variantes. 11

Les algorithmes pour le produit $C \leftarrow AB$. 12

Produit avec accumulation $C \leftarrow C + AB$. 13

Implantations dans les logiciels de calcul exact. 13

Plan de la première partie. 14

Dans cette partie, nous nous allons étudier des techniques de multiplication matricielle dense et exacte [5]. Nous allons dans un premier temps montrer comment réduire la complexité spatiale pour les multiplications rapides du type Strassen–Winograd. Notre objectif consiste à les rendre *en place*, c'est-à-dire de ne pas utiliser plus de mémoire qu'il n'en faut pour stocker les entrées et les sorties. Puis nous exposerons des méthodes pour produire de nouvelles formules sous-cubiques.

Nous commençons par rappeler quelques résultats existants sur l'algorithme de Strassen [Str69], algorithme de multiplication matricielle qui a ouvert la voie aux complexités sous-cubiques en utilisant seulement le minimum de 7 multiplications ([HK71, Win71]) au lieu des 8 pour produit le naïf des matrices de taille 2×2.

L'algorithme de Strassen et ses variantes.

Nous commençons par présenter divers algorithmes qui sont des variantes de l'algorithme de Strassen, tout d'abord pour le produit standard $C = AB$ puis pour le produit avec accumulation $C \leftarrow C+AB$. Mais avant d'aller plus loin, fixons une notation pour le découpage de matrices. Soit A une matrice de dimensions paires. On la découpe alors en blocs A_{ij} de taille moitié de la manière suivante :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

Algorithme 0.1 : L'algorithme de Strassen.

10 pré-additions

$S_1 \leftarrow A_{12} - A_{22}$ $\quad T_1 \leftarrow B_{21} + B_{22}$
$S_2 \leftarrow A_{11} + A_{22}$ $\quad T_2 \leftarrow B_{11} + B_{22}$
$S_3 \leftarrow A_{11} - A_{21}$ $\quad T_3 \leftarrow B_{11} + B_{12}$
$S_4 \leftarrow A_{11} + A_{12}$ $\quad T_4 \leftarrow B_{12} - B_{22}$
$S_5 \leftarrow A_{21} + A_{22}$ $\quad T_5 \leftarrow B_{21} - B_{11}$

7 multiplications

$P_1 \leftarrow S_1 \times T_1$ $\quad P_5 \leftarrow A_{11} \times T_4$
$P_2 \leftarrow S_2 \times T_2$ $\quad P_6 \leftarrow A_{22} \times T_5$
$P_3 \leftarrow S_3 \times T_3$ $\quad P_7 \leftarrow S_5 \times B_{11}$
$P_4 \leftarrow S_4 \times B_{22}$

8 post-additions

$U_1 \leftarrow P_1 + P_2$ $\quad U_3 \leftarrow P_2 - P_3$
$U_2 \leftarrow U_1 - P_4$ $\quad U_4 \leftarrow U_3 + P_5$
$C_{11} \leftarrow U_2 + P_6$ $\quad C_{22} \leftarrow U_4 - P_7$
$C_{12} \leftarrow P_4 + P_5$ $\quad C_{21} \leftarrow P_6 + P_7$

Algorithme 0.2 : L'algorithme de Winograd.

8 pré-additions

$S_1 \leftarrow A_{21} + A_{22}$ $\quad T_1 \leftarrow B_{12} - B_{11}$
$S_2 \leftarrow S_1 - A_{11}$ $\quad T_2 \leftarrow B_{22} - T_1$
$S_3 \leftarrow A_{11} - A_{22}$ $\quad T_3 \leftarrow B_{22} - B_{12}$
$S_4 \leftarrow A_{12} - S_2$ $\quad T_4 \leftarrow T_2 - B_{21}$

7 multiplications

$P_1 \leftarrow A_{11} \times B_{11}$ $\quad P_5 \leftarrow S_1 \times T_1$
$P_2 \leftarrow A_{12} \times B_{21}$ $\quad P_6 \leftarrow S_2 \times T_2$
$P_3 \leftarrow S_4 \times B_{22}$ $\quad P_7 \leftarrow S_3 \times T_3$
$P_4 \leftarrow A_{22} \times T_4$

7 post-additions

$C_{11} \leftarrow P_1 + P_2$ $\quad C_{12} \leftarrow U_4 + P_3$
$U_2 \leftarrow P_1 + P_6$ $\quad C_{21} \leftarrow U_3 - P_4$
$U_3 \leftarrow U_2 + P_7$ $\quad C_{22} \leftarrow U_3 + P_5$
$U_4 \leftarrow U_2 + P_5$

Les algorithmes pour le produit $C \leftarrow AB$.

Présentons tout d'abord l'algorithme de Strassen (algorithme 0.1). Cet algorithme a été amélioré trois ans plus tard par Winograd [Win71] (algorithme 0.2) en réduisant le nombre d'additions de 18 à 15, le minimum ([Pan84]). Faisons rapidement un calcul de complexité. Le nombre d'opérations dans l'algorithme de Strassen (algorithme 0.1) satisfait à la récurrence $S(n) = 7S\left(n/2\right) + 18\left(n/2\right)^2$, qui, avec la condition initiale $S(1) = 1$, donne :

$$S(n) = 7n^{\omega} - 6n^2 \qquad \text{avec} \qquad \omega = \log_2(7) = \ln(7)/\ln(2).$$

Nous remarquons [6] que réduire le nombre d'additions permet de diminuer la constante principale, dans l'algorithme de Winograd à $6n^{\omega} - 5n^2$.

Tout récemment, une autre amélioration consistant à chercher des formules « symétriques » a été publiée par Marco Bodratto ([Bod10]). Son algorithme, présenté dans l'algorithme 0.3, utilise une phase de pré-calculs identiques pour les variables S et T. Tout en gardant le nombre minimal d'opérations, cette écriture est plus simple et elle permet éventuellement la parallélisation des pré-calculs et la possibilité d'améliorer le calcul des puissances de matrices.

Notons aussi que les algorithmes du type Strassen sont tout à fait utilisables en pratique. Les premiers à s'être appliqués à décrire précisément des implantations de l'algorithme de Winograd sont [HLJJ+96a]. Leur rapport technique a beaucoup influencé les exemples de la partie suivante ainsi que plusieurs des chapitres suivants. Parmi les autres références pour la mise en pratique cet algorithme, nous pouvons aussi citer [Bai88, HLJJ+96b, DHSS94].

Remarques. Il existe de meilleures complexités asymptotiques pour la multiplication de matrices, notamment, la meilleure connue est $\mathcal{O}\left(n^{2.376}\right)$ dans [CW90] (*cf.* aussi [BP94, CBS97]). Cependant, c'est actuellement l'algorithme de Strassen–Winograd qui est le plus efficace en pratique et qui est donc le plus souvent implanté.

5 — Les matrices ont pour coefficient des éléments de corps « à mémoire fixe », par exemple des éléments de $\mathbf{F}_p$ avec p de la taille d'un mot machine.

6 — Dans le reste de ce manuscrit, sauf mention contraire, nous utiliserons la notation $\omega = \log_2(7)$.

Algorithme 0.3 : L'algorithme de Bodrato.

pré-additions symétriques

$$\begin{array}{ll} S_1 \leftarrow A_{22} + A_{12} & T_1 \leftarrow B_{22} + B_{12} \\ S_2 \leftarrow A_{22} - A_{21} & T_2 \leftarrow B_{22} - B_{21} \\ S_3 \leftarrow S_2 + A_{12} & T_3 \leftarrow T_2 + B_{12} \\ S_4 \leftarrow S_3 - A_{11} & T_4 \leftarrow T_3 - B_{11} \end{array}$$

$$\sim$$

$$\begin{array}{ll} P_1 \leftarrow S_1 \times T_1 & P_5 \leftarrow A_{12} \times B_{21} \\ P_2 \leftarrow S_2 \times T2 & P_6 \leftarrow S_4 \times B_{12} \\ P_3 \leftarrow S_3 \times T3 & P_7 \leftarrow A_{21} \times T4 \\ P_4 \leftarrow A_{11} \times B_{11} & \end{array}$$

$$\sim$$

$$\begin{array}{ll} U_1 \leftarrow P_3 + P_5 & U_2 \leftarrow P_1 - U_1 \\ U_3 \leftarrow U_1 - P_2 & \\ C_{11} \leftarrow P_4 + P_5 & C_{12} \leftarrow U_3 - P_6 \\ C_{21} \leftarrow U_2 - P_7 & C_{22} \leftarrow P_2 + U_2 \end{array}$$

Produit avec accumulation $C \leftarrow C + AB$.

Il est souvent utile de calculer le produit plus général $C = C + AB$, que nous appellerons dans la suite *produit avec accumulation*. En effet, ce produit permet d'effectuer en même temps qu'une multiplication une opération (addition) supplémentaire. Généralement, il est plus rapide en pratique d'utiliser ce genre d'opérations (on trouve par exemple `fma` (*fused add-multiply*) dans le fichier standard `math.h`, ou `axpy` ($a \times x + y$) dans Givaro ou LinBox). Dans le tableau suivant, nous montrons à titre d'exemple les temps d'exécution pour les opérations '`C=AB`', '`C=C+AB`' et '`D=AB;C=C+D`' en utilisant la fonction `dgemm` sur des matrices 3 000×3 000. Ce tableau montre bien l'utilité de fusionner une addition avec une multiplication

Tab. 0.1 — Comparaison des temps (secondes) mis pour divers produits matriciels $3\,000 \times 3\,000$ utilisant `dgemm` (GotoBLAS2-1.13) sur Palo-Alto@ujf

opération	`C=AB`	`C+=AB`	`D=AB` `C+=D`
temps (1 cœur)	4,45	4,42	4,52
temps (8 cœurs)	0,59	0,57	0,66

quand cela est possible. Une explication à cela est que l'addition est aussi rapide que la vitesse du bus (très lent) alors que dans `dgemm` , l'addition supplémentaire s'effectue au pire à la vitesse de la mémoire RAM : c'est un problème de localité des données.

Nous rappelons maintenant l'algorithme de [HLJJ+96b, fig. 6] (algorithme 0.4) pour le produit avec accumulation. Cet algorithme utilise 7 multiplications et 19 additions. Bien sûr, certaines de ces multiplications/additions peuvent être regroupées en un appel récursif. D'ailleurs, ils proposent un ordonnancement performant sur lequel nous reviendrons longuement dans le chapitre 1.

Implantations dans les logiciels de calcul exact.

La bibliothèque FFLAS-FFPACK[7] [Per01, DGP02, DGP04, DGP08] implémente efficacement l'algorithme de Winograd dans sa routine `fgemm` sur les corps $\mathbf{Z}/p\mathbf{Z}$ avec p plus petit qu'un mot machine. Notamment, cette routine est très efficace sur les corps finis premiers représentés par des doubles et avec en général $p < 2^{23}$. Cette routine est une des pierres angulaires de FFLAS-FFPACK.

7 — Disponible sur `http://linalg.org/projects/fflas-ffpack`.

Algorithme 0.4 : L'algorithme $C \leftarrow AB + C$ de [HLJJ+96b].

S_i et T_i comme dans l'algorithme 0.2.

$\approx$

P_i comme dans l'algorithme 0.2.

$\approx$

$$\begin{array}{ll} C_{12} \leftarrow C_{12} + P_5 & C_{22} \leftarrow C_{22} + P_5 \\ U_2 \leftarrow P_1 + P_6 & U_3 \leftarrow U_2 + P_7 \\ C_{11} \leftarrow C_{11} + P_1 + P_2 & C_{12} \leftarrow C_{12} + P_3 + U_2 \\ C_{21} \leftarrow C_{21} + U_3 - P_4 & C_{22} \leftarrow C_{22} + U_3 \end{array}$$

L'algorithme de Bodrato a été implémenté par lui-même dans m4ri en remplacement de celui de Winograd, apportant un gain tangible de performance (1% d'après la page personnelle de l'auteur [8]). Il est aussi implémenté dans les algorithmes de multiplications rapides génériques de LinBox (*cf.* section 5.2).

Plan de la première partie.

Cette partie se divise autour de deux problèmes liés à l'algorithme de Strassen.

✦ Nous cherchons d'abord à minimiser l'utilisation de la mémoire (chapitre 1). Nous y développons un *jeu de galet* (section 1.1) qui nous aide à trouver de nouveaux ordonnancements (section 1.2) utilisant le moins possible de mémoire supplémentaire tout en restant performants. Pour parvenir à nos fins, nous combinerons diverses techniques :

— nous cherchons des ordonnancements qui autorisent d'écraser des opérandes ;
— nous combinons finement des produits standards `C=AB` avec des produits avec accumulation `C+=AB` ;
— nous modifions des algorithmes et en construisons de nouveaux en rajoutant par exemple des pré-additions ;
— nous créons des ordonnancements hybrides qui combinent divers algorithmes (classiques, rapides, en place...).

✦ Nous cherchons ensuite à améliorer l'exposant ω (chapitre 2). Nous mettons en œuvre des méthodes numériques pour tenter d'obtenir (semi-)automatiquement de nouveaux algorithmes pour des tailles de matrices plus générales.

8 — *cf.* http://bodrato.it/software/strassen.html.

Chap. 1 Amélioration du placement en mémoire pour la multiplication de matrices.

Sommaire

1.1 Meilleur placement en mémoire : un jeu de galet. 16
1.1.1 Présentation générale du problème. 16
1.1.2 Le jeu de galet. 17
1.1.3 Implantation du jeu de galet. 18
1.1.4 Conclusion et prolongements. 21
1.2 Ordonnancements efficaces en mémoire. 21
1.2.1 Ordonnancements avec entrées constantes : une première approche. . . . 22
1.2.2 Ordonnancements avec écrasement des opérandes : cas carré. 27
1.2.3 Ordonnancements avec entrées constantes : cas carré. 32
1.2.4 Cas général : ordonnancements hybrides. 32
1.2.5 Synthèse. 36
1.2.6 Implantation des nouveaux ordonnancements. 36
1.3 Prolongements. 36

La multiplication naïve de matrices n'utilise pas d'allocations en plus de celles de ses entrées et sorties (opération dite *en place*). Par contre, même les meilleurs ordonnancements connus de multiplication rapide de matrice en utilisent. D'autre part, multiplier deux matrices de taille $10\,000 \times 10\,000$ sur un corps représentable sur un mot machine peut s'effectuer en environ 200 secondes sur un processeur Intel Core 2 duo avec 4Go de mémoire RAM. Par contre, la multiplication de matrices à peine plus grosses sur la même architecture avec l'algorithme de Winograd (algorithme 0.2, ordonnancement du tableau 1.1, p. 22) jusqu'au dernier niveau de récursion, ne tient pas en mémoire.

Ces remarques mènent au problème : « dans quelle mesure peut-on limiter voire supprimer l'usage de cette mémoire supplémentaire ? » Ce chapitre s'attache à répondre à cette question et à proposer des implantations efficaces à la fois en complexité arithmétique et spatiale. Nous allons exposer un algorithme (jeu de galet) qui permet de créer des ordonnancements pour un algorithme de multiplication et des contraintes données. Notamment nous allons autoriser des algorithmes qui écrasent leurs entrées. C'est l'objet de la section 1.1. Nous expliquons d'abord en quoi un jeu de galet nous permet de créer de nouveaux algorithmes qui répondent à nos attentes, puis nous définissons nos règles du jeu et finalement montrons comment nous pouvons produire une implantation générique — i.e. réutilisable sur d'autres jeux, extensible à d'autres règles avec un minimum de code à fournir.

Nous allons aussi créer dans la section 1.2 de nouveaux algorithmes à l'aide de diverses techniques (utilisation de pré-additions, première étape de calcul classique sur des blocs, jonglage entre le nombre d'appels récursifs et des appels à des fonctions spécialisées...). Nous commençons par présenter les ordonnancements existants et nous exhibons de nouveaux algorithmes et ordonnancements qui nous permettent de les améliorer. Nous étudions le cas des multiplications et accumulations qui écrasent un ou deux opérandes ; avec des calculs complets de complexités et une recombinaison fine des divers ordonnancements, nous parvenons à créer des ordonnancements efficaces dont les entrées sont constantes. Nous concluons par un tableau récapitulatif.

1.1 Meilleur placement en mémoire : un jeu de galet.

Nous allons développer dans cette section un algorithme qui permet de trouver les meilleurs ordonnancements en terme de mémoire pour une formule de multiplication rapide donnée. Ordonnancer des tâches est un problème ancien en informatique et particulièrement pour les compilateurs ou les systèmes d'exploitation. C'est en général un problème NP-complet ([Set73]) cependant, pour de petites tailles, nous pouvons utiliser des techniques de force brute. L'idée initiale pour trouver et certifier des ordonnancements se retrouve dans [HLJJ+96b]. Nous avons adapté leurs techniques à nos besoins en modifiant les règles du jeu classique.

1.1.1 Présentation générale du problème.

Nous représentons un ordonnancement par un graphe acyclique orienté. Les nœuds *initiaux* sont les données de l'algorithme, les nœuds *terminaux* ses résultats. Un nœud peut à la fois être initial et terminal. En chaque nœud s'effectue une opération, les parents étant les opérandes (possiblement ordonnés). Un jeu de galet représente un espace mémoire. Au début du jeu, tous les nœuds initiaux sont alloués, donc possèdent un galet.

Voici par exemple le graphe qui représente [1] l'algorithme de Winograd (figure 1.1). Dans la figure 1.2,

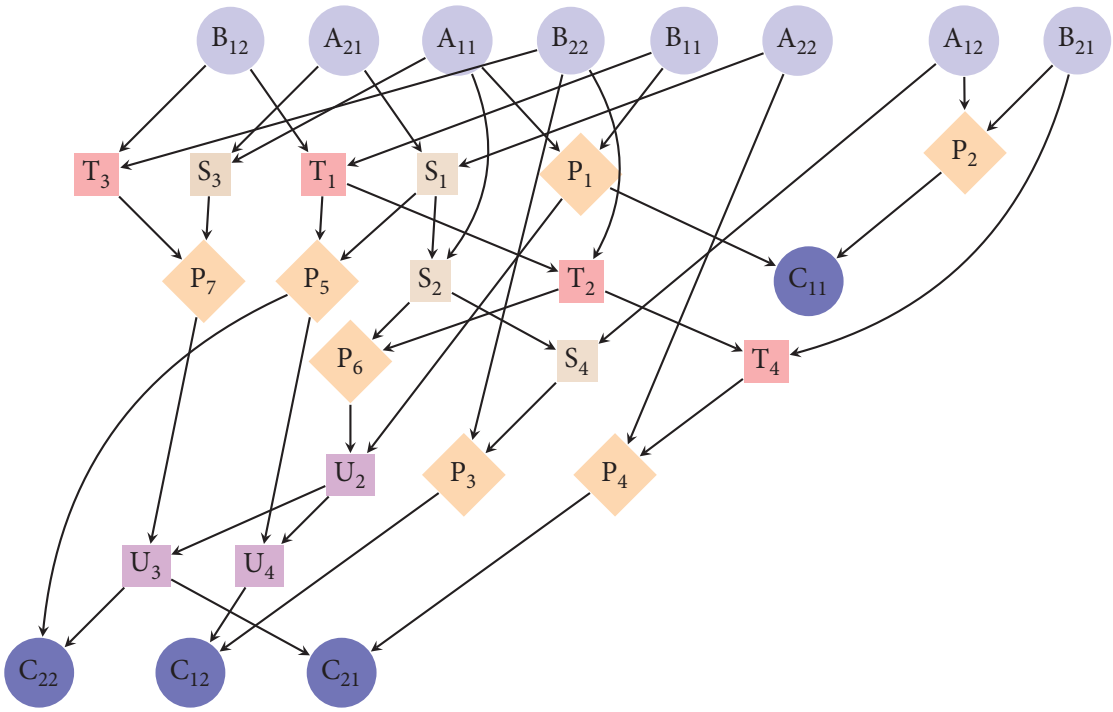

Fig.. 1.1 — Arbre des dépendances dans l'algorithme de Winograd.

1 — Les losanges correspondent aux multiplications, les cercles aux états finaux ou initiaux, les carrés aux autres variables intermédiaires. Le bleu foncé est utilisé pour mettre en avant les nœuds finaux correspondant à C.

nous représentons [2] l'arbre des dépendances pour le nouvel algorithme d'accumulation (algorithme 1.2, voir p.ex. page 24), tandis que dans la figure 1.3, nous représentons le graphe de l'algorithme utilisé par exemple pour l'ordonnancement du tableau 1.13 (voir page 31).

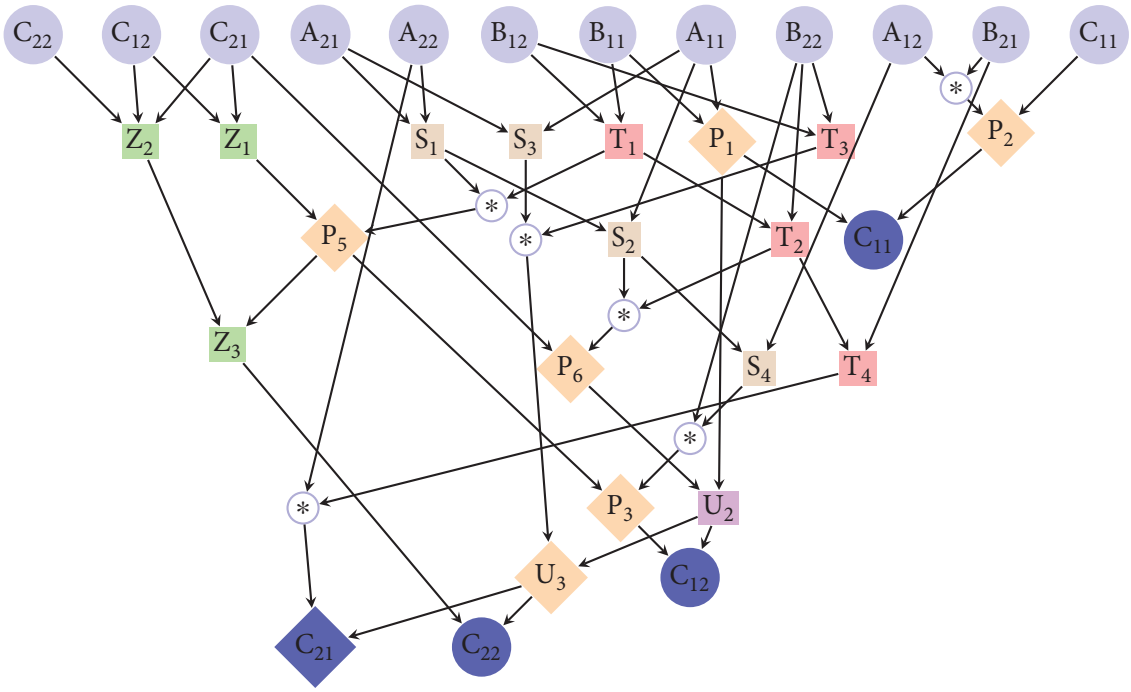

Fig.. 1.2 — Arbre des dépendances pour $C \leftarrow \alpha AB + \beta C$ avec 6 accumulations.

Nous rappelons que nous cherchons à ordonnancer les opérations de ces algorithmes de manière à utiliser le moins possible de mémoire temporaire. Pour y parvenir, l'idée dans [HLJJ+96b] est de représenter un espace mémoire par un galet et de trouver une suite d'opérations (de nœuds) qui minimise l'utilisation de ces galets. Un ordonnancement correspondra à la suite des opérations sur les galets dans le graphe des dépendances, jusqu'à ce que tous les nœuds terminaux en possèdent un. Le cas contraire — aucun ordonnancement trouvé après une recherche exhaustive — certifie qu'il n'existe pas d'ordonnancements qui répondent aux règles fixées, au nombre de galets et à l'arbre donnés. Définissons maintenant plus précisément ce jeu de galets.

1.1.2 Le jeu de galet.

Intuitivement, nous nous autorisons à déplacer un galet sur l'arbre (effectuer une opération en place dans cette mémoire), le supprimer (libérer la mémoire correspondante) ou en ajouter un (allouer de la mémoire). C'est sur ces idées que [HLJJ+96b] ont créé les règles du jeu de galet (*pebble game*). Nous précisons donc les règles du jeu pour le jeu classique — à la différence notable où, pour nous, des galets sont initialement placés sur les nœuds initiaux.

Suppression. Un galet peut être supprimé de tout nœud non initial (i.e. un espace mémoire est libéré).

Ajout. Si tous les parents d'un nœud ont un galet, alors un galet peut être placé sur ce nœud (i.e. une opération peut être effectuée lorsque opérandes et résultat sont alloués).

Déplacement. Si tous les prédécesseurs d'un nœud sans galet ont un galet et que l'opération est une addition ou soustraction, alors le galet d'un parent qui n'est pas initial peut être placé sur ce nœud (i.e. cette opération peut être effectuée en place dans la mémoire correspondant au galet déplacé).

Le jeu s'arrête lorsque tous les nœuds terminaux ont un galet. La suite des opérations effectuées constitue alors un *ordonnancement* pour l'algorithme joué.

2 — Ici, l'étiquette $*$ dans un nœud indique la multiplication dans le produit avec accumulation.

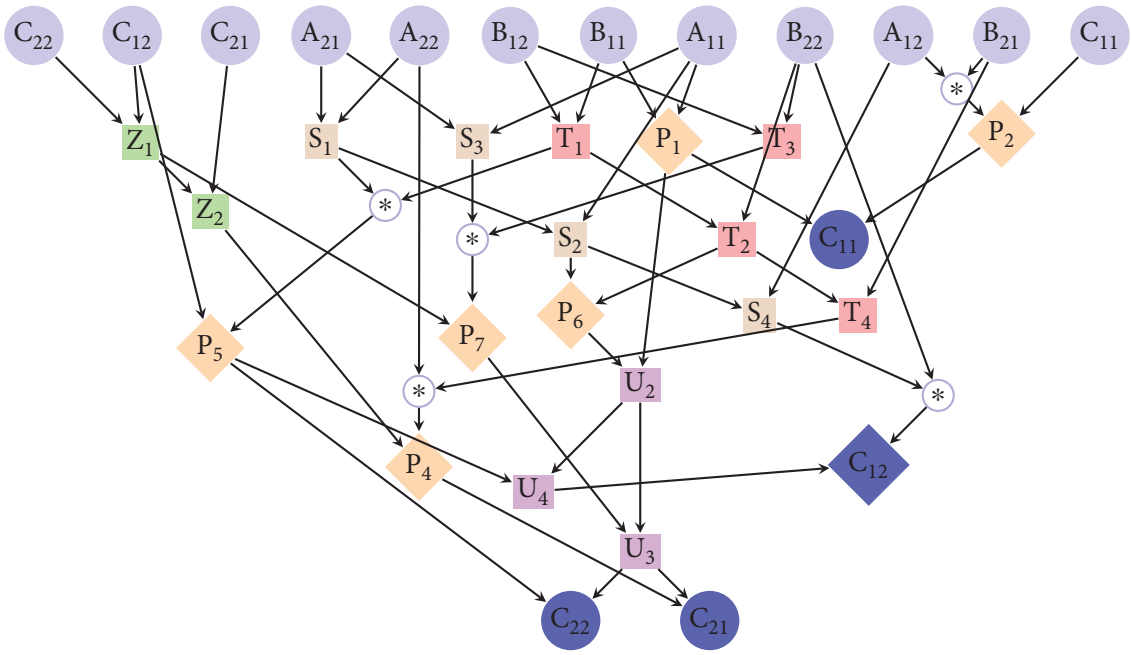

Fig.. 1.3 — Arbre des dépendances pour $C \leftarrow \alpha AB + \beta C$ avec 5 accumulations.

Nous allons maintenant complexifier ce jeu pour ajouter de nouvelles contraintes. Nous décrivons ainsi quatre nouvelles règles qui relaxent un peu les hypothèses classiques : il peut être autorisé de rajouter des opérations, d'altérer l'arbre, de jouer avec des galets non homogènes (spécifier des contraintes sur les zones mémoires).

- Tout d'abord, nous remarquons que laisser les galets initiaux fixes correspond à conserver les entrées constantes au cours du jeu. Nous pouvons cependant autoriser des galets initiaux à être déplacés, supprimés, voire écrasés (i.e. des données de l'algorithme sont modifiées lors de son exécution, il n'est plus « à entrées constantes »).
- Un galet peut être dupliqué (copié/restauré). Algorithmiquement, cela revient à effectuer une opération quadratique supplémentaire.
- Un galet peut avoir des attributs : par exemple la taille de la mémoire qu'il représente, un drapeau...
- Une variante d'une opération peut être sélectionnée sur un nœud. Cela permet notamment de pouvoir choisir entre divers algorithmes (p.ex. entre des multiplications ou accumulations qui peuvent ou non écraser leurs entrées) mais cela permet aussi de pouvoir décomposer certaines opérations (p.ex., un produit avec accumulation peut devenir une multiplication suivie d'une addition).

Cet algorithme a été implanté une première fois en mettant l'accent sur la généralité dans les possibilités sur les opérations et les galets. Il était difficilement maintenable. Il a donc été totalement réécrit dans un souci de *généricité* d'une part et de meilleure *efficacité* et de *simplicité* d'autre part.

1.1.3 Implantation du jeu de galet.

L'algorithme va rechercher récursivement (par force brute) s'il est possible de créer un ordonnancement avec des conditions initiales prescrites (par exemple le nombre de galets autorisés). Nous montrons dans l'algorithme 1.1 la structure simple de cet algorithme. Le programme l'implantant s'appelle galet, son code source est bien sûr libre (licence GPLv2+) et il est disponible à l'adresse `https://forge.imag.fr/projects/galet/`. L'implantation que nous avons cherchée à créer devait rendre ce jeu de galet le plus *générique* possible.

En effet, faisons quelques instants une digression sur un jeu des galets bicolores ([HK81]) dont le traitement par galet m'a été proposé par [MM09]. Ce jeu modélise des problèmes d'entrée/sortie, par exemple sur une mémoire cache. Des galets bleus représentent une mémoire lente tandis qu'un galet

Algorithme 1.1 : `Explore`(G,$\mathscr{R}$)

Entrées : Un graphe G représentant un algorithme `alg`, un ensemble $\mathscr{R}$ de règles du jeu.
Sorties : un/tout/aucun ordonnancement pour `alg` satisfaisant aux règles.

```
𝒞 ← listeCoupsPossibles(G)
pour tout coup c ∈ 𝒞 faire
    G′ ← joueCoup(c, G)
    si nonExploré(G′) alors
        Explore(G′, ℛ)
    fin
fin
```

rouge est une mémoire rapide. Mettre un galet coloré sur un nœud d'un arbre signifie placer cet élément dans l'espace mémoire correspondant. Le cache étant le plus proche des instructions il est petit, rare, donc le nombre de galets rouges sera très limité. Par contre, le nombre de galets bleus peut être infini. Le jeu est terminé quand tous les éléments finaux ont un galet bleu. Les règles du jeu sont alors les suivantes :

Initialisation. Un galet bleu peut être placé sur un nœud initial à n'importe quel moment.

Entrée. Un galet rouge peut être placé sur un nœud ayant un galet bleu.

Sortie. Un galet bleu peut être placé sur un nœud contenant un galet rouge.

Calcul. Un galet rouge peut être placé sur un nœud si tous ses parents ont un galet rouge.

Suppression. Un galet peut être retiré de n'importe quel nœud.

Ce jeu de galet bleu–rouge est très similaire au nôtre dans sa formulation : nous cherchons donc à factoriser un maximum de code pour pouvoir traiter également ces deux exemples (*cf.* la sous-section 1.1.3.1 suivante), l'idéal étant de ne proposer à coder, pour chacun, que des règles et une description des galets et ce de manière simple (*cf.* la sous-section 1.1.3.2). Nous terminons maintenant avec la présentation de galet par la description de ses parties génériques et spécifiques.

1.1.3.1 La partie générique.

Pour proposer une vue d'ensemble, le logiciel galet est constitué d'en-têtes C++ structurés en classes pour chaque objet, aux noms évocateurs [3] : `Tree`, `Node`, `Pebble` et `Move`. La partie non générique se retrouve dans un dossier `./games/<+game_name+>/`. La question qui se pose alors est : parmi toutes ces classes et leurs membres, qu'est ce qui est générique ? Toutes les fonctions suivantes sont tout à fait génériques.

- Comme son nom l'indique, la classe `Tree` implante la structure d'arbre et les opérations sur les galets. La majorité de ses fonctions membres peut être factorisée. La classe `Node` décrit génériquement un nœud dont `Tree` est composé.
- Une fonction `Explore` explore récursivement un arbre. Elle permet de rechercher tous les ordonnancements possibles ou s'arrête au premier trouvé. Nous notons que pour éviter de rentrer dans une boucle infinie (p.ex. en situation de boucle, lorsque des opérations successives ramènent à une position déjà explorée), chaque arbre est haché, c'est ce qui correspond à la fonction `nonExploré` de l'algorithme 1.1. En outre, pour limiter le nombre d'ordonnancements à traiter en sortie, il peut être décidé que la fonction `joueCoup` propage tous les coups qui ne nécessitent pas de décision. Finalement, il est tenu à jour une liste des nœuds sur lesquels une opération peut être effectuée, ce qui permet à `listeCoupsPossibles` de ne tester qu'un nombre limité de nœuds et d'opérations.

3 — Contrairement à l'erreur faite dans la première mouture de galet, l'anglais est choisi pour rendre les noms de classes, fonctions, *etc.* encore plus évocateur...

- Les entrées (un graphe) et sorties (l'ordonnancement) se font dans des fichiers au format XML (et en utilisant la petite bibliothèque tinyxml [4]. En particulier, ce format les rend facile à lire, transformer. C'est notamment utile pour traiter les ordonnancements en sortie.
- De plus, il est possible de visualiser à tout moment le graphe ou de suivre son avancement dans des fichiers `.dot` grâce à Graphviz [5].

Nous concluons ce paragraphe par un exemple de graphe (simple) qui représente la suite d'opérations `S3=S1+S2; T2=T1*S3;` reproduit dans le code 1.1.

```
<?xml version="1.0" ?>
<Tree>
     <NodeList size="5" depth="2">
          <Node name="S1">
               <Property initial="true" terminal="true"/>
          </Node>
          <Node name="S2">
               <Property initial="true" terminal="true"/>
          </Node>
          <Node name="S3">
               <Property initial="true"/>
          </Node>
          <Node name="T1">
               <ParentList ordered="true">
                    <Parent node="S1"/>
                    <Parent node="S2"/>
               </ParentList>
               <Operation name="mul"/>
          </Node>
          <Node name="T2">
               <ParentList>
                    <Parent node="T1"/>
                    <Parent node="S3"/>
               </ParentList>
               <Operation name="add"/>
               <Property terminal="true"/>
          </Node>
     </NodeList>
</Tree>
```

Code 1.1 — Exemple d'arbre en entrée de galet.

1.1.3.2 La partie spécifique.

La partie non générique — et donc le travail à apporter lorsque l'on crée de nouvelles règles — se réduit à :

— les règles (détection/exécution d'un coup possible) ;
— la classe `Pebble` qui décrit les caractéristiques d'un galet ;
— la classe `Move` (quelques lignes) qui énumère seulement les noms des actions.

Cette partie générique est donc réduite de manière satisfaisante à une portion congrue [6].

4 — Disponible sur `http://www.grinninglizard.com/tinyxml/index.html` et dans toutes les bonnes distributions.

5 — Disponible sur `http://www.graphviz.org/`, convertit des fichiers au format `dot` en fichiers au format PDF en particulier.

6 — Leur implantation se fait sous un répertoire `games/mygame/` du source. Un fichier d'en-tête `games/mygame.h`, incluant les classes d'arbre générique et de galet, règles, opérations correspondantes spécifiques, est ensuite simplement inclu avant `Explorer.h`.

1.1.4 Conclusion et prolongements.

Nous avons donc réussi à réduire la partie spécifique au codage minimum pour nos besoins. Cet algorithme a été utilisé avec succès pour trouver de nouveaux ordonnancements, exposés dans la section 1.2 suivante. Il existe cependant quelques idées et techniques qui n'ont pas encore été introduites dans galet. Parmi les tâches faciles à implanter et qui ne demandent que du temps, il manque une véritable documentation au projet, une fonction qui génère automatiquement du code C++ à partir d'un ordonnancement et une batterie de tests. Parmi les tâches plus ardues, cet algorithme ne fait pas de réécriture de graphe (notamment, ajouter des pré-additions ou savoir extraire un produit avec accumulation). Nous pouvons aussi imaginer de le coupler avec des méthodes symboliques. Cet algorithme explore rapidement les graphes du type Strassen–Winograd (figure 1.1, p. 16) mais n'a pas été testé sur des arbres plus importants.

Les ordonnancements de tâches ont été naturellement beaucoup étudiés dans le cas des systèmes d'exploitation et des compilateurs par exemple et aussi dans le cadre du parallélisme (de tâches, d'instructions *cf.* [TOU02]). Nous pourrions tirer des idées à partir de leurs techniques, tout en gardant à l'esprit que notre problème est aussi et surtout un problème de minimisation et de certification.

1.2 Ordonnancements efficaces en mémoire.

Dans cette section, nous allons rappeler certains ordonnancements et en créer de nouveaux pour les produits standards ou avec accumulation. L'objectif est de réduire au maximum, voire de supprimer tout recours à de la mémoire supplémentaire. Pour arriver à nos fins, nous allons créer toute une série de nouveaux ordonnancements qui écrasent ou non certains de leurs opérandes et nous allons ensuite les recombiner entre eux. Nous allons voir que si nous gagnons en complexité spatiale, nous perdons très peu en terme de complexité algorithmique.

Plus précisément, dans la section 1.2.1, nous rappelons les ordonnancements classiques pour les produits qui n'écrasent pas leurs entrées et nous en proposons de nouveaux avec une meilleure complexité. Ensuite, dans les sections 1.2.2 et 1.2.3, nous nous restreignons au cas des tailles en entrée carrées. Nous nous autorisons ou non à écraser certains opérandes, cela nous permet de créer des cas de base aux caractéristiques spécifiques pour d'autres algorithmes plus généraux. Finalement, dans la section 1.2.4, nous combinons tous les résultats précédents avec des techniques hybrides pour parvenir à des ordonnancements très efficaces, ce que nous montrerons par la suite dans l'implantation (section 1.2.6). Un tableau de synthèse regroupe dans la section 1.2.5 les résultats de complexité des ordonnancements et algorithmes présentés dans cette section.

Nous allons représenter les ordonnancements trouvés dans des tableaux, comme par exemple le tableau 1.1 ci-après, dont la lecture, par colonne, indique l'ordre dans lequel les appels sont faits (n°), l'opération algorithmique effectuée et la variable algorithmique dans laquelle cette dernière est effectuée (loc.). À la suite de chaque ordonnancement, nous allons calculer diverses complexités, soit, dans l'ordre :

— la complexité spatiale (mémoire supplémentaire maximale nécessaire), notée E ;
— la complexité algorithmique (le nombre d'opérations), notée W ;
— le nombre d'allocations nécessaires (la taille cumulée de la mémoire utilisée), noté A.

En outre, dans les tableaux suivants, la notation $\mathtt{alg}(A_{ij}B_{ij})$ fait appel à l'algorithme noté `alg` pour effectuer l'opération $A_{ij}B_{ij}$. De la même manière, les complexités pour cet algorithme seront notées $E_{\mathtt{alg}}$, $W_{\mathtt{alg}}$ ou $A_{\mathtt{alg}}$. Par similarité avec les notations de [HLJJ+96b], la matrice C se découpe comme suit :

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} U_1 & U_5 \\ U_6 & U_7 \end{pmatrix}.$$

Finalement, les tailles de A et B sont respectivement $m \times k$ et $k \times n$.

1.2.1 Ordonnancements avec entrées constantes : une première approche.

Dans un premier temps, nous rappelons le cas classique de la multiplication standard avec entrées constantes (sous-section 1.2.1.1) puis nous considérons le cas du produit avec accumulation (sous-section 1.2.1.2) et enfin proposons notamment, dans la sous-section 1.2.1.3, un nouvel algorithme de produit avec accumulation et des pistes pour améliorer les résultats existants de ces deux premières sections, que nous développerons ensuite (section 1.2.2 et suivantes).

1.2.1.1 Multiplication standard.

Nous considérons donc d'abord l'opération $C \leftarrow A \times B$ pour laquelle [DHSS94] propose l'ordonnancement `wino` (*cf.* tableau 1.1). Comme [HLJJ+96b] le montre, il n'est pas possible pour cet ordonnancement d'utiliser moins de deux temporaires. Faisons les calculs de complexité.

Tab. 1.1 — Ordonnancement `wino` de Winograd pour $C \leftarrow A \times B$, avec deux temporaires

n°	opération	loc.	n°	opération (suite)	loc.
1	$S_3 = A_{11} - A_{21}$	X	12	$P_1 = \texttt{wino}\left(A_{11}B_{11}\right)$	X
2	$T_3 = B_{22} - B_{12}$	Y	13	$U_2 = P_1 + P_6$	C_{12}
3	$P_7 = \texttt{wino}\left(S_3T_3\right)$	C_{21}	14	$U_3 = U_2 + P_7$	C_{21}
4	$S_1 = A_{21} + A_{22}$	X	15	$U_4 = U_2 + P_5$	C_{12}
5	$T_1 = B_{12} - B_{11}$	Y	16	$\mathbf{U_7} = U_3 + P_5$	C_{22}
6	$P_5 = \texttt{wino}\left(S_1T_1\right)$	C_{22}	17	$\mathbf{U_5} = U_4 + P_3$	C_{12}
7	$S_2 = S_1 - A_{11}$	X	18	$T_4 = T_2 - B_{21}$	Y
8	$T_2 = B_{22} - T_1$	Y	19	$P_4 = \texttt{wino}\left(A_{22}T_4\right)$	C_{11}
9	$P_6 = \texttt{wino}\left(S_2T_2\right)$	C_{12}	20	$\mathbf{U_6} = U_3 - P_4$	C_{21}
10	$S_4 = A_{12} - S_2$	X	21	$P_2 = \texttt{wino}\left(A_{12}B_{21}\right)$	C_{11}
11	$P_3 = \texttt{wino}\left(S_4B_{22}\right)$	C_{11}	22	$\mathbf{U_1} = P_1 + P_2$	C_{11}

Complexités de l'ordonnancement `wino`.

✦ Une étude attentive de chaque étape de cet ordonnancement montre que l'ordonnancement `wino` requiert deux blocs temporaires X et Y dont les dimensions sont respectivement $m/2 \times \max\left(k/2, n/2\right)$ et $k/2 \times n/2$. La mémoire supplémentaire utilisée vérifie donc la relation :

$$E_{\texttt{wino}}(m,k,n) = m/2 \max\left(k/2, n/2\right) + k/2\, n/2 + E_{\texttt{wino}}\left(m/2, k/2, n/2\right).$$

En sommant ces allocations de blocs temporaires jusqu'au dernier niveau de récursion, on obtient une mémoire supplémentaire (où $M = \min\{m,k,n\}$) :

$$\begin{aligned} E_{\texttt{wino}}(m,k,n) &= \sum_{i=1}^{\log_2(M)} \frac{1}{4^i}\left(m \max(k,n) + kn\right) \\ &= \frac{1}{3}\left(1 - \frac{1}{M^2}\right)\left(m \max(k,n) + kn\right) \\ &< 1/3\left(m \max(k,n) + kn\right). \end{aligned} \tag{1.1}$$

De la même manière, on prouve le lemme suivant :

Lemme 1.1. — Soient m, k et n des puissances de deux, $g(x, y, z)$ une fonction homogène, $\mathrm{M} = \min\{m, k, n\}$ et $f(m, k, n)$ une fonction telle que

$$f(m,k,n) = \begin{cases} g\left(m/2, k/2, n/2\right) + f\left(m/2, k/2, n/2\right) & \text{si } m, n \text{ et } k > 1\,; \\ 0 & \text{sinon.} \end{cases}$$

Alors $f(m, k, n) = \frac{1}{3}\left(1 - \frac{1}{\mathrm{M}^2}\right) g(m, k, n) < 1/3\, g(m, k, n)$.

En supposant que $m = n = k$, on obtient $\mathrm{E}_{\texttt{wino}}(n, n, n) < 2/3 n^2$, quantité en particulier inférieure à la taille d'un opérande (n^2).

✦ Ici, 7 multiplications et 15 additions sont utilisées, donc la complexité arithmétique du tableau 1.1 est donnée par la relation :

$$\begin{cases} \mathrm{W}_{\texttt{wino}}(n) = 7\mathrm{W}_{\texttt{wino}}\left(n/2\right) + 15\left(n/2\right)^2 \\ \mathrm{W}_{\texttt{wino}}(1) = 1 \end{cases}.$$

On obtient $\mathrm{W}_{\texttt{wino}}(n) = 6n^{\omega} - 5n^2$.

✦ À chaque étape de récurrence, deux temporaires sont alloués. Le nombre total d'allocations satisfait donc à :

$$\begin{cases} \mathrm{A}_{\texttt{wino}}(n) = 2\left(n/2\right)^2 + 7\mathrm{A}_{\texttt{wino}}\left(n/2\right) \\ \mathrm{A}_{\texttt{wino}}(1) = 0 \end{cases},$$

ce qui donne $\mathrm{A}_{\texttt{wino}}(n) = 2/3\left(n^{\omega} - n^2\right)$.

1.2.1.2 Produit avec accumulation.

Pour l'opération plus générale $\mathrm{C} \leftarrow \alpha\mathrm{A} \times \mathrm{B} + \beta\mathrm{C}$, il existe un meilleur ordonnancement que la méthode naïve qui aurait consisté à effectuer $\mathrm{C}' \leftarrow \alpha\mathrm{A} \times \mathrm{B}$ avec le tableau 1.1, puis $\mathrm{C} \leftarrow \mathrm{C}' + \beta\mathrm{C}$, en utilisant donc une mémoire supplémentaire de taille $(1 + 2/3)n^2$ dans le cas carré. En effet, le tableau 1.2 de [HLJJ+96b, fig. 6] ne requiert que trois blocs temporaires pour le même nombre d'opérations (7 multiplications et 4 + 15 additions) et permet comme on va le voir de de limiter la mémoire supplémentaire nécessaire. Le nombre de trois blocs temporaires est, d'après la même référence, minimal pour cet algorithme.

Tab. 1.2 — Ordonnancement `wacc` pour l'opération $\mathrm{C} \leftarrow \alpha\mathrm{A} \times \mathrm{B} + \beta\mathrm{C}$ avec 3 temporaires

n°	opération	loc.	n°	opération (suite)	loc.
1	$\mathrm{S}_1 = \mathrm{A}_{21} + \mathrm{A}_{22}$	X	12	$\mathrm{S}_4 = \mathrm{A}_{12} - \mathrm{S}_2$	X
2	$\mathrm{T}_1 = \mathrm{B}_{12} - \mathrm{B}_{11}$	Y	13	$\mathrm{T}_4 = \mathrm{T}_2 - \mathrm{B}_{21}$	Y
3	$\mathrm{P}_5 = \texttt{wino}\left(\alpha\mathrm{S}_1\mathrm{T}_1\right)$	Z	14	$\mathrm{C}_{12} = \texttt{wacc}\left(\alpha\mathrm{S}_4\mathrm{B}_{22} + \mathrm{C}_{12}\right)$	C_{12}
4	$\mathrm{C}_{22} = \mathrm{P}_5 + \beta\mathrm{C}_{22}$	C_{22}	15	$\mathbf{U}_5 = \mathrm{U}_2 + \mathrm{C}_{12}$	C_{12}
5	$\mathrm{C}_{12} = \mathrm{P}_5 + \beta\mathrm{C}_{12}$	C_{12}	16	$\mathrm{P}_4 = \texttt{wacc}\left(\alpha\mathrm{A}_{22}\mathrm{T}_4 - \beta\mathrm{C}_{21}\right)$	C_{21}
6	$\mathrm{S}_2 = \mathrm{S}_1 - \mathrm{A}_{11}$	X	17	$\mathrm{S}_3 = \mathrm{A}_{11} - \mathrm{A}_{21}$	X
7	$\mathrm{T}_2 = \mathrm{B}_{22} - \mathrm{T}_1$	Y	18	$\mathrm{T}_3 = \mathrm{B}_{22} - \mathrm{B}_{12}$	Y
8	$\mathrm{P}_1 = \texttt{wino}\left(\alpha\mathrm{A}_{11}\mathrm{B}_{11}\right)$	Z	19	$\mathrm{U}_3 = \texttt{wacc}\left(\alpha\mathrm{S}_3\mathrm{T}_3 + \mathrm{U}_2\right)$	Z
9	$\mathrm{C}_{11} = \mathrm{P}_1 + \beta\mathrm{C}_{11}$	C_{11}	20	$\mathbf{U}_7 = \mathrm{U}_3 + \mathrm{C}_{22}$	C_{22}
10	$\mathrm{U}_2 = \texttt{wacc}\left(\alpha\mathrm{S}_2\mathrm{T}_2 + \mathrm{P}_1\right)$	Z	21	$\mathbf{U}_6 = \mathrm{U}_3 - \mathrm{C}_{21}$	C_{21}
11	$\mathbf{U}_1 = \texttt{wacc}\left(\alpha\mathrm{A}_{12}\mathrm{B}_{21} + \mathrm{C}_{11}\right)$	C_{11}			

Complexité de l'algorithme wacc.

✦ Les trois blocs temporaires X, Y, Z ont des dimensions $m/2 \times k/2$, $k/2 \times n/2$ et $m/2 \times n/2$. Puisque les deux blocs temporaires dans l'ordonnancement wino sont plus petits, on a $E_{\texttt{wacc}} \geqslant E_{\texttt{wino}}$. Donc, d'après le lemme 1.1, nous obtenons :

$$E_{\texttt{wacc}}(m,k,n) = \frac{1}{3}\left(1 - \frac{1}{M^2}\right)(mk + kn + mn). \tag{1.2}$$

Avec $m = n = k$, cela donne en particulier $E_{\texttt{wacc}}(n,n,n) < n^2$.

✦ La complexité arithmétique satisfait à la relation :

$$\begin{cases} W_{\texttt{wacc}}(n) = 5W_{\texttt{wacc}}(n/2) + 2W_{\texttt{wino}}(n/2) + 14(n/2)^2 \\ W_{\texttt{wacc}}(1) = 2 \end{cases}. \tag{1.3}$$

Donc $W_{\texttt{wacc}}(n) = 6n^{\omega} - 4n^2$.

✦ Son nombre d'allocations est donné par : $A_{\texttt{wacc}}(n) = 3(n/2)^2 + 5A_{\texttt{wacc}}(n/2) + 2A_{\texttt{wino}}(n/2)$. Cette récurrence se résout en $A_{\texttt{wacc}}(n) = 2/3 n^{\omega} + n^{\log_2(5)} - 5/3 n^2$.

1.2.1.3 Pistes pour améliorer les complexités des produits avec ou sans accumulation.

En observant attentivement ces ordonnancements, nous pouvons faire plusieurs remarques qui nous guiderons par la suite. C'est l'objet des paragraphes suivants.

Modifier les algorithmes.

En tout premier lieu, avec les algorithmes présentés tels quels (algorithme 0.1 et suivants), nous ne pouvons pas espérer réduire le nombre de blocs temporaires nécessaires ; par contre, nous pouvons modifier les algorithmes eux-mêmes. Nous pouvons remarquer dans l'algorithme 0.4 que le moment où C est accumulée au produit AB est en fait postérieur aux deux premières phases (pré-additions et multiplications). Nous avons cherché « à la main » (en fait aidé par Maple) des algorithmes qui feraient aussi des pré-additions sur C. C'est ainsi que, par exemple, est apparu l'algorithme 1.2. Par contre, cet algorithme utilise $21 > 19$ additions. Ensuite, le logiciel galet a permis de montrer que seulement

Algorithme 1.2 : Un nouvel algorithme de Winograd pour $C \leftarrow AB + C$.

$$C_{22} \leftarrow C_{22} - C_{12} \qquad C_{12} \leftarrow C_{12} - C_{21}$$

$$\approx$$

S_i et T_i comme dans l'algorithme 0.2.

$$\approx$$

$$\begin{array}{ll} P_1 \leftarrow A_{11} \times B_{11} & P_3 \leftarrow S_4 \times B_{22} + P_5 \\ P_2 \leftarrow A_{12} \times B_{21} + C_{11} & P_6 \leftarrow S_2 \times T_2 + C_{12} \\ P_5 \leftarrow S_1 \times T_1 + C_{12} & \end{array}$$

$$\approx$$

$$\begin{array}{ll} C_{11} \leftarrow P_1 + P_2 & C_{21} \leftarrow -A_{22} \times T_4 + U_3 \\ U_2 \leftarrow P_1 + P_6 & C_{22} \leftarrow P_5 + C_{22} \\ U_3 \leftarrow S_3 \times T_3 + U_2 & C_{22} \leftarrow U_3 + C_{22} \\ C_{12} \leftarrow U_2 + P_3 & \end{array}$$

deux blocs temporaires étaient nécessaires pour cet algorithme. L'ordonnancement accw proposé (tableau 1.3) est spécial dans le sens où l'on a en outre autorisé le *recalcul* de certaines variables (ici T_2, au prix de deux additions supplémentaires). Nous proposons donc dans le tableau 1.3 un nouvel or-

donnancement, nommé accw, pour cette opération $C \leftarrow \alpha A \times B + \beta C$ qui n'utilise que *deux* blocs temporaires et nous étudions ensuite ses complexités.

Tab. 1.3 — Ordonnancement accw pour $C \leftarrow \alpha A \times B + \beta C$ avec deux temporaires

n°	opération	loc.	n°	opération (suite)	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	14	$P_2 = \texttt{accw}\left(\alpha A_{12}B_{21} + \beta C_{11}\right)$	C_{11}
2	$Z_3 = C_{12} - C_{21}$	C_{12}	15	$\mathbf{U_1} = P_1 + P_2$	C_{11}
3	$S_1 = A_{21} + A_{22}$	X	16	$\mathbf{U_5} = U_2 + P_3$	C_{12}
4	$T_1 = B_{12} - B_{11}$	Y	17	$S_3 = A_{11} - A_{21}$	X
5	$P_5 = \texttt{accw}\left(\alpha S_1 T_1 + \beta Z_3\right)$	C_{12}	18	$T_3 = B_{22} - B_{12}$	Y
6	$S_2 = S_1 - A_{11}$	X	19	$U_3 = P_7 + U_2$	C_{21}
7	$T_2 = B_{22} - T_1$	Y		$= \texttt{accw}\left(\alpha S_3 T_3 + U_2\right)$	
8	$P_6 = \texttt{accw}\left(\alpha S_2 T_2 + \beta C_{21}\right)$	C_{21}	20	$\mathbf{U_7} = U_3 + W_1$	C_{22}
9	$S_4 = A_{12} - S_2$	X	21	$T'_1 = B_{12} - B_{11}$	Y
10	$W_1 = P_5 + \beta Z_1$	C_{22}	22	$T'_2 = B_{22} - T'_1$	Y
11	$P_3 = \texttt{accw}\left(\alpha S_4 B_{22} + P_5\right)$	C_{12}	23	$T_4 = T'_2 - B_{21}$	Y
12	$P_1 = \texttt{wino}\left(\alpha A_{11} B_{11}\right)$	X	24	$\mathbf{U_6} = U_3 - P_4$	C_{21}
13	$U_2 = P_6 + P_1$	C_{21}		$= \texttt{accw}\left(-\alpha A_{22} T_4 + U_3\right)$	

Complexité de l'ordonnancement accw.

- Les deux blocs temporaires X, Y ont des dimensions $X_s = m/2 \times \max\left\{k/2, n/2\right\}$ et $Y_s = k/2 \times n/2$. Nous pouvons donc écrire $E_{\texttt{accw}}(m,k,n) = E_{\texttt{accw}}\left(m/2, k/2, n/2\right) + X_s + Y_s$. D'après le lemme 1.1, nous obtenons $E_{\texttt{accw}}(m,k,n) < 1/3\,(m \max(k,n) + kn)$. Avec $m = n = k$, cela donne $E_{\texttt{accw}}(n,n,n) < 2/3 n^2 < E_{\texttt{wacc}}$.
- La complexité arithmétique satisfait la relation $W_{\texttt{accw}}(n) = 6W_{\texttt{accw}}\left(n/2\right) + W_{\texttt{wino}}\left(n/2\right) + 17\left(n/2\right)^2$. Donc $W_{\texttt{accw}}(n) = 6n^{\omega} + 2n^{\log_2(6)} - 6n^2$.
- Son nombre d'allocations est $A_{\texttt{accw}}(n) = 2\left(n/2\right)^2 + 6A_{\texttt{accw}}\left(n/2\right) + A_{\texttt{wino}}\left(n/2\right)$. Cette récurrence se résout en $A_{\texttt{accw}}(n) = 2/3\left(n^{\omega} - n^2\right)$.

Nous remarquons que la complexité $W_{\texttt{accw}}$ contient un terme en $n^{\log_2(6)} \approx n^{2,58}$, ce qui, il faut bien le dire, est mauvais. Par contre, $A_{\texttt{accw}}$ est meilleure (pas de terme en $n^{\log_2(5)} \approx n^{2,32}$). Nous allons maintenant étudier des solutions pour limiter la quantité de mémoire utilisée tout en réduisant cette complexité arithmétique.

Optimiser le nombre d'appels récursifs ?

Nous constatons que l'utilisation de deux blocs temporaires seulement nous permet de faire passer la quantité de mémoire supplémentaire de n^2 à $2/3 n^2$. Cependant, dans la complexité arithmétique, un terme en $n^{\log_2(6)} \approx 2{,}58$ est apparu. Afin de rester le plus proche des complexités classiques, nous fixons nos recherches à des complexités arithmétiques dans la classe $6n^{\omega} + \mathcal{O}^{\sim}\left(n^2\right)$; donc cette complexité $W_{\texttt{accw}}$ n'est pas acceptable.

Pour mieux comprendre la relation de récurrence (1.3) que satisfait l'ordonnancement wacc, étudions la relation suivante, pour un ordonnancement alg dont u est un paramètre entier entre 0 et 7, représentant le nombre d'appels récursifs, et v décrit le nombre d'additions. Si l'ordonnancement alg est un produit avec accumulation, alors les paramètres u et v sont liés par le fait que l'ordonnancement alg contiennent $k = u + v$ additions au total (en comptant les additions dans les produits avec accumulation).

$$\begin{cases} W_{\text{alg}}(n) = uW_{\text{alg}}\left(n/2\right) + (7-u)W_{\texttt{wino}}\left(n/2\right) + v\left(n/2\right)^2 \\ W_{\text{alg}}(1) = 2 \end{cases} . \tag{1.4}$$

Pour $u \in [\![0,6]\!]$, cette relation de récurrence prend la forme $6n^{\omega} + f(k,n)\,n^2 + \mathcal{O}(n^2)$. Pour $u = 7$, le terme dominant est $7{,}3 > 6$ ce qui est exclu. Dans la figure 1.4 suivante, nous représentons ce terme $f(k)$ afin de mieux cerner le rôle de du nombre d'appels récursifs u par rapport au nombre total d'additions.

u	0	1	2	3	4
$f(k)$	$1/4\,(k-35)$	$1/3\,(k-31)$	$1/2\,(k-27)$	$k-23$	$\frac{\log_2(n)}{4}\,(k-19)-4$

u	5	6
$f(k)$	$n^{\log_2(5)-2}\,(k-19)+15-k$	$1/2\left(n^{\log_2(3)-1}\,(k-19)+11-k\right)$

Fig.. 1.4 — Importance du nombre d'appels récursifs dans la complexité de wacc.

Nous constatons que le cas $u = 5$ (pour l'algorithme wacc) n'est intéressant que si $k \leqslant 19$, ce qui est le cas. Nous remarquons aussi que nous avons intérêt à produire des algorithmes avec u et k petits... Ainsi avisés, nous transformons l'ordonnancement accw en l'ordonnancement accw2 , comme indiqué [7] dans le tableau 1.4. Nous avons donc cherché à remplacer certains appels récursifs par des appels à l'ordonnancement wino. En effet, d'après la figure 1.4 pour $k = 23$, parmi les possibilités $u \in [\![3,6]\!]$, le plus petit $f(k)$ correspond à $u = 3$. Ainsi, nous atteignons une complexité $\mathrm{W}_{\texttt{accw2}} = 6n^{\omega} - 4n^{\log_2(3)}$, avec toujours $\mathrm{E}_{\texttt{accw2}} \leqslant 2/3\mathrm{E}_{\texttt{wacc}}$. Par ailleurs, quels que soient u et k, le nombre d'allocations reste constant et $\mathrm{A}_{\texttt{accw2}} = \mathrm{A}_{\texttt{accw}} = 2/3\left(n^{\omega} - n^2\right)$.

Tab. 1.4 — Optimisation accw2 de l'ordonnancement accw.

n°	opération	loc.
11	$\mathrm{W}_3 = \alpha\,\texttt{wino}\left(\mathrm{S}_4\mathrm{B}_{22}\right)$	Y
11bis	$\mathrm{P}_3 = \mathrm{W}_3 + \mathrm{P}_5$	C_{12}
	$\sim$	
14	$\mathrm{W}_4 = \alpha\,\texttt{wino}\left(\mathrm{A}_{12}\mathrm{B}_{21}\right)$	Y
14bis	$\mathrm{P}_2 = \mathrm{W}_4 + \beta\mathrm{C}_{11}$	C_{11}
	$\sim$	
24	$\mathrm{P}_4 = \alpha\,\texttt{wino}\left(\mathrm{A}_{22}\mathrm{T}_4\right)$	X
24bis	$\mathrm{U}_6 = \mathrm{U}_3 - \mathrm{P}_4$	C_{21}

Nous allons explorer par la suite des possibilités supplémentaires d'améliorations de ces ordonnancements.

Comment améliorer la complexité spatiale de ces algorithmes ?

Il est raisonnable de penser qu'en relâchant des hypothèses sur les règles du *jeu de galet* on pourra exhiber de meilleurs algorithmes que ceux présentés jusqu'alors. Par exemple, nous pouvons laisser un algorithme écraser ses entrées (en autorisant les galets initiaux à être déplacés ou écrasés par exemple). Nous remarquons alors que, dans l'ordonnancement proposé pour accw, à l'étape 5 par exemple, Z_3 peut être écrasé, ou dans l'étape 19, les opérandes S_3 et T_3 peuvent tous deux être écrasés. Cette idée a été proposée par [Kre76]. Il va alors s'avérer capital de trouver des algorithmes efficaces qui écrasent (certaines de) leurs entrées.

Toutes ces améliorations vont être testées, appliquées et évaluées dans la section 1.2.2 et suivantes. Le cas particulier des ordonnancements pour des entrées carrés est traité d'abord.

7 — Dans ces tableaux, la notation 'Xbis' signifie que l'opération numérotée 'Xbis' suit l'opération numérotée 'X'.

1.2.2 Ordonnancements avec écrasement des opérandes : cas carré.

Nous allons commencer par étudier le cas de la multiplication de matrices carrées. Nous développons des algorithmes qui écrasent leurs entrées à droite, à gauche ou des deux côtés, pour le produit standard (sous-section 1.2.2.1) et pour le produit avec accumulation (sous-section 1.2.2.2). Se restreindre au cas carré correspond à relâcher une contrainte sur les tailles des galets — cela revient à autoriser par exemple d'utiliser un bloc mémoire alloué pour une sous-matrice de A pour stocker un bloc mémoire de C ou réciproquement.

1.2.2.1 Produit standard.

Ordonnancement en place écrasant ses entrées.

Plus nous relâchons de contraintes, plus nous aurons de facilités à créer un ordonnancement, telle est notre conviction de départ. Dans le tableau 1.5 nous commençons donc par créer un nouvel ordonnancement (IP) pour le produit $C \leftarrow A \times B$ totalement en place. Une recherche exhaustive avec galet montre que l'on ne peut pas écraser moins de quatre sous-blocs.

Tab. 1.5 — Ordonnancement IP en place pour l'opération $C \leftarrow A \times B$.

n°	opération	loc.	n°	opération (suite)	loc.
1	$S_3 = A_{11} - A_{21}$	C_{11}	12	$S_4 = A_{12} - S_2$	A_{22}
2	$S_1 = A_{21} + A_{22}$	A_{21}	13	$P_6 = \mathtt{IP}\left(S_2T_2\right)$	C_{22}
3	$T_1 = B_{12} - B_{11}$	C_{22}	14	$U_2 = P_1 + P_6$	C_{22}
4	$T_3 = B_{22} - B_{12}$	B_{12}	15	$P_2 = \mathtt{IP}\left(A_{12}B_{21}\right)$	C_{12}
5	$P_7 = \mathtt{IP}\left(S_3T_3\right)$	C_{21}	16	$\mathbf{U_1} = P_1 + P_2$	C_{11}
6	$S_2 = S_1 - A_{11}$	C_{12}	17	$U_4 = U_2 + P_5$	C_{12}
7	$P_1 = \mathtt{IP}\left(A_{11}B_{11}\right)$	C_{11}	18	$U_3 = U_2 + P_7$	C_{22}
8	$T_2 = B_{22} - T_1$	B_{11}	19	$\mathbf{U_6} = U_3 - P_4$	C_{21}
9	$P_5 = \mathtt{IP}\left(S_1T_1\right)$	A_{11}	20	$\mathbf{U_7} = U_3 + P_5$	C_{22}
10	$T_4 = T_2 - B_{21}$	C_{22}	21	$P_3 = \mathtt{IP}\left(S_4B_{22}\right)$	A_{12}
11	$P_4 = \mathtt{IP}\left(A_{22}T_4\right)$	A_{21}	22	$\mathbf{U_5} = U_4 + P_3$	C_{12}

Nous remarquons que cet ordonnancement utilise seulement deux blocs de B (et tout A) mais écrase tout B (et tout A). En effet, le calcul de P_2 écrase des parties de B_{21} aussi. Cependant, au prix de copies et restaurations, nous parvenons à n'écraser que deux blocs de A et deux blocs de B. Nous le montrons dans le tableau 1.6.

Tab. 1.6 — IPbis : réductions des parties écrasées dans l'algorithme IP.

n°	opération	loc.	n°	opération (suite)	loc.
10bis	$C_{12} = A_{22}$	C_{12}	20bis	$A_{21} = A_{12}$	A_{21}
11	$P_4 = \mathtt{IP}\left(A_{22}T_4\right)$	A_{21}	20ter	$B_{12} = B_{21}$	B_{12}
11bis	$A_{22} = C_{12}$	A_{22}	21	$P_2 = \mathtt{IP}\left(A_{12}B_{21}\right)$	B_{11}
	$\sim$		21bis	$B_{21} = B_{12}$	B_{21}
15bis	$B_{12} = B_{22}$	B_{12}	21ter	$A_{12} = A_{21}$	A_{12}
16	$P_3 = \mathtt{IP}\left(S_4B_{22}\right)$	B_{11}			
16bis	$B_{22} = B_{12}$	B_{22}			

Par la suite, IP (pour *in-place*) désignera l'un ou l'autre de ces ordonnancements. Les complexités sont ici aisées à calculer.

Complexités de l'ordonnancement IP.

- Dans la version proposée dans le tableau 1.5, la complexité arithmétique W_{IP} satisfait à la même relation que W_{wino} donc lui est égale : $W_{IP} = 6n^{\omega} - 5n^2$. Dans le tableau 1.6, il faut bien sûr compter en plus le coût quadratique des copies et restaurations.
- Cet algorithme est en place donc $A_{IP} = E_{IP} = 0$.

Ordonnancements écrasant seulement une entrée.

Dans les tableaux 1.7 et 1.8 (page 28) nous proposons de n'écraser que l'opérande gauche ou le droit. Ces deux ordonnancements (OvL et OvR) utilisent de la mémoire supplémentaire — seulement un temporaire — mais une recherche exhaustive a montré qu'il n'existe aucun ordonnancement n'écrasant qu'un côté qui soit en place. Par ailleurs, une recherche exhaustive montre aussi qu'il n'existe pas d'ordonnancement écrasant seulement un bloc de l'opérande A.

Tab. 1.7 — Ordonnancement OvL pour l'opération $C \leftarrow A \times B$ en utilisant deux blocs de A et un temporaire

n°	opération	loc.	n°	opération (suite)	loc.
1	$S_3 = A_{11} - A_{21}$	C_{22}	12	$P_6 = \texttt{OvL}\left(S_2T_2\right)$	C_{21}
2	$S_1 = A_{21} + A_{22}$	A_{21}	13	$T_4 = T_2 - B_{21}$	A_{11}
3	$S_2 = S_1 - A_{11}$	C_{12}	14	$U_2 = P_1 + P_6$	C_{21}
4	$T_1 = B_{12} - B_{11}$	C_{21}	15	$U_4 = U_2 + P_5$	C_{12}
5	$P_1 = \texttt{OvL}\left(A_{11}B_{11}\right)$	C_{11}	16	$U_3 = U_2 + P_7$	C_{21}
6	$T_3 = B_{22} - B_{12}$	A_{11}	17	$\mathbf{U_7} = U_3 + P_5$	C_{22}
7	$P_7 = \texttt{IP}\left(S_3T_3\right)$	X	18	$\mathbf{U_5} = U_4 + P_3$	C_{12}
8	$T_2 = B_{22} - T_1$	A_{11}	19	$P_2 = \texttt{OvL}\left(A_{12}B_{21}\right)$	X
9	$P_5 = \texttt{IP}\left(S_1T_1\right)$	C_{22}	20	$\mathbf{U_1} = P_1 + P_2$	C_{11}
10	$S_4 = A_{12} - S_2$	C_{21}	21	$P_4 = \texttt{IP}\left(A_{22}T_4\right)$	A_{21}
11	$P_3 = \texttt{OvL}\left(S_4B_{22}\right)$	A_{21}	22	$\mathbf{U_6} = U_3 - P_4$	C_{21}

Tab. 1.8 — Ordonnancement OvR pour $C \leftarrow A \times B$ (pendant symétrique de OvL)

n°	opération	loc.	n°	opération (suite)	loc.
1	$S_3 = A_{11} - A_{21}$	C_{22}	12	$P_4 = \texttt{OvR}\left(A_{22}T_4\right)$	B_{12}
2	$S_1 = A_{21} + A_{22}$	C_{21}	13	$S_4 = A_{12} - S_2$	B_{11}
3	$T_1 = B_{12} - B_{11}$	C_{12}	14	$U_2 = P_1 + P_6$	C_{21}
4	$P_1 = \texttt{OvR}\left(A_{11}B_{11}\right)$	C_{11}	15	$U_4 = U_2 + P_5$	C_{12}
5	$S_2 = S_1 - A_{11}$	B_{11}	16	$U_3 = U_2 + P_7$	C_{21}
6	$T_3 = B_{22} - B_{12}$	B_{12}	17	$\mathbf{U_7} = U_3 + P_5$	C_{22}
7	$P_7 = \texttt{IP}\left(S_3T_3\right)$	X	18	$\mathbf{U_6} = U_3 - P_4$	C_{21}
8	$T_2 = B_{22} - T_1$	B_{12}	19	$P_3 = \texttt{IP}\left(S_4B_{22}\right)$	B_{12}
9	$P_5 = \texttt{IP}\left(S_1T_1\right)$	C_{22}	20	$\mathbf{U_5} = U_4 + P_3$	C_{12}
10	$T_4 = T_2 - B_{21}$	C_{12}	21	$P_2 = \texttt{OvR}\left(A_{12}B_{21}\right)$	B_{12}
11	$P_6 = \texttt{OvR}\left(S_2T_2\right)$	C_{21}	22	$\mathbf{U_1} = P_1 + P_2$	C_{11}

En autorisant des copies et en combinant les deux algorithmes OvL et OvR, nous remarquons que nous pouvons écraser seulement deux blocs de A dans OvL, en modifiant cet ordonnancement comme

présenté dans le tableau 1.9. La même technique s'applique pour OvR et nous pouvons écraser seulement deux blocs de B (voir le tableau 1.10).

Tab. 1.9 — OvLbis : modifications de OvL pour n'écraser que deux blocs de A

n°	opération	loc.
18bis	$A_{21} = A_{12}$	A_{21}
	$\sim$	
19bis	$A_{12} = A_{21}$	A_{12}
	$\sim$	
21	$P_4 = \texttt{OvR}\left(A_{22}T_4\right)$	A_{21}

Tab. 1.10 — OvLbis : modifications de OvR pour n'écraser que deux blocs de B

n°	opération	loc.
19	$P_3 = \texttt{OvL}\left(S_4B_{22}\right)$	B_{12}
	$\sim$	
20bis	$B_{11} = B_{21}$	B_{11}
	$\sim$	
21bis	$B_{21} = B_{11}$	B_{21}

Complexités des ordonnancements OvL et OvR.

Considérons par exemple le tableau 1.8 pour l'ordonnancement OvR. Comme l'ordonnancement OvL satisfait aux mêmes relations, il vérifie donc les mêmes expressions pour C, W et A.

- La taille du bloc temporaire X est $\left(n/2\right)^2$, donc la mémoire supplémentaire requise par l'ordonnancement OvR vérifie $E_{\texttt{OvR}}(n,n,n) < 1/3n^2$.
- Sa complexité arithmétique est donnée par $W_{\texttt{OvR}}(n) = 15\left(n/2\right)^2 + 4W_{\texttt{OvR}}\left(n/2\right) + 3W_{\texttt{IP}}\left(n/2\right)$. On obtient donc $W_{\texttt{OvR}}(n) = W_{\texttt{wino}} = 6n^{\omega} - 5n^2$.
- Son nombre d'allocations satisfait $A_{\texttt{OvR}}(n) = \left(n/2\right)^2 + 4A_{\texttt{OvR}}\left(n/2\right) + 3A_{\texttt{IP}}\left(n/2\right)$. On a donc $A_{\texttt{OvR}}(n) = 1/4n^2\log_2(n)$.

Ici, faire le maximum d'appels à IP permet de réduire le nombre d'allocations. Par exemple, si le dernier appel (ligne 21, tableau 1.8) avait été fait avec wino dans le bloc temporaire X, la complexité arithmétique n'aurait pas diminué et A aurait significativement augmenté.

1.2.2.2 Produit avec accumulation.

Ordonnancement écrasant ses deux entrées.

Considérons donc maintenant l'opération $C \leftarrow \alpha A \times B + \beta C$ où les opérandes A et B peuvent être écrasés. Dans le tableau 1.11, nous proposons un ordonnancement AcLR nécessitant seulement deux blocs temporaires au lieu des trois dans l'ordonnancement wacc. Nous revisitons en effet notre ordonnancement accw en utilisant notamment l'ordonnancement IP. Dans notre jeu de galet, nous ne pouvons par exemple utiliser cet algorithme pour l'opération mul que lorsque ses deux arguments ont un galet qui peut être écrasé (c'est le cas en particulier lorsque ces galets n'ont pas d'autres fils que celui impliqué dans la multiplication).

Complexités de l'ordonnancement AcLR.

- La mémoire supplémentaire requise dans le tableau 1.11 pour X et Y est $2\left(n/2\right)^2$. Donc, avec le lemme 1.1, on obtient $E_{\texttt{AcLR}}(n,n,n) < 2/3n^2$.
- La complexité arithmétique du tableau 1.11 vérifie $W_{\texttt{AcLR}}(n) = 2W_{\texttt{AcLR}}\left(n/2\right) + 5W_{\texttt{IP}}\left(n/2\right) + 19\left(n/2\right)^2$. Ainsi $W_{\texttt{AcLR}}(n) = 6n^{\omega} - 3n^2 - n < W_{\texttt{accw2}}(n)$.
- Son nombre d'allocations est donné par la récurrence $A_{\texttt{AcLR}}(n) = 2\left(n/2\right)^2 + 2A_{\texttt{AcLR}}\left(n/2\right) + 5A_{\texttt{IP}}\left(n/2\right)$. Cela se résout en $A_{\texttt{AcLR}}(n) = 2n^2 - 2n^{\log_3(2)}$.

Remarque. Nous notons que plusieurs appels pourraient être remplacés par des appels récursifs, notamment les calculs de W_1 et W_2 (et aussi, assez facilement, ceux de P_1, P_3) mais $W_{\texttt{AcLR}}$ croît.

Tab. 1.11 — Ordonnancement AcLR pour $C \leftarrow \alpha A \times B + \beta C$ écrasant A et B avec deux temporaires et deux appels récursifs

nº	opération	loc.	nº	opération (suite)	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	14	$S_4 = A_{12} - S_2$	A_{22}
2	$S_1 = A_{21} + A_{22}$	X	15	$P_6 = \alpha \mathtt{IP}\left(S_2 T_2\right)$	X
3	$T_1 = B_{12} - B_{11}$	Y	16	$W_2 = \mathtt{IP}\left(\alpha A_{12} B_{21}\right)$	Y
4	$Z_2 = C_{21} - Z_1$	C_{21}	17	$P_2 = W_2 + \beta C_{11}$	C_{11}
5	$T_3 = B_{22} - B_{12}$	B_{12}	18	$P_1 = \alpha \mathtt{IP}\left(A_{11} B_{11}\right)$	Y
6	$S_3 = A_{11} - A_{21}$	A_{21}	19	$\mathbf{U_1} = P_1 + P_2$	C_{11}
5	$P_7 = \mathtt{AcLR}\left(\alpha S_3 T_3 + \beta Z_1\right)$	C_{22}	20	$U_2 = P_1 + P_6$	X
8	$S_2 = S_1 - A_{11}$	A_{21}	21	$U_3 = U_2 + P_7$	C_{22}
9	$T_2 = B_{22} - T_1$	B_{12}	22	$U_4 = U_2 + P_5$	X
10	$P_5 = \mathtt{AcLR}\left(\alpha S_1 T_1 + \beta C_{12}\right)$	C_{12}	23	$\mathbf{U_6} = U_3 - P_4$	C_{21}
11	$T_4 = T_2 - B_{21}$	X	24	$\mathbf{U_7} = U_3 + P_5$	C_{22}
12	$W_1 = \mathtt{IP}\left(\alpha A_{22} T_4\right)$	Y	25	$P_3 = \alpha \mathtt{IP}\left(S_4 B_{22}\right)$	C_{12}
13	$P_4 = W_1 - \beta Z_2$	C_{21}	26	$\mathbf{U_5} = U_4 + P_3$	C_{12}

Ordonnancement écrasant son opérande droit.

Nous créons aussi le tableau 1.12 qui représente un ordonnancement similaire à AcLR mais qui n'écrase que l'opérande de droite. Il utilise lui aussi seulement deux temporaires et fait appel à l'ordonnancement OvR.

Tab. 1.12 — Ordonnancement AccR pour $C \leftarrow \alpha A \times B + \beta C$ écrasant B avec deux temporaires et trois appels récursifs.

nº	opération	loc.	nº	opération (suite)	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	14	$S_2 = S_1 - A_{11}$	Y
2	$T_1 = B_{12} - B_{11}$	X	15	$P_6 = \alpha \mathtt{OvR}\left(S_2 T_2\right)$	B_{21}
3	$Z_2 = C_{21} - Z_1$	C_{21}	16	$S_4 = A_{12} - S_2$	Y
4	$T_3 = B_{22} - B_{12}$	B_{12}	17	$P_1 = \alpha \mathtt{OvR}\left(A_{11} B_{11}\right)$	X
5	$S_3 = A_{11} - A_{21}$	Y	18	$U_2 = P_1 + P_6$	B_{21}
6	$P_7 = \mathtt{AcLR}\left(\alpha S_3 T_3 + \beta Z_1\right)$	C_{22}	19	$U_3 = U_2 + P_7$	C_{22}
7	$S_1 = A_{21} + A_{22}$	Y	20	$U_4 = U_2 + P_5$	B_{21}
8	$T_2 = B_{22} - T_1$	B_{12}	21	$\mathbf{U_6} = U_3 - P_4$	C_{21}
9	$P_5 = \mathtt{AccR}\left(\alpha S_1 T_1 + \beta C_{12}\right)$	C_{12}	22	$\mathbf{U_1} = P_1 + P_2$	C_{11}
10	$T_4 = T_2 - B_{21}$	X	23	$\mathbf{U_7} = U_3 + P_5$	C_{22}
11	$P_4 = \mathtt{AccR}\left(\alpha A_{22} T_4 - \beta Z_2\right)$	C_{21}	24	$P_3 = \alpha \mathtt{IP}\left(S_4 B_{22}\right)$	C_{12}
12	$W_1 = \mathtt{OvR}\left(\alpha A_{12} B_{21}\right)$	X	25	$\mathbf{U_5} = U_4 + P_3$	C_{12}
13	$P_2 = W_1 + \beta C_{11}$	C_{11}			

Complexités de l'ordonnancement AccR.

✦ La mémoire supplémentaire $E_{\mathtt{AccR}}(n, n, n)$ est $\left(n/2\right)^2 + \left(n/2\right)^2 + \max\left(E_{\mathtt{AccR}}, E_{\mathtt{OvR}}, E_{\mathtt{AcLR}}\right)\left(n/2, n/2, n/2\right)$. Nous avons $E_{\mathtt{AccR}} > E_{\mathtt{OvR}}$ car l'ordonnancement OvR n'utilise qu'un bloc temporaire. Au final, d'après le lemme, et puisque AcLR utilise le même nombre de blocs temporaires, nous obtenons $E_{\mathtt{AccR}}(n, n, n) < 2/3 n^2$.

- La complexité algorithmique du tableau 1.12 vérifie $W_{\mathtt{AccR}}(n) = 2W_{\mathtt{AccR}}\left(n/2\right) + W_{\mathtt{AcLR}}\left(n/2\right) + 3W_{\mathtt{OvR}}\left(n/2\right) + W_{\mathtt{IP}}\left(n/2\right) + 18\left(n/2\right)^2$. Donc $W_{\mathtt{AccR}} = 6n^{\omega} - 5/2n^2 - 1/2n\log_2(n) - 3/2n < W_{\mathtt{accw2}}$.
- Par ailleurs, le nombre d'allocations supplémentaires de AccR satisfait la relation $A_{\mathtt{AccR}}(n) = 2\left(n/2\right)^2 + 2A_{\mathtt{AccR}}\left(n/2\right) + A_{\mathtt{AcLR}}\left(n/2\right) + 3A_{\mathtt{OvR}}\left(n/2\right) + A_{\mathtt{IP}}\left(n/2\right)$, ce qui donne $A_{\mathtt{AccR}}(n) = 3/8n^2\log_2(n) + 5/4n^2 - 2n^{\log_2(3)} + 3/4n$.

Remarque. Ici aussi W_1 aurait pu être calculé avec un appel récursif, mais les complexités n'auraient qu'augmenté.

Ordonnancement écrasant son opérande gauche.

Nous présentons finalement dans le tableau 1.13 un tout nouvel ordonnancement qui est l'analogue à gauche du tableau 1.12.

Tab. 1.13 — Ordonnancement AccL pour $C \leftarrow \alpha A \times B + \beta C$ écrasant A avec deux temporaires et 4 appels récursifs.

n°	opération	loc.	n°	opération (suite)	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	14	$\mathbf{U_1} = P_1 + P_2$	C_{11}
2	$Z_2 = C_{21} - Z_1$	C_{21}	15	$P_6 = \alpha\mathtt{OvL}\left(S_2T_2\right)$	A_{12}
3	$S_3 = A_{11} - A_{21}$	X	16	$T_4 = T_2 - B_{21}$	Y
4	$S_1 = A_{21} + A_{22}$	A_{21}	17	$W_2 = \mathtt{IP}\left(\alpha A_{22}T_4\right)$	A_{21}
5	$T_3 = B_{22} - B_{12}$	Y	18	$P_4 = W_2 - \beta Z_2$	C_{21}
6	$P_7 = \mathtt{AcLR}\left(\alpha S_3T_3 + \beta Z_1\right)$	C_{22}	19	$U_2 = P_6 + P_1$	X
7	$T_1 = B_{12} - B_{11}$	X	20	$U_3 = U_2 + P_7$	C_{22}
8	$T_2 = B_{22} - T_1$	Y	21	$\mathbf{U_6} = U_3 - P_4$	C_{21}
9	$P_5 = \mathtt{AccR}\left(\alpha S_1T_1 + \beta C_{12}\right)$	C_{12}	22	$\mathbf{U_7} = U_3 + P_5$	C_{22}
10	$S_2 = S_1 - A_{11}$	A_{21}	23	$U_4 = U_2 + P_5$	C_{12}
11	$P_1 = \alpha\mathtt{OvL}\left(A_{11}B_{11}\right)$	X	24	$W_3 = \mathtt{OvL}\left(\alpha S_4B_{22}\right)$	Y
12	$S_4 = A_{12} - S_2$	A_{11}	25	$\mathbf{U_5} = W_3 + U_4$	C_{12}
13	$P_2 = \mathtt{AccL}\left(\alpha A_{12}B_{21} + \beta C_{11}\right)$	C_{11}			

Complexités de l'ordonnancement AccL.

- La mémoire supplémentaire $E_{\mathtt{AccL}}(n,n,n)$ satisfait à la relation de récurrence $\left(n/2\right)^2 + \left(n/2\right)^2 + \max\left\{E_{\mathtt{AccL}}, E_{\mathtt{AccR}}, E_{\mathtt{OvL}}, E_{\mathtt{AcLR}}\right\}\left(n/2, n/2, n/2\right)$. Nous avons $E_{\mathtt{AccL}} > E_{\mathtt{OvR}}$ pour les mêmes raisons que précédemment. Comme $E_{\mathtt{AccL}}$, $E_{\mathtt{AcLR}}$ et $E_{\mathtt{AccR}}$ utilisent le même nombre de blocs, nous obtenons $E_{\mathtt{AccL}}(n,n,n) < 2/3n^2$.
- La complexité algorithmique de l'ordonnancement AccL vérifie $W_{\mathtt{AccL}}(n) = W_{\mathtt{AccL}}\left(n/2\right) + W_{\mathtt{AccR}}\left(n/2\right) + W_{\mathtt{AcLR}}\left(n/2\right) + 3W_{\mathtt{OvL}}\left(n/2\right) + W_{\mathtt{IP}}\left(n/2\right) + 18\left(n/2\right)^2$. On trouve $W_{\mathtt{AccL}}(n) = 6n^{\omega} - 5/2n^2 - n^{\log_2(3)} - 1/2n\log_2(n) - 3/2n < W_{\mathtt{accw2}}$.
- Son nombre d'allocations supplémentaires est $A_{\mathtt{AccL}}(n) = 2\left(n/2\right)^2 + A_{\mathtt{AccL}}\left(n/2\right) + A_{\mathtt{AccR}}\left(n/2\right) + A_{\mathtt{AcLR}}\left(n/2\right) + 3A_{\mathtt{OvL}}\left(n/2\right) + A_{\mathtt{IP}}\left(n/2\right)$, ce qui donne $A_{\mathtt{AccL}}(n) = 3/8n^2\log_2(n) + 5/4n^2 - 2n^{\log_2(3)} + 3/4n$.

Remarques sur ces calculs de complexités.

Même si ces tableaux et ces calculs de complexités peut apparaître austères ou énumératifs, ils n'en sont pas moins essentiels pour parvenir à les comprendre et, en les combinant, retrouver, comme nous l'avons formulé plus haut, une complexité de la forme $6n^{\omega} + \mathcal{O}^{\sim}(n^2)$. Mieux encore, nous avons réussi à confiner les complexités en deçà de $6n^{\omega} - 2n^2$. Globalement, pour arriver à nos fins, à partir de l'ordonnancement wacc, nous avons dû :

— créer de nouveaux ordonnancements aux propriétés d'écrasement des opérandes diverses ;

— jouer finement sur la complexité de nouveaux ordonnancements ;
— jongler entre nombre d'appels récursifs et appel à des fonctions spécialisées.

Par contre, nos ordonnancements ne sont valables que dans le cas d'opérandes carrés.

1.2.3 Ordonnancements avec entrées constantes : cas carré.

En combinant les techniques des sections 1.2.1 et 1.2.2 précédentes, nous parvenons à améliorer significativement l'ordonnancement accw. Nous proposons donc, dans le tableau 1.14, un ordonnancement hybride qui calcule $C \leftarrow \alpha A \times B + \beta C$ avec des opérandes constants et une mémoire supplémentaire plus petite que celle de [HLJJ+96b] (tableau 1.2) et dont la complexité entre dans la classe ciblée $(6n^{\omega} + \mathcal{O}^{\sim}(n^2))$.

Tab. 1.14 — Ordonnancement Acc pour l'opération $C \leftarrow \alpha A \times B + \beta C$ avec 2 blocs temporaires

n°	opération	loc.	n°	opération (suite)	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	15	$W_2 = \texttt{wino}\left(\alpha A_{12} B_{21}\right)$	Y
2	$Z_3 = C_{12} - C_{21}$	C_{12}	16	$P_2 = W_2 + \beta C_{11}$	C_{11}
3	$S_1 = A_{21} + A_{22}$	X	17	$\mathbf{U_1} = P_1 + P_2$	C_{11}
4	$T_1 = B_{12} - B_{11}$	Y	18	$\mathbf{U_5} = U_2 + P_3$	C_{12}
5	$P_5 = \texttt{Acc}\left(\alpha S_1 T_1 + \beta Z_3\right)$	C_{12}	19	$S_3 = A_{11} - A_{21}$	X
6	$S_2 = S_1 - A_{11}$	X	20	$T_3 = B_{22} - B_{12}$	Y
7	$T_2 = B_{22} - T_1$	Y	21	$U_3 = \alpha\texttt{AcLR}\left(S_3 T_3 + U_2\right)$	C_{21}
8	$P_6 = \texttt{AccR}\left(\alpha S_2 T_2 + \beta C_{21}\right)$	C_{21}	22	$\mathbf{U_7} = U_3 + Z_3$	C_{22}
9	$S_4 = A_{12} - S_2$	X	23	$T'_1 = B_{12} - B_{11}$	Y
10	$Z_3 = P_5 + \beta Z_1$	C_{22}	24	$T'_2 = B_{22} - T'_1$	Y
11	$W_1 = \texttt{OvL}\left(\alpha S_4 B_{22}\right)$	Y	25	$T_4 = T'_2 - B_{21}$	Y
12	$P_3 = W_1 + P_5$	C_{12}	26	$P_4 = \alpha\texttt{OvR}\left(A_{22} T_4\right)$	Y
13	$P_1 = \texttt{wino}\left(\alpha A_{11} B_{11}\right)$	X	27	$\mathbf{U_6} = U_3 - P_4$	C_{21}
14	$U_2 = P_6 + P_1$	C_{21}			

Complexité de l'ordonnancement Acc.

- Encore une fois, les blocs X et Y ont des dimensions $(n/2)^2$ de telle sorte que : $E_{\texttt{Acc}} = 2(n/2) + \max\left\{E_{\texttt{Acc}}, E_{\texttt{AcLR}}, E_{\texttt{OvR}}, E_{\texttt{wino}}, E_{\texttt{AccR}}\right\}\left(n/2, n/2, n/2\right)$. Dans tous les cas, ces ordonnancements n'utilisent pas plus de deux temporaires, d'où $E_{\texttt{Acc}}(m, k, n) < 2/3 n^2$.
- La complexité algorithmique de l'ordonnancement Acc satisfait $W_{\texttt{Acc}}(n) = W_{\texttt{Acc}}\left(n/2\right) + 2W_{\texttt{wino}}\left(n/2\right) + W_{\texttt{AcLR}}\left(n/2\right) + W_{\texttt{AccR}}\left(n/2\right) + 2W_{\texttt{OvR}}\left(n/2\right) + 20\left(n/2\right)^2$. On obtient $W_{\texttt{Acc}}(n) = 6n^{\omega} - 11/6 n^2 - 1/2 n \log_2(n) - 3/2 n - 3/2$.
- Son nombre total d'allocations satisfait $A_{\texttt{Acc}}(n) = A_{\texttt{Acc}}\left(n/2\right) + 2A_{\texttt{wino}}\left(n/2\right) + A_{\texttt{AcLR}}\left(n/2\right) + A_{\texttt{AccR}}\left(n/2\right) + 2A_{\texttt{OvR}}\left(n/2\right) + 2\left(n/2\right)^2$, soit $A_{\texttt{Acc}}(n) = 2/9 n^{\omega} + 7/24 n^2 \log_2(n) + 11/12 n^2 - 2n^{\log_2(3)} + 3/4 n + 1/9$.

1.2.4 Cas général : ordonnancements hybrides.

Nous avons jusqu'à présent considéré des algorithmes pour le cas où les entrées étaient carrées ($m = k = n$). Nous étendons maintenant ces algorithmes au cas général.

Quelles contraintes pour des tailles générales ?

Tout d'abord, nous expliquons dans la proposition 1.1 suivante pourquoi nous avons dû nous restreindre à étudier le cas carré. L'idée est naturellement que si nous réutilisons de l'espace mémoire, alors cela va contraindre la taille des variables qui l'utilise.

Proposition 1.1. — Pour l'algorithme de Winograd (algorithme 0.2), il n'existe pas d'ordonnancement en place pour des entrées de tailles générales.

Démonstration. Cette démonstration consiste à jouer au jeu de galet sur l'arbre de la figure 1.1 avec huit galets non fixes sur les entrées et quatre supplémentaires pour recouvrir la sortie.

Tout d'abord, les tailles de A et B ne doivent pas être plus grandes que celles de C (i.e. nous imposons $k \leqslant \min(m, n)$). En effet, jouons un jeu de galet avec 4 galets supplémentaires, de la taille des $C_{i,j}$. Aucun galet sur un nœud initial ne peut être déplacé puisqu'au moins deux arrêtes partent de chacun. Si la taille de A_{ij} est plus grande que celle des galets libres, alors aucun galet ne peut être placé sur un S_i. On ne peut pas non plus en mettre un sur P_1 ou P_2 puisque leurs entrées seraient alors écrasées. La taille des A_{ij} est donc plus petite que celle des C_{ij}. Le même raisonnement s'applique pour B_{ij}.

Maintenant, si nous considérons un jeu qui a terminé, on peut prouver de la même manière que, parmi la taille de A et celle de B, l'une doit être plus petite que celle de C. En particulier, l'un ou l'autre doit être de la taille de C. □

Le tableau 1.5 (`IP`) montre en outre qu'il est possible de trouver un ordonnancement tel que décrit dans la preuve de la proposition 1.1, avec $k = n \leqslant m$. Il est aussi possible d'échanger les rôles de m et n. En outre, dans les tableaux 1.5 à 1.8, les tailles de A, B et C doivent être égales. Il n'est donc pas possible avec ces arguments d'utiliser nos tableaux pour des entrées de tailles générales. Une solution que nous proposons consiste à découper les matrices en entrée en sous-blocs carrés et d'utiliser notre ensemble d'algorithmes spécialisés pour le cas carré dessus. Le niveau le plus haut dans l'algorithme n'est donc autre qu'une multiplication naïve.

Puisque les algorithmes précédents utilisent moins de mémoire supplémentaire que la taille d'une de ces petites matrices carrées, nous pouvons les utiliser dès que l'une est libre. Nous proposons l'algorithme 1.3 (`IPOvMM`, *In-Place Overwrite Matrix Multiply*) pour le cas $n < \min(m, k)$:

Algorithme 1.3 : `IPOvMM` : multiplication de matrice en place avec écrasement

Entrées : A et B de taille $m \times k$ et $k \times n$
Entrées : $n < \min(m, k)$ et m, k, n des puissances de 2.
Sorties : $C = A \times B$

1 Soit $k_0 = k/n$ et $m_0 = m/n$
2 Découpe $A = \left[\begin{array}{c|c|c} A_{1,1} & \cdots & A_{1,k_0} \\ \hline \vdots & & \vdots \\ \hline A_{m_0,1} & \cdots & A_{m_0,k_0} \end{array}\right]$
3 Découpe $B = \left[\begin{array}{c} B_1 \\ \hline \vdots \\ \hline B_{k_0} \end{array}\right]$ et $C = \left[\begin{array}{c} C_1 \\ \hline \vdots \\ \hline C_{k_0} \end{array}\right]$ /* où $A_{i,j}$ et B_j ont pour dimension $n \times n$ */
4 $C_1 \leftarrow$ `wino` $(A_{1,1}B_1)$ /* avec la mémoire C_2. */
5 **pour** $i = 2 \ldots m_0$ **faire**
6 | $C_i \leftarrow$ `OvL` $(A_{i,1}B_1)$ /* avec la mémoire $A_{1,1}$. */
7 **fin**
8 **pour** $j = 2 \ldots k_0$ **faire**
9 | **pour** $i = 1 \ldots m_0$ **faire**
10 | | $C_i \leftarrow$ `wacc` $(A_{i,j}B_j + C_i)$ /* avec la mémoire $A_{1,1}$. */
11 | **fin**
12 **fin**

Il est facile de voir que l'algorithme 1.3 (`IPOvMM`) calcule bien le produit $C = A \times B$, et qu'il est en place, en écrasant seulement une partie de A. En effet, les tailles des sous-matrices $A_{i,j}$, B_i et C_j sont

toutes égales à n^2 et la mémoire nécessaire pour des multiplications de taille (n, n, n) est inférieure à n^2 avec les ordonnancements `OvL` ou `wacc` et `wino`.

Remarques. Nous pouvons faire diverses remarques sur cet algorithme.

- Tout d'abord, supposons que $m = k > n$. Dans ce cas, un petit calcul de complexité donne

$$\mathrm{W}_{\texttt{wino}}(m, m, n) = \frac{7}{3} n^{\omega} \left(\frac{m^2}{n^2} + \frac{11m}{7n} \right) - 1/3 \left(m^2 + mn \right) \tag{1.5}$$

$$\mathrm{W}_{\texttt{IPOvMM}}(m, m, n) = 6n^{\omega} \times \frac{m^2}{n^2} - \left(4m^2 + mn \right). \tag{1.6}$$

La complexité obtenue n'est donc pas extrêmement intéressante... Nous montrerons dans l'algorithme 1.4 ci-après une meilleure approche.

- Cet algorithme peut éventuellement avoir une meilleure complexité en commençant à l'étape 4 par la plus grande multiplication possible — c'est-à-dire le plus grand (m, k, n) tel que la somme des tailles de la mémoire supplémentaire de `wino` et de son résultat soit maximale dans la taille de C.
- Pour avoir les meilleures complexités nous utilisons `wino` et `wacc` ! En effet, même si les complexités spatiales sont moins bonnes, leurs complexités arithmétiques est meilleure.
- En remplaçant `OvL` par `wino`, il est possible de n'écraser que $\mathrm{A}_{1,1}$, et donc, à une mémoire de sauvegarde de n^2 près, cet algorithme ne modifie pas ses entrées.

Nous proposons maintenant, dans l'algorithme 1.4 (`IPMM`, *In-Place Matrix Multiply*), un algorithme pour $\mathrm{C} \leftarrow \mathrm{A} \times \mathrm{B}$ en $\mathcal{O}\left(n^{\omega}\right)$ opérations arithmétiques sans mémoire supplémentaire et sans écraser ses opérandes. En conséquence, cet algorithme permettra d'améliorer la complexité spatiale du produit avec accumulation $\mathrm{C} \leftarrow \alpha \mathrm{A} \times \mathrm{B} + \beta \mathrm{C}$.

L'idée principale est encore de découper la matrice C en quatre sous-matrices de dimension moitié et de les calculer en utilisant dans un premier niveau l'algorithme trivial. Pour chacune des trois sous matrices C_{11}, C_{12} et C_{21}, on fait $2k/n$ multiplications rapides sur des matrices de taille $n/2 \times n/2$. La mémoire temporaire pour ces calculs est prise dans C_{22}. Finalement, le bloc C_{22} est calculé récursivement.

Algorithme 1.4 : `IPMM` : multiplication matricielle rapide en place

Entrées : A et B, de dimensions resp. $n \times k$ et $k \times n$ avec k, n des puissances de 2 et $k \geqslant n$.
Sorties : $\mathrm{C} = \mathrm{A} \times \mathrm{B}$

Découpe $\mathrm{C} = \begin{bmatrix} \mathrm{C}_{11} & \mathrm{C}_{12} \\ \mathrm{C}_{21} & \mathrm{C}_{22} \end{bmatrix}$, $\mathrm{A} = \left[\begin{array}{c|c|c} \mathrm{A}_{1,1} & \dots & \mathrm{A}_{1,2k/n} \\ \hline \mathrm{A}_{2,1} & \dots & \mathrm{A}_{2,2k/n} \end{array}\right]$

Découpe $\mathrm{B} = \left[\begin{array}{c|c} \mathrm{B}_{1,1} & \mathrm{B}_{1,2} \\ \hline \vdots & \vdots \\ \hline \mathrm{B}_{2k/n,1} & \mathrm{B}_{2k/n,2} \end{array}\right]$ /* où chaque $\mathrm{A}_{i,j}, \mathrm{B}_{i,j}$ et $\mathrm{C}_{i,j}$ ont pour dimension $n/2 \times n/2$. */

début
- $\mathrm{C}_{11} = \texttt{wino}\left(\mathrm{A}_{1,1}\mathrm{B}_{1,1}\right)$ /* avec C_{22} comme espace temp. */
- $\mathrm{C}_{12} = \texttt{wino}\left(\mathrm{A}_{1,1}\mathrm{B}_{1,2}\right)$
- $\mathrm{C}_{21} = \texttt{wino}\left(\mathrm{A}_{2,1}\mathrm{B}_{1,1}\right)$

fin

pour $i = 2 \dots \frac{2k}{n}$ **faire**
- $\mathrm{C}_{11} = \texttt{wacc}\left(\mathrm{A}_{1,i}\mathrm{B}_{i,1} + \mathrm{C}_{11}\right)$ /* avec C_{22} comme espace temporaire : */
- $\mathrm{C}_{12} = \texttt{wacc}\left(\mathrm{A}_{1,i}\mathrm{B}_{i,2} + \mathrm{C}_{12}\right)$
- $\mathrm{C}_{21} = \texttt{wacc}\left(\mathrm{A}_{2,i}\mathrm{B}_{i,1} + \mathrm{C}_{21}\right)$

fin

$\mathrm{C}_{22} = \texttt{IPMM}\left(\mathrm{A}_{2,*} \times \mathrm{B}_{*,2}\right)$ /* récursivement. */

Théorème 1.1. — La complexité de l'algorithme 1.4 (IPMM) est, lorsque $k = n$:

$$\mathrm{W}_{\mathtt{IPMM}}(n) = 7{,}2n^{\omega} - 13n^2 + 6{,}8n.$$

Démonstration. Nous rappelons que le coût d'un appel à l'algorithme de Winograd est $\mathrm{W}_{\mathtt{wino}}(n) = 6n^{\omega} - 5n^2$ et $\mathrm{W}_{\mathtt{wacc}}(n) = 6n^{\omega} - 4n^2$ pour opération $\mathrm{C} \leftarrow \mathrm{A} \times \mathrm{B} + \mathrm{C}$. Le coût $\mathrm{W}_{\mathtt{IPMM}}(n, k)$ est donné par la relation de récurrence

$$\mathrm{W}_{\mathtt{IPMM}}(n, k) = 3\mathrm{W}_{\mathtt{wino}}\left(n/2\right) + 3\left(2k/n - 1\right)\mathrm{W}_{\mathtt{wacc}}\left(n/2\right) + \mathrm{W}_{\mathtt{IPMM}}\left(n/2, k\right),$$

le cas de base étant un produit scalaire $\mathrm{W}_{\mathtt{IPMM}}(1, k) = 2k - 1$. Ainsi, $\mathrm{W}_{\mathtt{IPMM}}(n, k) = 7{,}2kn^{\omega-1} - 12kn - n^2 + 34k/5$. □

Théorème 1.2. — Pour tout m, n et k, l'algorithme IPMM est en place.

Démonstration. Sans perte de généralité, on peut supposer que $m \geqslant n > 1$ (sinon, on transpose). Nous avons déjà calculé $\mathrm{E}_{\mathtt{wino}}$ et $\mathrm{E}_{\mathtt{wacc}}$.

Si nous découpons B en p_i bandes à l'étape de récurrence i, alors les tailles des sous-matrices sont pour A (resp. B) $m/2^i \times k/p_i$ (resp. $k/p_i \times n/2^i$). Or C_{22} a pour taille $m/2^i \times n/2^i$. Nous avons donc besoin de vérifier que :

$$\max\left(\mathrm{E}_{\mathtt{wino}}, \mathrm{E}_{\mathtt{wacc}}\right)\left(\frac{m}{2^i}, \frac{k}{p_i}, \frac{n}{2^i}\right) \leqslant \frac{m}{2^i}\frac{n}{2^i}. \tag{1.7}$$

Il est clair que $\mathrm{E}_{\mathtt{wino}} < \mathrm{E}_{\mathtt{wacc}}$, ce qui simplifie l'inégalité précédente. Soit $\mathrm{K} = k/p_i$, $\mathrm{M} = m/2^i$ et $\mathrm{N} = n/2^i$. Nous avons besoin de trouver, pour chaque i un entier $p_i > 1$ tel que l'équation (1.7) soit vraie. En d'autres mots, montrons qu'il existe $\mathrm{K} < k$ tel que pour tout couple (M, N), l'inégalité $\mathrm{E}_{\mathtt{wacc}}(\mathrm{M}, \mathrm{K}, \mathrm{N}) \leqslant \mathrm{MN}$ a lieu. Alors, le fait que $\mathrm{E}(\mathrm{M}, 2, \mathrm{N}) < 1/3(2\mathrm{M} + 2\mathrm{N} + \mathrm{MN}) \leqslant 1/3(4\mathrm{M} + \mathrm{MN}) \leqslant \mathrm{MN}$ permet de trouver au moins un tel K.

Comme les pré-requis de l'algorithme IPMM permettent d'avoir $k > \mathrm{N}$ et $\mathrm{M} = \mathrm{N}$, il reste simplement à prouver que $\mathrm{E}(\mathrm{M}, \mathrm{N}, \mathrm{N}) \leqslant \mathrm{MN}$. Puisque $\mathrm{E}(\mathrm{M}, \mathrm{N}, \mathrm{N}) < 1/3(2\mathrm{MN} + \mathrm{N}^2)$ et que $\mathrm{M} \geqslant \mathrm{N}$, l'algorithme IPMM est bien en place. □

Nous avons donc un algorithme en place, avec une complexité de $\mathcal{O}\left(n^{\omega}\right)$. La constante multiplicative a cependant un peu augmenté (de 6 à 7,2, soit 20%). Nous pouvons bien sûr étendre cette approche à des tailles plus générales, en découpant généralement la matrice selon des carrés de taille $\min(m, k, n)$. Nous y reviendrons dans la section 5.2.

Produit avec accumulation : réduction de la mémoire temporaire.

Dans le cas du produit avec accumulation, la matrice C ne peut pas directement être utilisée comme espace mémoire et nous ne pouvons pas nous abstraire d'une mémoire supplémentaire Nous utilisons encore l'idée de la multiplication classique par blocs au premier niveau puis les appels aux routines rapides.

Comme précédemment, C est divisée en quatre blocs puis le produit peut être fait avec 8 appels à l'algorithme de Winograd sur les petits blocs et un temporaire de dimension $n/2 \times n/2$.

Plus généralement, on peut découper une matrice C de taille $n \times n$ en t^2 blocs de dimension $n/t \times n/t$. Alors on peut calculer chaque bloc avec l'algorithme de Winograd en utilisant une mémoire temporaire de taille $\left(n/t\right)^2$. La complexité arithmétique satisfait alors $\mathrm{R}_t(n) = t^2 + t\mathrm{W}_{\mathrm{acc}}\left(n/t\right)$, qui se résout en $\mathrm{R}_t(n) = 6t^{3-\omega}n^{\omega} - 4tn^2$. On peut alors trouver le meilleur paramètre t. Par exemple, avec $t = 2$,

$$\mathrm{R}_2 = 6{,}857n^{\omega} - 8n^2 \qquad \text{et} \qquad \mathrm{ExtraMem} = \frac{n^2}{4}$$

et avec $t = 3$,

$$\mathrm{R}_3 = 7{,}414n^{\omega} - 12n^2 \qquad \text{et} \qquad \mathrm{ExtraMem} = \frac{n^2}{9}.$$

On remarque que l'on peut utiliser l'ordonnancement Acc (tableau 1.14) et alors la mémoire supplémentaire utilisée devient $2n^2/3t^2$ tandis que la complexité arithmétique croît à $R_t(n) + t^{2-\log_2(3)} n^{\log_2(6)} - tn^2$.

1.2.5 Synthèse.

Nous proposons dans le tableau 1.15 une synthèse des résultats obtenus dans ce chapitre. Nous avons réussi à réduire les besoins en mémoire pour l'opération $C \leftarrow \alpha A \times B + \beta C$ avec des entrées constantes (de n^2 à $2/3n^2$). L'ajout d'opérations supplémentaires (les pré-additions, les recalculs) fait certes croître les complexités mais c'est en quelque sorte amorti par les gains en mémoire.

Lorsque les opérandes peuvent être écrasés, nous avons un algorithme IP complètement en place pour l'opération $C \leftarrow A \times B$ sans changer la complexité arithmétique. Des variantes de ces opération (où l'on peut écraser certains opérandes) nous ont permis de réduire efficacement la mémoire supplémentaire dans $C \leftarrow \alpha A \times B + \beta C$.

Nous avons aussi considérablement amélioré et complété les résultats présentés dans [BDPZ09].

1.2.6 Implantation des nouveaux ordonnancements.

Nous nous intéressons maintenant à l'implantation de ces différents algorithmes. Si toutes les complexités précédemment énoncées étaient valables pour des algorithmes appelés jusqu'au dernier niveau, nous ne pousserons pas la récursion jusqu'au cas de base trivial (*cf.* section 5.2.1). Nous concluons donc par quelques mesures de performances . Nous notons aussi que la quantité A d'allocations nécessaires n'entre pas directement en compte dans notre implantation car nous pré-allouons un bloc de mémoire dans lequel seront puisés les espaces mémoires des blocs temporaires, à la manière d'une mémoire tampon (*buffer*) ; la quantité A donne alors peut-être une indication sur le nombre d'indirections.

Première implantation.

Les routines précédentes ont été d'abord implantées en C. Nous montrons dans le tableau 1.16 les performances de diverses procédures. Nous avons utilisé la routine dgemm (multiplication de matrice sur des double) fournie par atlas-3.9.4 et un seuil de 1 024. Nous observons que les nouveaux ordonnancement sont très compétitifs et rendent possible des tailles de matrices plus importantes (nous notons 'MT' pour « *memory thrashing* »)

Tab. 1.16 — Multiplication de matrices rectangulaires : temps de calculs en secondes sur Joran@imag.

Dims. (m, k, n)	Classique	[DHSS94]	IPMM	IPOvMM
(4 096, 4 096, 4 096)	14,03	11,93	13,59	11,98
(4 096, 8 192, 4 096)	28,29	23,39	27,16	23,88
(8 192, 8 192, 8 192)	113,07	85,97	98,75	85,02
(8 192, 16 384, 8 192)	231,86	MT	197,24	170,72

Nous avons donc trouvé des algorithmes performants en pratique qui limitent voire éliminent l'utilisation de mémoire supplémentaire. Nous reviendrons dans la section 5.2, sur l'implantation de ces algorithmes dans LinBox. Notons finalement l'utilisation dans m4ri de certains de nos ordonnancements (*cf.* src/strassen.c dans le source de m4ri).

1.3 Prolongements.

Parmi les prolongements les plus directs figure une automatisation des recherches précédentes de manière par exemple à pouvoir répliquer tout ce travail sur d'*autres algorithmes* comme celui de Bodrato (algorithme 0.3) ou sur une formule pour des matrices de tailles plus générales (3×3...).

Tab. 1.15 — Complexités des différents ordonnancements présentés dans ce chapitre

	Algorithme ou ordonnancement	entrées écrasées ? (□ si carrées)	nb blocs temp.	mémoire suppl. (E)	nb d'allocations suppl. (A)	complexité arithmétique (W)
$A \times B$	`wino` (tab. 1.1, [DHSS94])	constantes	2	$2/3 n^2$	$2/3(n^\omega - n^2)$	$6n^\omega - 5n^2$
	`IP` (tableau 1.5)	A, B écrasées ; □	0	0	0	$6n^\omega - 5n^2$
	`OvL`, `OvR` (tab. 1.7, 1.8)	A ou B écrasées ; □	1	$1/3 n^2$	$1/4 n^2 \log_2(n)$	$6n^\omega - 5n^2$
	`IPMM` (algorithme 1.4)	constantes	0	0	0	$7{,}2n^\omega - 13n^2$
$\alpha A \times B + \beta C$	`wacc` (tab. 1.2, [HLJJ+96b])	constantes	3	n^2	$2/3 n^\omega + n^{\log_2(5)} - 5/3 n^2$	$6n^\omega - 4n^2$
	`accw` (tableau 1.3)	constantes	2	$2/3 n^2$	$2/3 \left(n^\omega - n^2\right)$	$6n^\omega + 2n^{\log_2(6)} - 6n^2$
	`accw2` (tableau 1.4)	constantes	2	$2/3 n^2$	$2/3 \left(n^\omega - n^2\right)$	$6n^\omega - 4n^{\log_2(3)}$
	`AcLR` (tableau 1.11)	A, B écrasées ; □	2	$2/3 n^2$	$2n^2 - 2n^{\log_2(3)}$	$6n^\omega - 3n^2 - n$
	`AccR` (tableau 1.12)	B écrasée ; □	2	$2/3 n^2$	$3/8 n^2 \log_2(n) + 5/4 n^2 - 2n^{\log_2(3)}$	$6n^\omega - 5/2 n^2 \log_2(n) - 1/2 n \log_2(n)$
	`AccL` (tableau 1.13)	A écrasée ; □	2	$2/3 n^2$	$3/8 n^2 \log_2(n) + 5/4 n^2 - 2n^{\log_2(3)}$	$6n^\omega - 5/2 n^2 \log_2(n) - n^{\log_2(3)}$
	`Acc` (tableau 1.14)	constantes	2	$2/3 n^2$	$2/9 n^\omega - 7/14 n^2 \log_2(n) + 11/12 n^2$	$6n^\omega - 11/6 n^2 + 1/2 n \log_2(n)$
	section 1.2.4	constantes	N/A	$1/4 n^2$	$1/4 n^2$	$6{,}857n^\omega - 8n^2$
	section 1.2.4	constantes	N/A	$1/9 n^2$	$1/9 n^2$	$7{,}414n^\omega - 12n^2$

Il serait aussi intéressant de pouvoir automatiser la recherche des *pré-additions*, ici faites « à la main », par des techniques symboliques par exemple.

L'opération $Y \leftarrow A$ suivie de $A \leftarrow Y$ permet, si Y n'est pas modifié entre temps de faire une « copie » de A au prix de deux opérations quadratiques supplémentaires mais sans utiliser de mémoire supplémentaire. En particulier cette technique pourrait autoriser d'écraser un galet non écrasable (par exemple une entrée constante) car elle serait restaurée à son état initial.

Il serait aussi intéressant de pouvoir générer automatiquement du code avec des conditions sur la localité des appels afin de tester en pratique les meilleurs ordonnancements. Une recherche automatique de formules guidée par le calcul des complexités (p.ex. avec un script en Sage) serait tout à fait intéressant et permettrait d'automatiser une grande partie du travail « à la main » de cette section, voire d'apporter des certifications sur les complexités obtenues. Dans la même veine, même si paralléliser la multiplication rapide de matrice n'est pas tâche aisée, il peut être intéressant de produire des ordonnancements en vue de parallélisme de tâches/données ([RT92, DS02]).

Tous les ordonnancements de type Strassen–Winograd présentés sont utilisés dans le cadre du calcul exact, mais aussi utilisables numériquement. Des études ont été effectuées sur la stabilité des algorithmes de Strassen ou Winograd (**??**) et seraient très vraisemblablement valables pour nos ordonnancements. Cependant, nous ne sommes pas affectés par ces questions tant que nous utilisons des représentations exactes des coefficients.

Chap. 2 Recherche de nouvelles formules de multiplication.

Sommaire

2.1 Recherche automatique de formules de multiplication matricielle... 39
2.1.1 Présentation et formalisation du problème. 39
2.1.2 Préparation de l'implantation de la recherche de formes bilinéaires. . . . 40
2.1.3 Recherche automatique de nouvelles formules. 42
2.1.4 Utilisation de formules approchées sur les corps finis ? 45
2.2 Prolongements. 46

Nous cherchons dans ce chapitre à diminuer l'exposant $\omega = \log_2(7)$ en établissant d'autres formules à la Strassen–Winograd. Nous sommes notamment intéressés par de nouvelles formules pour des matrices rectangulaires ce qui permettrait d'écrire des algorithmes adaptatifs lors des récursions — par exemple dans `IPMM` lorsque les appels récursifs deviennent très « rectangulaires ». Nous allons essayer de construire des outils pour aller dans ces directions. Précédemment, [Lad76, JM86, Smi02] ont cherché à la main ou aidé d'un ordinateur de nouvelles formules. Nous avons voulu étudier et développer leurs techniques pour reproduire leurs résultats, et chercher à les améliorer en tirant par exemple parti de l'évolution matérielle.

2.1 Recherche automatique de formules de multiplication matricielle...

2.1.1 Présentation et formalisation du problème.

Pour fixer dès à présent les notations, nous utiliserons des matrices A et X de tailles $m \times k$, B et Y de tailles $k \times n$ et C ou Z de tailles $m \times n$. Si l'on multiplie A et B nous parlerons de format de multiplication (m, k, n). On peut préciser le nombre de multiplications μ et parler de multiplication de type $(m, k, n) - \mu$.

Dans l'algorithme de Winograd (algorithme 0.2) par exemple, une symétrie apparaît clairement dans le rôle joué par les sous-matrices de A d'une part et celles de B d'autre part.

Définition 2.1. — Soient U et V des matrices de taille $p \times q$. On note $\langle \cdot, \cdot \rangle$ le produit scalaire dit de Frobenius. On a par définition :

$$\langle U, V \rangle = \sum_{i=1}^{p} \sum_{j=1}^{q} U_{ij} V_{ij}.$$

Nous allons rechercher un algorithme qui « ressemble » à l'algorithme de Strassen et on dira qu'on recherche une *formule bilinéaire non commutative* ([Str73]). Lors du produit $A \times B$, on effectuera quelques combinaisons linéaires sur les éléments de A et B avant de les multiplier entre elles et de les réinsérer dans la matrice produit C. Cela revient à chercher un algorithme de la forme suivante.

$$C = \sum_{r=1}^{\mu} \left\langle A, X^{(r)} \right\rangle \left\langle B, Y^{(r)} \right\rangle Z^{(r)} \tag{2.1}$$

où les inconnues sont les 3μ matrices $X^{(r)}$, $Y^{(r)}$ et $Z^{(r)}$. Bien sûr, μ correspond au nombre de multiplications maximal que l'on souhaite réaliser. À titre d'exemple, voici dans le tableau 2.1 les matrices impliquées dans l'algorithme de Winograd.

Tab. 2.1 — L'algorithme de Winograd bis.

$$\begin{array}{lll}
X^{(1)} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} & Y^{(1)} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} & Z^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \\
X^{(2)} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & Y^{(2)} = \begin{pmatrix} 0 & 0 \\ -1 & -1 \end{pmatrix} & Z^{(2)} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \\
X^{(3)} = \begin{pmatrix} 1 & 1 \\ -1 & -1 \end{pmatrix} & Y^{(3)} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} & Z^{(3)} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \\
X^{(4)} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} & Y^{(4)} = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} & Z^{(4)} = \begin{pmatrix} 0 & 0 \\ -1 & 0 \end{pmatrix} \\
X^{(5)} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} & Y^{(5)} = \begin{pmatrix} -1 & 1 \\ 0 & 0 \end{pmatrix} & Z^{(5)} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \\
X^{(6)} = \begin{pmatrix} -1 & 0 \\ 1 & 1 \end{pmatrix} & Y^{(6)} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} & Z^{(6)} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\
X^{(7)} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix} & Y^{(7)} = \begin{pmatrix} 0 & -1 \\ 0 & 1 \end{pmatrix} & Z^{(7)} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}
\end{array}$$

Par la suite, nous appelons *formule* (de multiplication rapide) un tel tableau et c'est ce type de tableaux que nous cherchons à obtenir. Pour une telle formule, nous rappelons que, en notant λ l'exposant principal dans la complexité du produit matriciel, on a :

$$\lambda = \frac{3\ln(\mu)}{\ln(mnk)}.$$

Voici maintenant quelques propriétés de l'équation (2.1). On note δ_{ij} le symbole de Kronecker qui vaut 1 si $i = j$ et 0 sinon. On note comme à l'accoutumée E_{ij} les éléments de la base canonique des matrices $m \times n$, i.e. $\left(E_{ij}\right)_{k,l} = \delta_{ik}\delta_{jl}$.

2.1.2 Préparation de l'implantation de la recherche de formes bilinéaires.

Tout d'abord une telle formule est valable sur l'ensemble des matrices $\mathscr{M}_{m,k} \times \mathscr{M}_{k,n} \times \mathscr{M}_{m,n}$ si et seulement si elle l'est sur la base canonique. On utilise les propriétés $E_{ij}E_{kl} = \delta_{jk}E_{il}$ et $\left\langle E_{ij}, X \right\rangle = X_{i,j}$ pour énoncer que l'équation (2.1) est valide si et seulement si l'on a :

$$\delta_{ae}\delta_{df}\delta_{bc} = \sum_{r=1}^{\mu} X^{(r)}_{a,b} Y^{(r)}_{c,d} Z^{(r)}_{e,f}, \quad \forall 1 \leqslant a, e \leqslant m, \forall 1 \leqslant b, c \leqslant k, \forall 1 \leqslant d, f \leqslant n. \tag{2.2}$$

On obtient donc un système non linéaire d'équations d'inconnues $\left(X^{(r)}, Y^{(r)}, Z^{(r)}\right)_{r\in[\![1,\mu]\!]}$, soit au total $N = (mk + kn + mn)\mu$ inconnues — ou $3\mu n^2$ dans le cas carré.

2.1.2.1 Invariances.

La symétrie de cette relation fait apparaître plusieurs invariances. Pour simplifier leurs expressions, nous notons X le vecteur des matrices $X^{(1)}, \ldots, X^{(\mu)}$. On note aussi ${}^{\top}X$ le vecteur X dont les éléments sont les transposées ${}^{\top}X^{(i)}$ des matrices élémentaires. Si σ est une permutation de $\mathcal{S}_n$, alors le vecteur X^{σ} a pour i^{e} coordonnée $X^{(\sigma(i))}$.

On a alors les invariances suivantes, pour toutes matrices P, Q et R inversibles, pour tous scalaires a, b et c et pour toute permutation σ,

$$(X, Y, Z) \mapsto (X^{\sigma}, Y^{\sigma}, Z^{\sigma}) \tag{2.3}$$

$$(X, Y, Z) \mapsto ({}^{\top}Y, {}^{\top}X, {}^{\top}Z) \tag{2.4}$$

$$(X, Y, Z) \mapsto (aX, bY, cZ) \text{ où } abc = 1 \tag{2.5}$$

$$(X, Y, Z) \mapsto (PXR^{-1}, RYQ^{-1}, {}^{\top}P^{-1}Z\,{}^{\top}Q). \tag{2.6}$$

Ces invariances sont bien naturelles : elles correspondent à réordonner les triplés de matrices élémentaires, ou, lorsque $C = A\times B$, aux relations de transposition ${}^{\top}C = {}^{\top}B\times{}^{\top}A$ ou d'homotétie $cC = aA\times bB$ ou de changement de base $PCQ = PAR^{-1} \times RBQ$.

2.1.2.2 Re-formulation polynomiale.

Nous cherchons à résoudre le systèmes d'équations (2.2) via les racines d'un polynôme. La façon la plus directe est de former le *polynôme* :

$$Q(X, Y, Z) = \sum_{\substack{a\leqslant m\\ b\leqslant k}}\sum_{\substack{c\leqslant k\\ d\leqslant n}}\sum_{\substack{e\leqslant m\\ f\leqslant n}}\left(\delta_{a,e}\delta_{d,f}\delta_{b,c} - \sum_{r=1}^{\mu} X^{(r)}_{a,b}Y^{(r)}_{c,d}Z^{(r)}_{e,f}\right)^2. \tag{2.7}$$

Nous remarquons que l'on a une formule exacte (2.1) si et seulement si $Q = 0$. Minimiser Q sera donc notre objectif ; on verra comment on peut s'y prendre plus bas (section 2.1.3). Naturellement, on peut écrire cette fonction pour une autre norme mais la norme euclidienne nous sera très intéressante.

Un jeu de réécriture permet de trouver, en développant l'équation (2.7), une autre formulation qui fait intervenir moins d'opérations :

$$\begin{aligned} Q(X, Y, Z) &= mkn - 2\sum_{r=1}^{\mu}\sum_{\substack{a\leqslant m, b\leqslant k\\ d\leqslant n}} X^{(r)}_{a,b}Y^{(r)}_{b,d}Z^{(r)}_{a,d} + \\ &\quad \sum_{e,f=1}^{\mu}\left\langle X^{(e)}, X^{(f)}\right\rangle\left\langle Y^{(e)}, Y^{(f)}\right\rangle\left\langle Z^{(e)}, Z^{(f)}\right\rangle. \end{aligned} \tag{2.8}$$

Cette formule est plus agréable et elle conserve les symétries. En effet, on peut par exemple écrire que :

$$\begin{aligned} \sum_{\substack{a\leqslant m, b\leqslant k\\ c\leqslant n}} X^{(r)}_{a,b}Y^{(r)}_{b,c}Z^{(r)}_{a,c} &= \left\langle Z^{(r)}, X^{(r)}Y^{(r)}\right\rangle \\ &= \left\langle X^{(r)}, Y^{(r)}\,{}^{\top}Z^{(r)}\right\rangle \\ &= \left\langle Y^{(r)}, {}^{\top}X^{(r)}Z^{(r)}\right\rangle. \end{aligned}$$

Nous allons maintenant chercher à minimiser ce polynôme. Pour cela nous avons besoin de le dériver deux fois.

2.1.2.3 Différentiation de la formulation polynomiale.

L'écriture (2.8) permet de dériver facilement par rapport à chacune des variables :

$$\begin{aligned}
\frac{\partial Q}{\partial X^{(u)}_{s,t}} &= -2\left(Y^{(u)\,\top}Z^{(u)}\right)_{s,t} + 2\sum_{r\leqslant\mu} X^{(r)}_{s,t}\left\langle Y^{(r)}, Y^{(u)}\right\rangle\left\langle Z^{(r)}, Z^{(u)}\right\rangle \\
\frac{\partial Q}{\partial Y^{(u)}_{s,t}} &= -2\left({}^{\top}X^{(u)}Z^{(u)}\right)_{s,t} + 2\sum_{r\leqslant\mu} Y^{(r)}_{s,t}\left\langle X^{(r)}, X^{(u)}\right\rangle\left\langle Z^{(r)}, Z^{(u)}\right\rangle \\
\frac{\partial Q}{\partial Z^{(u)}_{s,t}} &= -2\left(X^{(u)}Y^{(u)}\right)_{s,t} + 2\sum_{r\leqslant\mu} Z^{(r)}_{s,t}\left\langle X^{(r)}, X^{(u)}\right\rangle\left\langle Y^{(r)}, Y^{(u)}\right\rangle.
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 Q}{\partial X^{(u_1)}_{s_1,t_1}\partial X^{(u_2)}_{s_2,t_2}} &= 2\delta_{s_1s_2}\delta_{t_1t_2}\left\langle Y^{(u_1)}, Y^{(u_2)}\right\rangle\left\langle Z^{(u_1)}, Z^{(u_2)}\right\rangle \\
\frac{\partial^2 Q}{\partial Y^{(u_1)}_{s_1,t_1}\partial Y^{(u_2)}_{s_2,t_2}} &= 2\delta_{s_1s_2}\delta_{t_1t_2}\left\langle X^{(u_1)}, X^{(u_2)}\right\rangle\left\langle Z^{(u_1)}, Z^{(u_2)}\right\rangle \\
\frac{\partial^2 Q}{\partial Z^{(u_1)}_{s_1,t_1}\partial Z^{(u_2)}_{s_2,t_2}} &= 2\delta_{s_1s_2}\delta_{t_1t_2}\left\langle X^{(u_1)}, X^{(u_2)}\right\rangle\left\langle Y^{(u_1)}, Y^{(u_2)}\right\rangle.
\end{aligned}$$

Fixons Y et Z. Alors Q est quadratique en X. Soit H sa matrice hessienne en 0, de taille $mk\mu$ et soit $-D$ son vecteur gradient, évalué en 0. Alors, on a $Q(X) = Q(0) - DX + \frac{1}{2}\,{}^{\top}XHX$. Minimiser Q revient alors à résoudre le système $HX = D$ si H est inversible ; sinon, on cherchera à minimiser $\|HX - D\|$ par exemple. C'est la base de notre algorithme de minimisation.

Nous avons maintenant amené tous les outils nécessaires pour chercher de nouvelles formules (semi-)automatiquement : nous expliquons par la suite quels problèmes nous avons rencontrés et quelle implantation nous en avons faite.

2.1.3 Recherche automatique de nouvelles formules.

Nous décrivons maintenant toutes les méthodes implémentées pour essayer de trouver une formule. Le premier problème est l'apparition de formules à coefficients réels dont le polynôme Q est petit mais non nul. Or, nous sommes intéressés par des coefficients qui sont essentiellement égaux à 0 et ± 1, voire de petits coefficients entiers ou rationnels. Comment obtenir des formules de « meilleure qualité » ?

2.1.3.1 Arrondir.

On peut arrondir aux rationnels et/ou aux entiers les plus proches. On peut n'arrondir que les valeurs très proches d'entiers et/ou de certains rationnels à petit dénominateur, pénaliser les termes qui s'éloignent trop de 0. En effet, un ennui important est l'existence de solutions approchées dont des termes grossissent vers l'infini, un peu comme un « zéro à l'infini ». Il nous faut donc aussi essayer de contraindre notre recherche près de zéro. Une solution est la suivante :

2.1.3.2 Maîtriser les coefficients.

On ajoute un poids quadratique de la forme $\varepsilon\times\left(\sum\left(X^{(u)}_{s,t}\right)^2 + \sum\left(Y^{(u)}_{s,t}\right)^2 + \sum\left(Z^{(u)}_{s,t}\right)^2\right)$ pour éviter que les termes ne partent à l'infini. Ce poids garde la simplicité de l'écriture lors de la différenciation. On introduit un ε « à la main » et par essais successifs car on cherche à ce que le poids n'aie pas un effet négligeable ou prépondérant. Il est aussi important de vérifier qu'un coefficient ne prend pas de

grandes valeurs (« grand » signifiant de l'ordre de la dizaine ou la centaine pour les petites valeurs de (m, k, n)) car cela conduit rapidement à des dépassements de capacité.

2.1.3.3 Réduire le nombre d'inconnues et normaliser.

Grâce à la transformation (2.6), on peut utiliser des matrices de passage pour convertir une formule vers une forme plus agréable. Par exemple, on peut essayer de transformer un certain nombre de matrices en matrices triangulaires, ou diagonales, sous forme normale. Par exemple on peut vouloir mettre $X^{(1)}$ sous une forme normale puis utiliser une autre transformation, par exemple QR sur ${}^{\top}Z^{(1)}$. Ensuite, on peut utiliser la transformation (2.5) pour se rapprocher de rationnels sur la ligne 1.

On peut aussi demander à chercher une formule en fixant certains coefficients, par exemple avec $X^{(1)} = \begin{pmatrix} * & 0 \\ 0 & * \end{pmatrix}$ et $Z^{(1)} = \begin{pmatrix} * & 0 \\ * & * \end{pmatrix}$. Nous avons donc permis à ce que l'on puisse bloquer certains termes en les forçant à certaines valeurs, de façon à réduire la taille de l'espace dans lequel on minimise ou à aider et diriger la recherche.

2.1.3.4 Implantation.

Nous avons utilisé les BLAS, LAPACK, GSL [1] pour implémenter ces opérations (Schur, QR, décomposition en valeurs singulières, multiplication matricielle...). Nous avons commencé par faire un petit programme avec une interface toute simple dans un terminal, tout d'abord très basique (*cf.* figure 2.1), puis en utilisant des séquences d'échappement VT100, puis en ncurses [2]. Cependant, devant les difficultés rencontrées, nous avons fini par considérer la très portable bibliothèque Qt-4 [3], qui avec son système de « signal–slot » permet d'écrire rapidement et aisément, en C++, des interfaces graphiques puissantes et interfactives. En outre, cette bibliothèque est intéressante par le grand nombre de contrôles (*widgets*) proposé, leur simplicité d'utilisation, ses modules « tout prêts « comme les piles faire/défaire (*undo/redo*), ou la gestion de fichiers, notamment en XML pour les imports et sorties de formules. Finalement, de manière à ne pas dépendre d'une interface graphique dans l'évolution de ce logiciel, le code de calculs est totalement indépendant et séparé de l'interface graphique (patron de conception MVC, modèle-vue-controlleur).

Nous présentons dans la figure 2.2 une capture d'écran du logiciel `winosearcher`.

2.1.3.5 Résultats.

Nous retrouvons assez rapidement une formule du type Strassen–Winograd. Par contre nous n'avons pas encore trouvé de nouvelles formules exactes pour des tailles plus grandes. Nous présentons quelques résultats dans la tableau 2.2.

Tab. 2.2 — Résultats (prémisses)

Type		Q	Type		Q
$(2,2,2)$	7	0	$(2,2,2)$	6	$1+\varepsilon$
$(2,2,3)$	10	0,000 003 8	$(2,2,3)$	9	1,000 54
$(2,3,2)$	10	0,000 037 9	$(2,3,2)$	9	1,000 02
$(2,4,2)$	14	0	$(2,4,2)$	13	1,000 34
$(3,3,3)$	23	0,001 351	$(3,3,3)$	22	1,001 29

Ces résultats approchés sont accessibles en quelques minutes. Par contre, converger vers des solutions entières ou rationnelles est un sujet plus délicat. Au cours de nos expérimentations, nous constatons que les minima de Q obtenus sont presque toujours à peine supérieurs à des entiers et que lorsqu'ils

1 — GNU Scientific Library, disponible à l'url `http://www.gnu.org/software/gsl/`.
2 — Bibliothèque d'affichage pour console, `http://www.gnu.org/software/ncurses/`.
3 — `http://qt.nokia.com/`, `http://qt-project.org/`.

```
> ./new_forumula -a2 -b2 -c2 -m7
        ××× on cherche une forumule du type :
<2, 2, 2> avec 7 multiplications.
la tolérance est : 1e-17 et le nombre max d'itérations : 5000.
        ××× On lance la minimisation...
        ××× Résultat obtenu : 1.19234e-05
        ××× On fait une pose.
On en profite pour imprimer, voir ./sortie_2_2_2-7.pdf. Le minimum est : 1.19234e-05 pour l'instant.
/----------------------------ACTION-----------------------------------\
|1  : on continue ?     2  : on recommence ?    3  : on arrête ?       |
|------------------------------NORME-----------------------------------|
|4  : arrondir ?        11 : Appliquer Schur ?  12 : en rationnels ?   |
|13 : homotéties ?      18 : SVD + QR ?         21 : auto_homot ?      |
|-------------------------IMPRESSION-----------------------------------|
|15 : le stdout ?       5  : pour maple ?       8  : en latex ?        |
|----------------------------CHANGER-----------------------------------|
|6  : la tolérance ?    7  : le nb d'itérations ?                      |
|----------------------------BLOQUER-----------------------------------|
|9  : une ligne ?       14 : un élément ?       20 : SVD + QR ?        |
|---------------------------MODIFIER-----------------------------------|
|10 : une ligne ?       16 : un élément ?                              |
|-------------------------CONTRAINTES----------------------------------|
|19 : rajouter un poids ?                                              |
|----------------------------AUTO--------------------------------------|
|              17 : Automotatique (heuristique...) ?                   |
\----------------------------------------------------------------------/
```

Fig.. 2.1 — Un exemple de prompt.

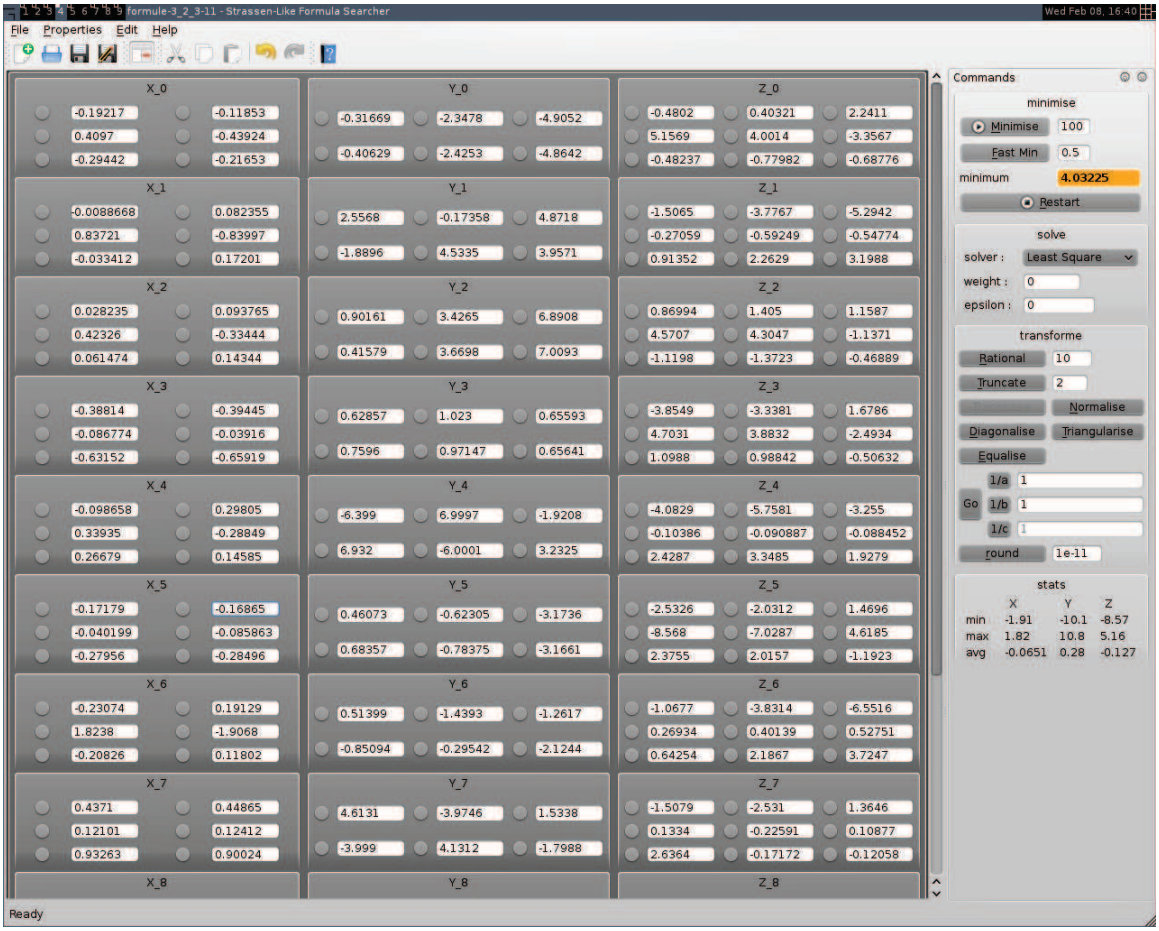

Fig.. 2.2 — Une capture d'écran de l'interface Qt.

sont près de zéro, il existe une formule connue. Il est très difficile pour l'optimiseur de descendre au dessous d'une barrière entière sans que nous ne sachions expliquer ce phénomène.

Les formules obtenues pour de petits Q ne sont pas sans valeur. En effet, elles nous permettent d'obtenir des formules approchées, très utiles en théorie (p.ex. [Bin80]) et qui ne semblent pas dénuées d'intérêt en pratique sur des corps finis comme nous allons le voir dans la sous-section suivante.

2.1.4 Utilisation de formules approchées sur les corps finis ?

Nous appellerons ici pour simplifier *formule approchée* une égalité de la forme $C = A_\varepsilon \times B_\varepsilon + D(\varepsilon)$ avec $D(\varepsilon) \underset{\varepsilon \to 0}{\longrightarrow} 0$. Soit la formule approchée suivante (tableau 2.3) due[4] à [Bin80] qui fournit une multiplication approchée de type $(3, 2, 2)$ en 10 multiplications (et donc $\omega \approx 2.78$).

Tab. 2.3 — Formule approchée $(3, 2, 2)$ de Bini.

$$X^{(1)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad Y^{(1)} = \begin{pmatrix} \varepsilon & 0 \\ 0 & 1 \end{pmatrix} \quad Z^{(1)} = \begin{pmatrix} \varepsilon^{-1} & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

$$X^{(2)} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad Y^{(2)} = \begin{pmatrix} 0 & 0 \\ -1 & -1 \end{pmatrix} \quad Z^{(2)} = \begin{pmatrix} \varepsilon^{-1} & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

$$X^{(3)} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \quad Y^{(3)} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad Z^{(3)} = \begin{pmatrix} -\varepsilon^{-1} & -\varepsilon^{-1} \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

$$X^{(4)} = \begin{pmatrix} 0 & \varepsilon \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad Y^{(4)} = \begin{pmatrix} -\varepsilon & 0 \\ 1 & 0 \end{pmatrix} \quad Z^{(4)} = \begin{pmatrix} \varepsilon^{-1} & 0 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$$

$$X^{(5)} = \begin{pmatrix} 1 & \varepsilon \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \quad Y^{(5)} = \begin{pmatrix} 0 & \varepsilon \\ 0 & 1 \end{pmatrix} \quad Z^{(5)} = \begin{pmatrix} 0 & \varepsilon^{-1} \\ 0 & -1 \\ 0 & 0 \end{pmatrix}$$

$$X^{(6)} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad Y^{(6)} = \begin{pmatrix} 1 & 0 \\ 0 & \varepsilon \end{pmatrix} \quad Z^{(6)} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & \varepsilon^{-1} \end{pmatrix}$$

$$X^{(7)} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} \quad Y^{(7)} = \begin{pmatrix} -1 & -1 \\ 0 & 0 \end{pmatrix} \quad Z^{(7)} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & \varepsilon^{-1} \end{pmatrix}$$

$$X^{(8)} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \quad Y^{(8)} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad Z^{(8)} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ -\varepsilon^{-1} & -\varepsilon^{-1} \end{pmatrix}$$

$$X^{(9)} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ \varepsilon & 0 \end{pmatrix} \quad Y^{(9)} = \begin{pmatrix} 0 & 1 \\ 0 & -\varepsilon \end{pmatrix} \quad Z^{(9)} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & \varepsilon^{-1} \end{pmatrix}$$

$$X^{(10} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ \varepsilon & 1 \end{pmatrix} \quad Y^{(10)} = \begin{pmatrix} 1 & 0 \\ \varepsilon & 0 \end{pmatrix} \quad Z^{(10)} = \begin{pmatrix} 0 & 0 \\ -1 & 0 \\ \varepsilon^{-1} & 0 \end{pmatrix}$$

L'utilisation d'une telle formule se prête particulièrement aux `double` pour p.ex. représenter des entiers modulaires. Pour que le résultat d'une telle multiplication soit correcte, il suffit de choisir un ε tel que $D(\varepsilon) < 1/2$. Dans le cas de l'algorithme tiré du tableau 2.3, en utilisant une multiplication

4 — En fait, certains termes de W dans la référence sont incorects, nous les avons corrigés ici.

« exacte » (dgemm) sur des doubles pour chacune des 10 multiplications, nous avons, pour $\varepsilon < 1/2$, l'inégalité $D(\varepsilon) \leqslant 7\varepsilon\|AB\|_\infty$. Par exemple, avec la représentation standard des éléments des anneaux $\mathbb{Z}/p\mathbb{Z}$, nous obtenons $D(\varepsilon) \leqslant 7n\left(p-1\right)^2$. Nous choisissons $\varepsilon = 2^{-26}$ de manière à ce que $\varepsilon^2 = 0$. Le plus grand modulo que nous pouvons alors utiliser est donné par $n/2\left(p-1\right)^2 < 2^{26}$. En particulier, pour $p = 101$, nous pourrions utiliser des dimensions jusque 13 000 — à compter d'avoir assez de mémoire...

Nous proposons dans le tableau 2.4 suivant un comparatif pour l'opération $C \leftarrow AB$ entre l'algorithme approché de Bini avec un seul niveau de récursion sur des `double` et `FFLAS::fgemm` sur un corps `Modular<double>`.

Tab. 2.4 — Algorithme approché de Bini *vs.* Winograd (`fgemm`) sur $\mathbb{Z}/101\mathbb{Z}$ avec seuil de 1 000.

	temps (s)	
dimensions	`fgemm-bini`	`fgemm-wino`
(1 080, 1 080, 1 080)	0,33	**0,32**
(1 800, 1 800, 1 800)	1,37	**1,35**
(2 700, 2 700, 2 700)	**4,38**	4,56
(2 700, 1 800, 1 800)	**3,18**	3,24
(4 500, 1 800, 1 800)	**1,95**	1,98
(4 050, 2 700, 2 700)	**6,35**	6,75

Nous constatons que notre première implantation de l'algorithme $(3, 2, 2)$ de Bini est compétitive avec `fgemm`, en particulier pour les multiplications rectangulaires. En écrivant de même des algorithmes dérivés pour les cas symétrique $(2, 3, 2)$ ou $(2, 2, 3)$, nous pourrions, pour les petits anneaux $\mathbb{Z}/p\mathbb{Z}$, bénéficier d'une accélération — d'autant plus que l'algorithme implémenté pourrait être encore travaillé au niveau de l'ordonnancement des additions et des opérations avec ε.

2.2 Prolongements.

Le programme `winosearcher` peut encore être amélioré de diverses manières. D'une part, pour éviter les problèmes liés aux arrondis et aux dépassements de capacité, il nous faudrait utiliser des flottants de plus grande précision — par exemple des flottants multi-précision. Il serait aussi intéressant de paralléliser le code de calcul. Comme nous sommes partis d'une interface où l'utilisateur guide les recherches, il n'est pas pour le moment destiné à distribuer des recherches sur un *cluster* par exemple. Si nous pouvons relaxer l'influence de l'utilisateur sur les recherches, nous pourrions imaginer, avec un *cluster* de N ordinateurs, lancer simultanément N recherches avec chacune un paramètre fixé.

Une autre idée que nous avons essayé de suivre consiste à énumérer toutes les formules valables dans $\mathbb{Z}/2\mathbb{Z}$ et de les remonter sur $\mathbb{Z}$ ([Bod10]). Cependant, nous n'avons pas réussi à réduire suffisamment le nombre d'inconnues et diminuer la combinatoire pour les tailles intéressantes comme $(3, 3, 3)$.

II

Multiplication matrice creuse et vecteur dense.

Introduction.

Sommaire

Pourquoi l'opération SpMV ? 49
Généralités sur SpMV. 50
Formats de stockage. 50
Outils logiciels. 50
L'évolution du matériel. 51
Plan du chapitre. 52

Au cours des dernières années, un énorme travail a été effectué pour rendre les routines d'algèbre linéaire dense sur des corps premiers de la taille d'un mot machine extrêmement efficace ([DGP02]). Une des idées clefs fut de mettre à profit les routines numériques existantes (BLAS, LAPACK). Ces routines sont bien définies, avec une grande communauté d'utilisateurs et des distributions libres (ATLAS, Goto2) ou propriétaires (ACML, MKL) extrêmement performantes. Un développement logiciel peut donc être bâti sainement sur ces bases. Dans [DGP08], une étude approfondie est menée sur ces techniques et les performances obtenues.

Pourquoi l'opération SpMV ?

Cependant, dans LinBox en particulier, il existe encore de nombreux domaines où les opérations matricielles de base ne sont pas optimisées : BLAS sur les corps non premiers, sur les entiers et les rationnels, opérations sur les matrices structurées et/ou creuses.

Dans ce chapitre, nous allons optimiser le produit d'une matrice creuse par un vecteur dense. C'est l'opération de base dans les algorithmes dits en « boîte noire » (*black box*, [KL94]) lorsque la boîte noire est une matrice creuse, ou une composée de matrices creuses par exemple. Il est donc crucial de rendre cette opération performante. Par exemple, cette opération est au cœur des algorithmes de type Lanzcos ou Wiedemann. Elle est aussi indispensable pour calculer le produit $A \cdot B \cdot \mathbf{v}$ sans avoir à calculer le produit $A \cdot B$ qui est de trop grande taille (et souvent dense).

Nous appelons par la suite SpMV (**s**parse **m**atrix-**v**ector product) cette opération. Nous implémentons une opération SpMV sur des anneaux finis $\mathbf{Z}/q\mathbf{Z}$ avec q de la taille d'un mot machine. Plus précisément, nous nous baserons sur l'opération $\mathbf{y} \leftarrow A \cdot \mathbf{x} + \mathbf{y}$ pour en déduire l'opération plus générale $\mathbf{y} \leftarrow \alpha A \cdot \mathbf{x} + \beta\, \mathbf{y}$, avec $\alpha \neq 0$. En particulier, l'opération `apply` de l'interface boîte noire de LinBox est réalisée par $\mathbf{y} \leftarrow 0; \mathbf{y} \leftarrow A \cdot \mathbf{x} + \mathbf{y}$. Nous produirons d'autre part une routine $\mathbf{y} \leftarrow {}^{\top}A \cdot \mathbf{x} + \mathbf{y}$. Enfin,

nous spécifierons ces opérations dans le cas où $\mathbf{x}$ et $\mathbf{y}$ sont des blocs de vecteurs (c'est-à-dire des matrices avec peu de colonnes par rapport aux lignes). Cette opération particulière rejoint une opération plus générale sur les matrices denses notée alors SpMM, le second 'M' étant pour matrice et est capitale pour les algorithmes par blocs. Les routines $\mathbf{y} \leftarrow A^s \cdot \mathbf{x} + \mathbf{y}$ ne sont fournies dans le cas où A est carrée qu'afin de comparer les performances entre une seule itération et des utilisations répétées typiques des processus itératifs.

Généralités sur SpMV.

Disons qu'une matrice creuse de taille n comprend $\mathcal{O}(n)$ entrées non nulles. En général, on peut considérer qu'il y en a cn, avec c une constante très petite devant 1, appelée sa densité ou sparsité (*sparsity*). La quantité c est souvent aussi considérée comme une approximation du nombre d'entrées non nulles sur une ligne. C'est un cas qui se retrouve souvent en pratique même si on doit pouvoir aussi traiter des distributions plus générales. La complexité du produit matrice vecteur est de $\mathcal{O}(n)$ opérations élémentaires. Elle est minimale dans le sens où elle est égale à la taille de la matrice creuse.

C'est une opération rapide, seulement en n, mais en pratique, nombre d'accès se font de manière aléatoire et il est donc difficile d'optimiser l'utilisation du cache et les défauts de page [HHM10, TS06]. C'est en fait, la difficulté principale pour une implantation efficace. D'autre part, il existe une diversité de formats de stockage que l'algèbre linéaire dense ne connaît pas (en général, une matrice dense est stockée sous un format BLAS selon ses colonnes ou ses lignes d'abord), ce qui introduit une difficulté logicielle supplémentaire.

Formats de stockage.

Il existe donc une multitude de formats de stockage. Certains reflètent des structures de données variées, d'autres sont apparus pour optimiser certaines opérations, notamment SpMV. Nous allons en lister les plus courants dans le chapitre 3.

Outils logiciels.

Les numériciens nous proposent quelques outils. Nous allons en citer quelques uns des plus connus. Nous reprenons ici quelques projets :

— sparseBLAS (`http://math.nist.gov/spblas/`) assez simple et complet ;
— sparselib++ (*cf.* [ALN+94], `http://math.nist.gov/sparselib++/`) plus complet, écrit et développé pour le C++ ;
— cxsparse (`http://www.cise.ufl.edu/research/sparse/CXSparse/`) est très complet et à ce jour encore développé et présent dans de nombreuses distribution Linux ;
— OSKI (*cf.* [VDY05, VM05]) une librairie qui s'auto-optimise mais n'est plus développé ;
— PSBLAS (`http://www.ce.uniroma2.it/psblas/`), une implantation style BLAS et parallèle (utilise MPI).
— Les spécifications BLAS incluent les *Sparse BLAS* [5] mais ces routines sont rarement complètement implémentées dans les implantations libres des BLAS.
— Metis [6] permet de partitionner une matrice pour l'optimiser par blocs.

Nous voyons que tous les projets ne sont pas tous régulièrement maintenus. Il n'y a en général pas de compatibilité directe entre eux (au niveau des formats) ni de leur interface (à part les rares respectant les spécifications BLAS). En outre, aucun n'est développé pour du parallélisme à mémoire partagée.

5 — `www.netlib.org/blas/blast-forum/chapter3.pdf`
6 — `http://glaros.dtc.umn.edu/gkhome/views/metis`

L'évolution du matériel.

Environnements multi-cœurs.

Comme nous l'avons déjà mentionné, il est impératif de considérer à la fois les versions séquentielles et à mémoire partagée des algorithmes que nous développons, pour tirer meilleur parti des architectures multi-cœurs aujourd'hui classiques, mais aussi pour laisser les algorithmes (au dessus) choisir la version séquentielle ou parallèle de SpMV. Nous n'avons pas connaissance de version multi-fils de SpMV disponibles. Nous avons choisi OpenMP (*Open Multi-Processing*) comme outil de parallélisation. Les raisons de ce choix sont multiples : une interface de programmation simple est très documentée, un support par de nombreux compilateurs (icc, gcc...) et une hybridisation facile avec MPI ou d'autres technologies.

Environnements graphiques.

Avec l'apparition puis la généralisation des cartes graphiques puissantes, le standard graphique OpenGL (*Open Graphics Library*) permettait de faire du calcul intensif (notamment via GLSL, *OpenGL Shading Language*) qui facilite l'utilisation des *shaders*. Cependant, programmer des shaders pour le calcul scientifique est encore loin d'un véritable langage de programmation général pour les cartes graphiques.

Les constructeurs de carte graphique, notamment AMD et NVidia, ont créé des langages de programmation pour le calcul de leur cartes graphiques, souvent seulement disponibles pour les dernières sorties et les plus puissantes et délaissant rapidement leur support pour leurs cartes une fois celles-ci vieillies.

Chez ATI, c'est Close to Metal (devenu Stream SDK en 2007, puis APP pour *Accelerated Parallel Processing*) tandis que chez NVidia, c'est Cuda (*Compute Unified Device Architecture*), dont la première version publique est aussi sortie en 2007. Ce sont tous les deux des extensions du langage C. Rapidement le standard OpenCL est apparu, notamment sous l'impulsion d'Apple, et les vendeurs ont fourni des implantations du standard. Pour NVidia, les *drivers* ont été fournis aux développeurs enregistrés en avril 2009, et en août pour AMD. La version 1.0 du support de OpenCL par AMD ou NVidia n'est sortie que fin 2009. Actuellement, la version 1.1 de OpenCL est disponible chez IBM, Intel, AMD, ATI. Le système d'exploitation Linux est supporté par chacune de ces implantations.

Motivations.

Nous voulons donc implanter cette opération sur des architectures parallèles (multi-cœurs et processeurs graphiques). Dans le cas des matrices denses, il existe des versions multi-threadées des BLAS (Goto2) ; il existe aussi une librairie pour Cuda de ATI : cuBLAS. Nous connaissons les accélérations liées à l'utilisation des BLAS parallèles. Celles liées aux BLAS sur GPU semblaient prometteuses (mi 2009 les performances de crête étaient quasiment atteintes, des tests rapides montrent que cette technologie est potentiellement extrêmement intéressante pour les opérations FFLAS).

Ces précédents nous motivent donc dans la volonté de parallélisation de l'opération (plus difficile que l'algorithme triple boucle) SpMV.

Pourquoi Cuda ?

La bibliothèque développée pour faire l'opération SpMV sur les corps fini se nomme ffspmvgpu [7]. Il fallait choisir un langage de programmation. Les cartes graphiques disponibles étaient conçues par NVidia et au début de l'écriture du code, le langage OpenCL n'était pas encore disponible ; Cuda en était déjà à sa version 2.0. Même s'il manquait de nombreuses fonctionnalités à Cuda (qui à l'heure de la rédaction en est à la version 4.1, beaucoup plus aboutie), ce langage semblait mûrir et avait une communauté d'utilisateurs grandissante. Un inconvénient majeur, dont nous étions conscient dès le départ est l'enfermement dans une solution propriétaire. Lors du portage du code dans LinBox, une réécriture avec le choix d'OpenCL à la place de Cuda sera vraiment considérée. Cela permettra une indépendance (toute relative pour le moment) envers les constructeurs et une généricité plus propre à LinBox.

7 — disponible sur `https://forge.imag.fr/projects/ffspmvgpu/`.

Travaux précédents sur SpMV sur GPU.

Quelques noyaux (*kernel*) ont été proposés pour l'opération SpMV sur GPU : [VGMF09, BFF+09, BG09]. Ce dernier a participé à la création — postérieure à ffspmvgpu — de la bibliothèque cuSPARSE [8] publiée par NVidia.

Plan du chapitre.

Tout d'abord, nous énoncerons les formats de stockage les plus connus (chapitre 3). Ensuite, nous décrirons les algorithmes utilisés et les comparerons. L'idée consiste à chercher à tirer profit des méthodes numériques, bien que le problème soit différent des FFLAS. En effet, contrairement au cas dense, il est difficile d'extraire une sous-matrice d'une matrice creuse. Pour améliorer les performances de SpMV, nous avons suivi les axes suivants.

— Nous introduirons de nouveaux formats et ferons des adaptations de formats existants pour en tirer des optimisations dans le cas des corps finis.
— Nous essaierons aussi de tirer automatiquement parti des architectures (par exemple multi-cœurs+GPU) pour gagner encore en rapidité.
— Nous proposons des formats hybrides, spécifiés par l'utilisateur ou découverts heuristiquement, une matrice creuse étant alors découpée en différentes parties selon différents formats pour tirer parti des architectures des machines.

Dans le chapitre 4, nous utiliserons cette bibliothèque pour améliorer l'algorithme par blocs de Wiedemann dans LinBox. Une application directe concerne alors le calcul du rang.

8 — Voir http://developer.nvidia.com/cusparse.

Chap. 3 Présentation de ffspmvgpu.

Sommaire

3.1 Formats traditionnels. 53
3.2 Implantation de base. 57
3.2.1 Représentation des corps finis. 57
3.2.2 Adapter les bibliothèques numériques. 59
3.2.3 Utilisation d'OpenMP. 59
3.3 Nouveaux formats. 59
3.3.1 Premier essai : compilation à la volée (*JIT*). 61
3.3.2 Prise en compte des ± 1. 62
3.3.3 Retour sur les formats de base. 63
3.3.4 Formats hybrides. 64
3.3.5 Algorithme heuristique pour le choix des formats. 64
3.4 Version itératives et par blocs de vecteurs. 64
3.4.1 Utiliser les multi-vecteurs. 65
3.4.2 Performance et problèmes. 65
3.5 Prolongements. 67

Nous commençons par présenter des formats usuels dans les logiciels ou la littérature (section 3.1) puis nous indiquons une première implantation de SpMV et les problèmes soulevés (section 3.2) avec comme première réponse de nouveaux formats (section 3.3). Finalement, dans la section 3.4, nous généralisons SpMV à des blocs de vecteurs.

3.1 Formats traditionnels.

Soit la matrice suivante qui va nous servir d'exemple dans tous les paragraphes suivants. Nous montrons comment elle est représentée dans divers formats de stockage.

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 8 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Format COO.

Le format de stockage par *coo*rdonnées est le format le plus intuitif et le plus commun. Il stocke un triplé de vecteurs[1] de taille nbnz, nommés data, idcol et idlig, tels que data[k] = A[idlig[k], idcol[k]]. Souvent, data est stocké ligne d'abord (comme des vecteurs C), rendant idlig croissant. On écrit A sous la forme :

idlig	0	0	0	1	1	1	3	3
idcol	1	2	3	0	4	6	2	6
data	1	2	3	1	8	4	5	1

Fig.. 3.1 — Format COO

Format CSR.

Le format CSR (*C*ompressed *S*torage *R*ow) stocke les lignes de manière plus efficace. Le vecteur idlig est remplacé par un vecteur de taille (lig + 1), nommé debut, tel que, pour la ligne i, si debut[i] $\leqslant k <$ debut[$i+1$], alors data[k] = A[i, idcol[k]]. En d'autres termes, debut indique à quel indice dans les deux autres champs une ligne commence et à quel indice elle s'est terminée.

debut	0	3	6	6	8			
idcol	1	2	3	0	4	6	2	6
data	1	2	3	1	8	4	5	1

Fig.. 3.2 — Format CSR

Format CSC.

C'est le même format que CSR mais ce sont les colonnes (idcol) qui sont compressées dans ${}^{\top}$A (lorsque l'on stocke les éléments non nuls de A colonne d'abord).

idlig	1	0	0	3	0	1	1	3
debut	0	1	2	4	5	6	6	8
data	1	1	2	5	3	8	4	1

Fig.. 3.3 — Format CSC

Nous notons que dans ce cas il n'y a pas de compression car le nombre d'entrée par colonne est proche de un.

Format ELL.

Le format ELL (*ELL*pack) essaie de rendre le stockage des donnés plus dense. Les vecteurs data et idcol sont tels que data[i, j_0] = A[i, idcol[i, j_0]], où j_0 varie entre 0 et le nombre maximum d'entrées non nulles sur une ligne de A. Ces deux tableaux peuvent être stockés colonne (style Fortran) ou ligne (C) d'abord.

1 — par analogie avec le langage C, nous notons **v**[0] le premier élément de **v**

$$\begin{array}{r|l} \texttt{idcol} & \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 0 & 0 & 0 \\ 2 & 6 & 0 \end{pmatrix} \\ \texttt{data} & \begin{pmatrix} 1 & 2 & 3 \\ 1 & 8 & 4 \\ 0 & 0 & 0 \\ 5 & 1 & 0 \end{pmatrix} \end{array}$$

Fig.. 3.4 — Format ELL

Ce format permet de stocker assez proprement des matrices dont la distribution des poids de lignes est constante (ou presque). Si certaines lignes sont de très grand poids, on devra allouer un espace mémoire beaucoup trop grand.

Format ELL_R.

Le format ELL_R est une variant de ELL, introduit notamment dans [VGMF09]. On ajoute un vecteur nblig de taille lig qui indique combien d'éléments non nuls il y a par ligne (supprimant ainsi le rôle des 0 terminant chaque ligne de ELL).

$$\begin{array}{r|l} \texttt{nblig} & 3 \;\; 2 \;\; 0 \;\; 2 \\ \texttt{idcol} & \begin{pmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ * & * & * \\ 2 & 6 & * \end{pmatrix} \\ \texttt{data} & \begin{pmatrix} 1 & 2 & 3 \\ 1 & 8 & 4 \\ * & * & * \\ 5 & 1 & * \end{pmatrix} \end{array}$$

Fig.. 3.5 — Format ELL_R

Format DNS.

C'est la représentation dense des matrices creuses, à ne pas oublier...

Formats DIA ou CDS.

Soit par exemple

$$B = \begin{pmatrix} 4 & 1 & 2 & 0 & 0 & 7 \\ 0 & 5 & 0 & 8 & 0 & 0 \\ 1 & 0 & 7 & 5 & 1 & 0 \\ 0 & 2 & 0 & 9 & 1 & 0 \end{pmatrix}.$$

Ce format est dédié aux matrices dont les valeurs non nulles sont alignées sur des diagonales. La matrice sous format DIA est représentée par un vecteur debut de taille nbdiag et une matrice data de taille nbdiag $\times$ col. Dans debut, l'indice 0 correspond à la diagonale, 1 à la première sur-diagonale, -1 à la première sous-diagonale. Dans data se trouvent les diagonales. Plus précisément, la diagonale j telle que debut$[i] = j$ est la ligne i de data.

$$\begin{array}{r|l} \texttt{debut} & -2 \;\; 0 \;\; 1 \;\; 2 \;\; 5 \\ \texttt{data} & \begin{pmatrix} 1 & 2 & * & * \\ 4 & 5 & 7 & 9 \\ 1 & 0 & 5 & 1 \\ 2 & 1 & 8 & 0 \\ 7 & * & * & * \end{pmatrix} \end{array}$$

Fig.. 3.6 — Format DIA

Format JDS.

Dans ce format, on considère la matrice sous forme ELL rangée colonne d'abord et on permute les lignes par longueur croissante. Une permutation perm enregistre ce ré-arrangement de sorte que la ligne j dans la forme JDS correspond à la ligne perm[j] dans A. Un vecteur debut enregistre les indices de début de colonne.

idcol	1 2 3 0 4 6 2 6
debut	0 3 6 8
data	1 2 3 1 8 4 5 1
perm	0 3 5

Fig.. 3.7 — Format JDS

Format BCSR.

Ce format est similaire à CSR, par blocs. Soit $r \times c$ la taille de chaque bloc. Le vecteur debut contient $\texttt{lig}/r + 1$ éléments désignant le début des blocs dans idcol et dans data. Si, pour la ligne i, $\texttt{debut}[i] \leqslant k < \texttt{debut}[i+1]$, alors $\texttt{data}[k]$ est le premier élément du bloc dont l'élément $(0,0)$ est $\mathrm{A}[i, \texttt{idcol}[k]]$. Si jamais r ne divise pas lig, alors on peut rajouter un triplé pour les blocs de taille $r' \times c$ où $r' \cong \texttt{lig} \bmod r$.

Format VBR.

On peut aussi imaginer des blocs de taille variable. C'est le format VBR (*v*ariable *b*lock *r*ow). Il a été introduit dans la bibliothèque SPARSKIT. Il consiste à partitionner la matrice A en sous matrices dont certaines sont nulles. On commence avec deux tableaux bcol et browstart qui décrivent, comme dans CSR, les positions des blocs non nuls. Sur la i^e ligne de blocs, pour k entre $\texttt{browstart}[i]$ et $\texttt{browstart}[i+1]$, on trouve un bloc sur la $\texttt{bcol}[k]^e$ colonne de bloc. C'est un codage CSR pour les blocs non nuls. Ensuite, un tableau blockid donne les pointeurs vers le début de chaque bloc dans A. Un tableau rowstart pointe du début de chaque bloc de lignes dans la matrice A, tandis que colstart code les indices des premières colonnes. Découpons la matrice A comme suit et indiquons son stockage (figure 3.8).

$$\mathrm{A} = \left(\begin{array}{c|ccc|c|c} 0 & 1 & 2 & 3 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 8 & 0 & 4 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 5 & 0 & 0 & 0 & 1 \end{array}\right)$$

browstart	0 1 4 4 6
bcol	1 0 2 4 1 4
rowstart	0 1 2 3 4
colstart	0 1 4 5 6 7
blockid	0 3 4 5 6 7 8
data	1 2 3; 1; 8; 4; 5; 1;

Fig.. 3.8 — Format BCSR

Remarque. Il est difficile de trouver un découpage optimal dans le sens où les blocs sont les plus denses et les plus gros possibles ou dans le sens où SpMV sera optimale ([VM05])

Autres formats.

Il existe de nombreux autres autres formats ; nous en citons quelques uns :

— RSB, Recursive Sparse Block, *cf.* [MFPT10] ;
— SKS, Skyline Storage pour les matrices du même nom ;
— HiSM, Hierarchical Sparse Matrix (*cf.* [SVC03]) ;
— BBSM, Block Based Compressed Storage (*cf.* [VCS00]) ;
— découpage en sous-matrices super-creuses ;
— ...

Devant cette multitude de formats diversement utilisés, implantés et courants, nous avons choisis les plus simples et les plus courants (COO, CSR, ELL, DIA), les autres pouvant génériquement être rajoutés.

3.2 Implantation de base.

Nous allons montrer comment, initialement, une opération SpMV sur un anneau $\mathbb{Z}/q\mathbb{Z}$ peut être implantée.

3.2.1 Représentation des corps finis.

Comme dans la librairie FFLAS, nous avons le choix d'une représentation centrée ou à droite ainsi que du type de données (int32_t, float, double...). Nous devons aussi minimiser le nombre de réductions modulaires très coûteuses. Par exemple, nous préférerons, chaque fois que c'est possible la

```
for (i=0 ; i<n ; ++i){
    y += a[i] * b[i] ;
    y = fmod(y, m);
}
```

```
for (i=0 ; i<n ; ++i){
    y += a[i] * b[i] ;
}
y = fmod(y, m);
```

Fig.. 3.9 — Retarder fmod

boucle de droite à celle de gauche dans la figure 3.9. Cependant, les matrices creuses ont en général peu d'éléments non nuls par ligne et donc le nombre d'appels à fmod sera souvent égal à la taille de **y** dans une représentation standard. Ce ne sera cependant pas le cas pour les formats qui ne permettent pas de parcourir les lignes en une seule traite. Soit en effet M_q le nombre maximal d'accumulations avant réductions (pour un type de donnée fixé). On effectue alors $\lceil \frac{cn}{M_q} \rceil$ réductions par ligne ; on remarque que cet entier est égal à 1 dès que M_q est assez grand et cn petit — c'est par exemple sur des double et $p = 65537$, pour n'importe quelle valeur de cn raisonnable.

L'art d'une opération SpMV rapide réside donc en partie dans un bon ajustement de ces paramètres en plus d'une représentation efficace. Dans la figure 3.10, nous illustrons notre propos en comparant les performances du format ELL_R pour différentes tailles d'anneau selon que le type de données est des flottant float ou double. Nous utilisons seulement un cœur de Palo-Alto@ujfet son GPU. Les temps donnés sont la moyenne de 50 opérations SpMV, où **x** et **y** sont générés aléatoirement sur le CPU et les temps de transfert entre CPU/GPU (*host*/*device*) sont tenus en compte. Les mesures sont en millions d'opérations par seconde et une opération SpMV requiert $2 \times$ nbnz opérations.

Les matrices [2] utilisées sont présentées dans le tableau 3.1. Les vignettes sont créées avec l'exécutable smf2png disponible sur http://ljk.imag.fr/membres/Brice.Boyer/progs/smf2png.tar.gz et avec la suite libre imagemagick pour exagérer les densités en redressant l'histogramme (convert $i -equalize $i). La densité de rouge représente la densité des ± 1 tandis que le bleu représente la densité des autres entrées non nulles.

2 — matrices disponibles sur http://www-ljk.imag.fr/membres/Jean-Guillaume.Dumas/simc.html

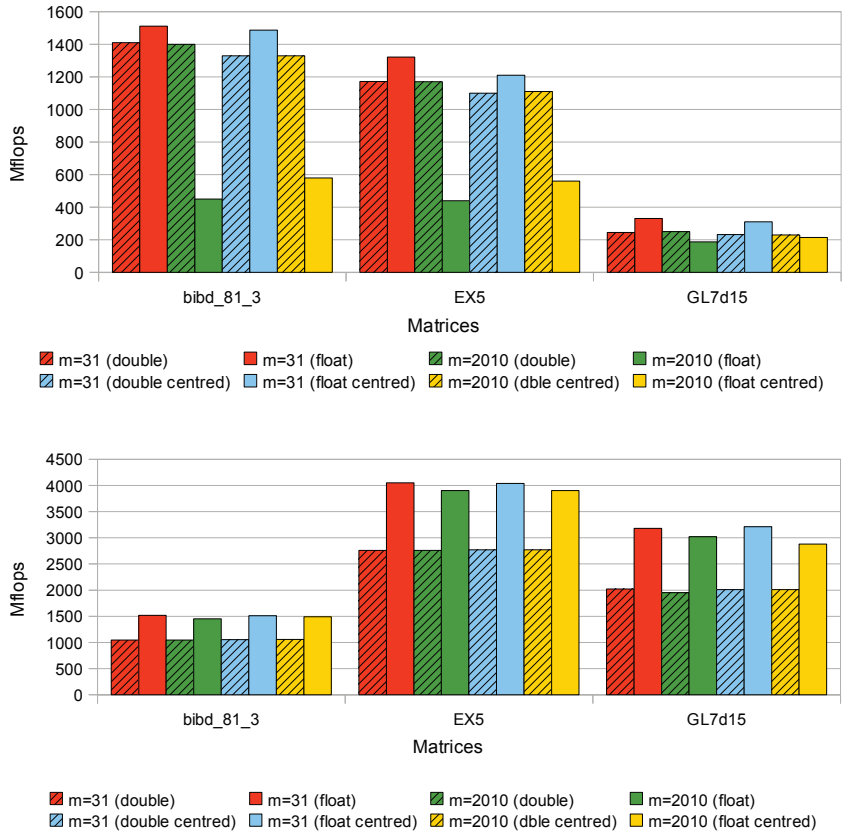

Fig.. 3.10 — Comparaison des performances entre `float` et `double` pour différentes tailles de $\mathbf{Z}/m\mathbf{Z}$, sur un CPU (en haut) et un GPU (en bas). Format ELL_R.

Nous constatons une chute de performance significative pour les opérations sur les flottants `double` sur ce GPU. Dans le cas tangent (gros nombre premier, petit type de données), le GPU ne tire pas autant à profit du format de données plus grand (`double`) autant que le CPU. Nous soupçonnons en outre que la parallélisation massive sur le GPU cache le coût de l'opération `fmod`.

Note : Sauf mention contraire, dans le reste de ce chapitre, les tableaux seront créés avec des `float`, $q = 31$ et la représentation classique.

3.2.2 Adapter les bibliothèques numériques.

Une approche type FFLAS consiste à utiliser des routines numériques classiques. Par exemple, nous proposons le pseudo-code 3.1 suivant pour illustrer comment génériquement une opération SpMV peut être effectuée sur un anneau $\mathbf{Z}/m\mathbf{Z}$ en utilisant des routines numériques.

```
spmv(y, A, x){
        foreach submatrix Ai in A do {
                spmv_num(y, Ai, x);     //guaranti sans overflow
                reduce(y, m);
        }
}
```

Code 3.1 — Utilisation de routines numériques.

La difficulté provient de créer ou d'extraire facilement une sous-matrice d'une matrice creuse, problème qui n'existe pas dans le cas dense. Supposons que l'on puisse effectuer b accumulations sur $\mathbf{y}[i]$ avant réduction. Supposons aussi que la ligne i de A contienne r_i éléments non nuls. Alors on doit découper cette ligne entre $\left\lceil \frac{r_i}{b} \right\rceil$ matrices. On peut améliorer cette technique avec une majoration plus fine, car les coefficients de la matrice sont connus au moment du découpage en sous-matrices ; la borne peut donc dépendre de ces coefficients. Découpons la ligne i en une union disjointe de κ_i ensembles $S_{i,k}$. Soit μ le plus grand élément représentable en valeur absolue (en général q ou $\lceil q/2 \rceil$). Pour tout i, k, on demande que $\sum_{\alpha \in S_{i,k}} |\alpha|\, \mu < M$ et créons $\max(\kappa_i)$ sous-matrices. Finalement, nous pouvons utiliser les bibliothèques numériques sur ces sous-matrices. Sur le CPU, des routines numériques ont été implémentées pour chaque format tandis que sur le GPU, nous avons adapté les routines de [BG09].

3.2.3 Utilisation d'OpenMP.

La simplicité de l'interface d'OpenMP (simples annotations pré-processeur `#pragma omp`) nous a permis de paralléliser rapidement notre code sur CPU. Cette technique a donné de bonnes performances comme montré dans la figure 3.11. Comme on peut le constater, OpenMP nous permet un gain de performances d'un facteur presque N (où N est le nombre de cœurs disponibles).

3.3 Nouveaux formats.

À part `COO`, tous les formats implémentés présentent un parallélisme au niveau des lignes. Le cas `COO` n'est pas évident à paralléliser et est généralement beaucoup plus lent. Les performances en mode parallèle des autres formats dépendent de la longueur des lignes et de la régularité des données. Par exemple, des lignes de tailles disparates sur un GPU laissent de nombreux fils inactifs (*idle*), ce qu'il faut éviter...

Plusieurs solutions existent : l'approche vectorielle de Bell *et al.* consiste à découper les lignes en morceaux plus courts et les réduire (*gather/scatter*) ou le réarrangement des lignes par poids décroissants avec des permutations de lignes (dans ce cas, une distribution en puissance par exemple ne marchera pas).

Lorsque les lignes sont de tailles sensiblement égales, le format `ELL` répond très bien au problème du parallélisme. Le cas des lignes de poids proche de c se retrouve très fréquemment en pratique. On

Tab. 3.1 — Profil des matrices creuses

name	lig	col	nbnz	rang	profil
mat1916	1916	1916	195985	1916	
bibd_81_3	3240	85320	255960	3240	
EX5	6545	6545	295680	4740	
GL7d15	460261	171375	6080381	13204	
mpolyout2	2410560	2086560	15707520	1352011	

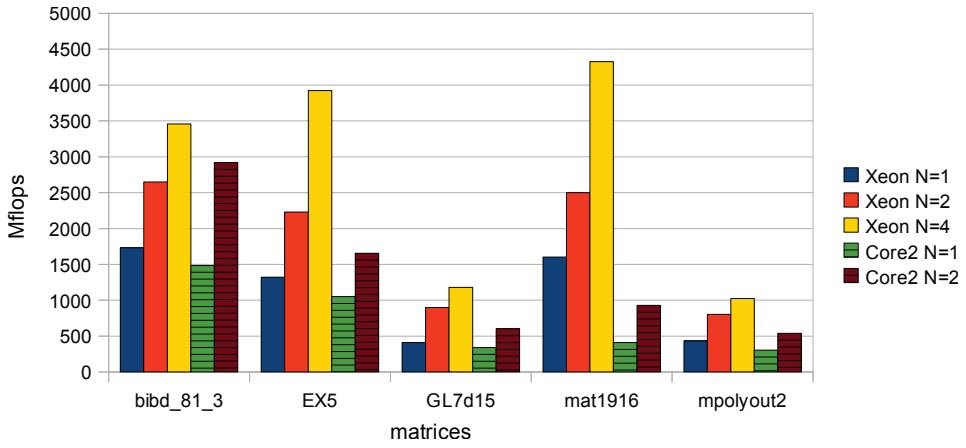

FIG.. 3.11 — Format CSR parallélisé par OpenMP avec N cœurs, sur Palo-Alto@ujf et sur Joran@imag

peut aussi couramment modéliser des matrices creuses par une distribution de poids de type $c + r_i$ avec c constant et r_i très variable. Alors, comme introduit dans par exemple [BG09], on peut partager la matrice A en une somme de matrices, avec une partie dense (correspondant à c) sous format ELL, et l'autre (r_i) sous format COO. Ils nomment HYB le format ainsi créé.

Enfin, un autre parallélisme correspond au découpage de la matrice A en sous-matrices par blocs de lignes et de les traiter en parallèle.

D'autre part, nous devons garder à l'esprit que cette bibliothèque est pensée pour une utilisation de type boîte noire et donc il y a un équilibre à trouver entre le temps passé sur l'optimisation de la matrice et celui passé dans SpMV — *cf.* la librairie introspective Oski. En particulier, d'autres pré-traitements incluent un ré-ordonnancement des lignes/colonnes pour créer des parties plus denses, extraire des parties très structurées, choisir le format le plus rapide, découper la matrice en sous-matrices plus efficaces ([VM05, TS06])... Par exemple, dans Oski, si le nombre de SpMV attendu est très grand, une meilleure optimisation de la matrice sera effectuée ; en cours de route, un retour sur les statistiques des précédents appels permet aussi, s'il reste du chemin, de ré-optimiser la matrice.

3.3.1 Premier essai : compilation à la volée (*JIT*).

Une approche tout à fait différente pour améliorer SpMV sur une matrice donnée a été de coder cette opération dans une bibliothèque dynamique. Nous lisons le fichier de la matrice et créons un `matname.cpp` qui applique cette matrice sur un vecteur. Par exemple, l'opération $\mathbf{y} \leftarrow \mathbf{y} + \mathrm{A}\,\mathbf{x}$ sur la matrice $\left(\begin{smallmatrix} 2 & 1 \\ 0 & 3 \end{smallmatrix}\right)$ deviendrait (si $q = 27$) le code 3.2.

```
// matname.cpp :
extern "C" {
void spmv(float * y, const float * x) {
   y[0] += 2*x[0] ;
   y[0] += x[1] ;
   y[0] = fmod(y[0],27);
   y[1] += 3*x[1] ;
   y[1] = fmod(y[1],27);
}
} // extern C
```

Code 3.2 — Compilation d'une matrice creuse

```
user$ g++ -shared -fPIC -o matname.so matname.cpp
```

Code 3.3 — Compilation de matrice en librairie dynamique.

Alors, on compile ce fichier source en une librairie dynamique (code 3.3). Et finalement nous utilisons `dlopen` pour accéder à ses fonctions (code 3.4).Nous avons aussi implémenté cette technique sur GPU, nous utilisons alors le code 3.5.

```
void * spmv_jit_lib ;
//...
spmv_jit_lib = dlopen(lib.c\_str(), RTLD\_LAZY);
```

Code 3.4 — Utilisation de la matrice dans la librairie dynamique.

```
user$ nvcc --shared -Xcompiler "-fPIC -W" -O0 -lcudart -o matname.so -shared
    matname.cu
```

Code 3.5 — Compilation d'une matrice sur GPU.

Comme on peut le voir dans cet exemple, on peut implémenter diverses optimisations directement dans le `.cpp`. Notamment, on peut replacer les ± 1 par des additions/soustractions.

Cependant, les grosses matrices prennent beaucoup de temps à compiler, même avec des options de compilations `-Ox` très basses, ou même si l'on découpe la matrice en petites matrices (plus simples à compiler). C'est avec cette dernière méthode seulement que l'on peut compiler `bibd_81_3`, mais cela prend 63s sur un Intel Xeon. Une fois compilée, la version CPU ne tourne qu'à 620 Mflops, ce qui est raisonnablement rapide mais pas utilisable ni compétitif.

Cette idée n'avait pas pris en compte la pression sur le cache d'instructions et une bande passante du CPU possiblement très limitée. Cependant, cela a produit l'idée suivante pour de nouveaux formats de données.

3.3.2 Prise en compte des ± 1.

L'observation précédente sur l'optimisation des opérations lors du JIT et la remarque du fait que beaucoup de matrices entières qui proviennent de diverses applications ont un grand nombre de $\pm 1_F$ a attiré notre attention sur ce cas particulier. En outre, en moyenne $2/(q-1)$ éléments sont des $\pm 1_F$ pour une distribution aléatoire des entrées non nulles. Sur un petit corps, cette quantité est non négligeable.

Nous extrayons donc deux matrices correspondant aux -1 et $+1$ et remplaçons les multiplications par des soustractions et additions, dans nos représentations bien moins coûteuses en espace. En effet, le champ `data` de nombreux formats (sauf `ELL`, `DIA`) peuvent être omis car on sait qu'ils correspondent uniquement à la valeur -1 ou $+1$. Finalement, faire des additions/soustractions au lieu des `axpy` permet de retarder significativement plus loin des réductions.

La figure 3.12 montre un gain significatif de performances lorsque l'on singularise les ± 1 (notés dans la légende '± 1' par opposition aux routines et formats standards, notés 'mul'). Précisons un instant les remarques visuelles que l'on peut faire sur la structure de ces matrices d'après le tableau 3.1. La matrice `GL7d15` compte près de moitié/moitié de 1 et de -1, les matrices `bibd_81_3` et `EX5` n'ont que des 1 et 55% des coefficients non nuls de `mat1916` sont des 1.

Il existe cependant deux problèmes majeurs liés à la ségrégation des ± 1. D'une part, à notre connaissance, il n'existe pas de librairie spécialisée dans ce domaine. Nous avons modifié spécialement les routines numériques implantées sur CPU et GPU pour s'adapter à ces nouveaux formats pour les ± 1. D'autre part, une structure qui était par ailleurs efficace avant la discrimination peut être alors cassée...

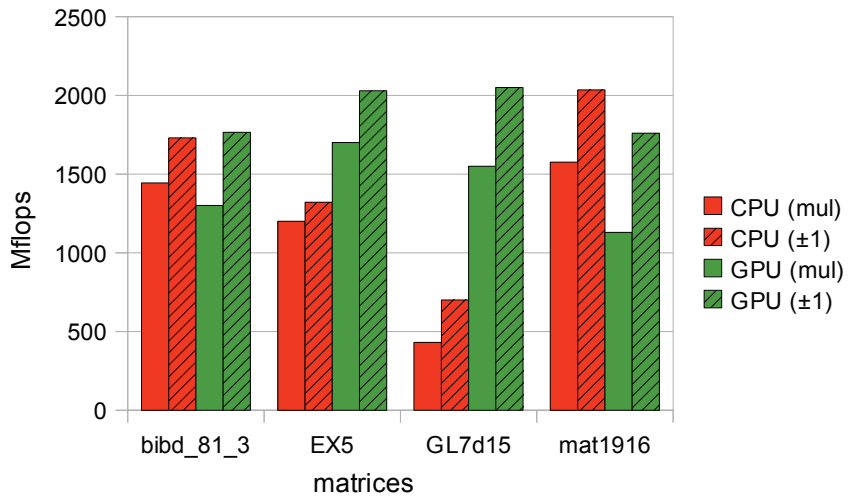

Fig.. 3.12 — Amélioration des performances sur Palo-Alto@ujfselon que l'on sépare ou non les ±1 ; format CSR

Cette dernière éventualité n'est cependant pas un argument qui doit nous réfréner dans la séparation des ±1 de la matrice initiale.

3.3.3 Retour sur les formats de base.

Comme évoqué précédemment, la matrice A peut être découpée en sous-matrices qui auraient un format spécial et qui pourraient être traitées différemment. Par exemple, on peut les découper par lignes et les distribuer pour du parallélisme en blocs de lignes et/ou des découper selon les colonnes dans le cas de la réduction retardée (code 3.1). Cela crée (possiblement) beaucoup de matrices creuses à optimiser individuellement.

Tout d'abord, le format COO est lent à cause des trop nombreux appels à fmod. Seulement dans les cas extrêmement creux est-il utilisable. Le format CSR est plus dense et peut permettre la réduction retardée, mais il faut s'assurer que les lignes sont bien équilibrées avant de paralléliser (surtout sur GPU). Les formats ELL sont très efficaces sur les matrices qui ont un poids de lignes à peu près constant. Une différence d'architecture CPU/GPU rend ELL plus performant « colonne d'abord » sur le GPU (meilleur *coalescing*) et « ligne d'abord » sur le CPU (meilleure utilisation du cache). La figure 3.13 suivante confirme, sur bibd_81_3, cette observation. Les chiffres sont normalisés pour que CSR soit 1 sur le CPU ou le GPU.

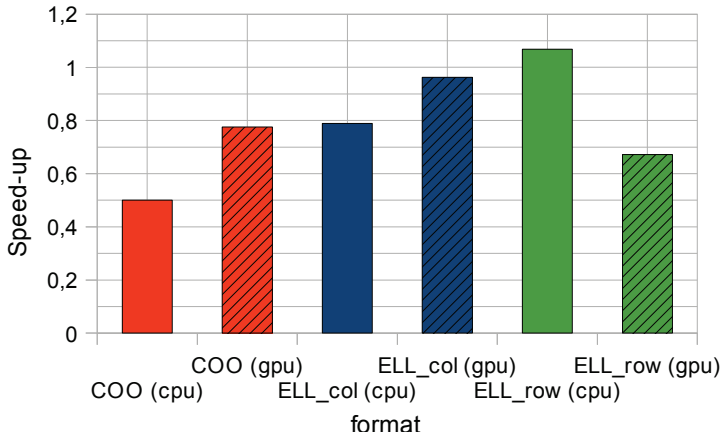

Fig.. 3.13 — Comparaison des performances des divers formats de stockage pour bibd_81_3 sur Palo-Alto@ujfentre CPU et GPU ; la référence est CSR sur chaque architecture

3.3.4 Formats hybrides.

Les points forts des différents formats de stockage nous entraînent à les combiner entre eux pour tirer avantage de chacun d'entre eux. Plus précisément, nous divisons notre matrice en deux sous-matrices, ou plus, pour chacune desquelles nous choisissons un format adapté. Des formats hybrides tels `ELL(_R)+COO` ou `ELL(_R)+CRS` donnent alors de bonnes performances, comme nous le montrons dans la sous-section suivante.

Quand le format `ELL` est extrait d'une matrice dont les lignes sont de poids semblables, beaucoup de lignes peuvent être laissées vides. Pour éviter les répétitions correspondant aux lignes vides dans le champ `debut` du format `CSR`, nous créons alors un nouveau format, nommé `COO_S`. Pratiquement, c'est un format basé sur `CSR` mais avec des pointeurs seulement sur les lignes non vides ; cela permet une bonne compression des informations de lignes, comparé aux formats `COO` ou `CSR` tout en gardant une structure `CSR` sur les lignes non vides. Le format `COO_S` possède des champs `data`, `idcol` comme `CSR` et `COO`. Le nombre `idlig`[k] correspond à la k^e ligne non vide qui commence dans `data` et `idcol` à `debut`[k]. Nous notons que ce format pourrait être omis si nous ordonnions préalablement les lignes par poids croissants par exemple. Nous donnons en exemple la figure 3.14, en reprenant la matrice A de la section 3.1.

idlig	0	1	3					
debut	0	3	6	8				
idcol	1	2	3	0	4	6	2	6
data	1	2	3	1	8	4	5	1

Fig.. 3.14 — Format `COO_S`

3.3.5 Algorithme heuristique pour le choix des formats.

Toutes les remarques précédentes montrent une grande complexité et une diversité dans les manières de découper/ordonner les matrices. Nous avons implémenté un algorithme heuristique qui permet d'aider à ce choix. L'utilisateur peut aussi forcer/aider l'algorithme dans ses choix. L'algorithme essaie de trouver pour chaque sous matrice (± 1, générale) un format efficace. L'efficacité d'un format est choisi à la compilation, selon l'architecture.

L'hybridisation est faite comme suit. Si la matrice est petite, nous choisissons `CSR`. Sinon, si les lignes sont à peu près de même taille, on choisit `ELL` ou `ELL_R`, selon la régularité. Le reste de la matrice est mis sous le format `CSR`, `COO` ou `COO_S` selon le nombre de lignes vides et le nombre d'éléments non nuls restants. Les paramètres qui décident du moment où découper les 1 et/ou les -1 sont choisis selon les architectures, de même que les meilleures tailles pour les matrices `ELL`, `ELL_R`. Ces réglages (*tuning*) sont définis manuellement à la compilation de la bibliothèque.

Les résultats donnés dans la figure 3.15 montrent que cette heuristique donne souvent des performances au moins égales sinon meilleures que les formats simples sur CPU et GPU.

3.4 Version itératives et par blocs de vecteurs.

Nous avons jusqu'à présent introduit de nouveaux formats en tirant parti des propriétés des matrices. Notre cadre est certes celui d'une opération SpMV performante, mais aussi d'une opération SpMV utilisable dans nos algorithmes « boîte noire ». Nous devons donc être efficaces sur un grand nombre d'itérations de SpMV (sur une même matrice creuse donc, par exemple la création d'un espace de Krylov) mais aussi sur des blocs de vecteurs (algorithmes par blocs). C'est ce que nous nous attachons à réaliser maintenant.

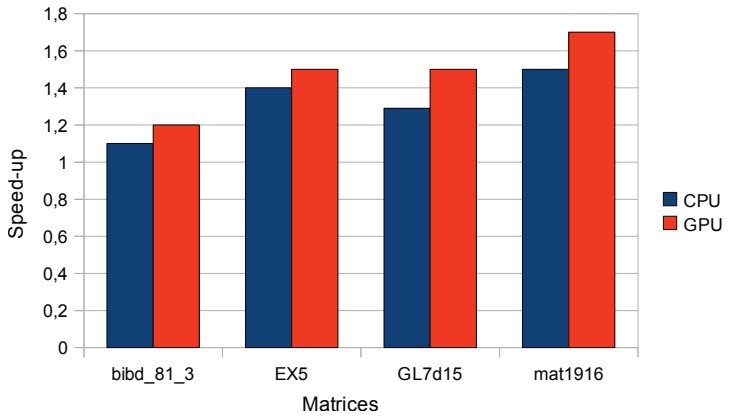

Fig.. 3.15 — Accélération du format auto-généré par rapport à CSR sur CPU et GPU.

3.4.1 Utiliser les multi-vecteurs.

Nous avons décrit l'opération SpMV pour $\mathbf{y} \leftarrow A\,\mathbf{x}$ où $\mathbf{x}$ et $\mathbf{y}$ sont des vecteurs. Nous avons aussi besoin du cas où $\mathbf{x}$ et $\mathbf{y}$ sont des multi-vecteurs, c'est-à-dire des matrices denses avec très peu de colonnes. C'est nécessaire en particulier dans les algorithmes par blocs. Il existe au moins deux manières de les représenter : ligne ou colonne d'abord. Dans le cas ligne d'abord, nous pouvons utiliser SpMV plusieurs fois (et aligner les vecteurs). Dans le cas colonne d'abord, nous devons écrire [3] des versions dédiées et essayer de faire bon usage du cache/de la réutilisation des données. En effet, dans ce cas, on ne traverse A qu'une seule fois et $\mathbf{x}$, $\mathbf{y}$ sont lus/écrits de manière contiguë. Sur la figure 3.16, nous notons que

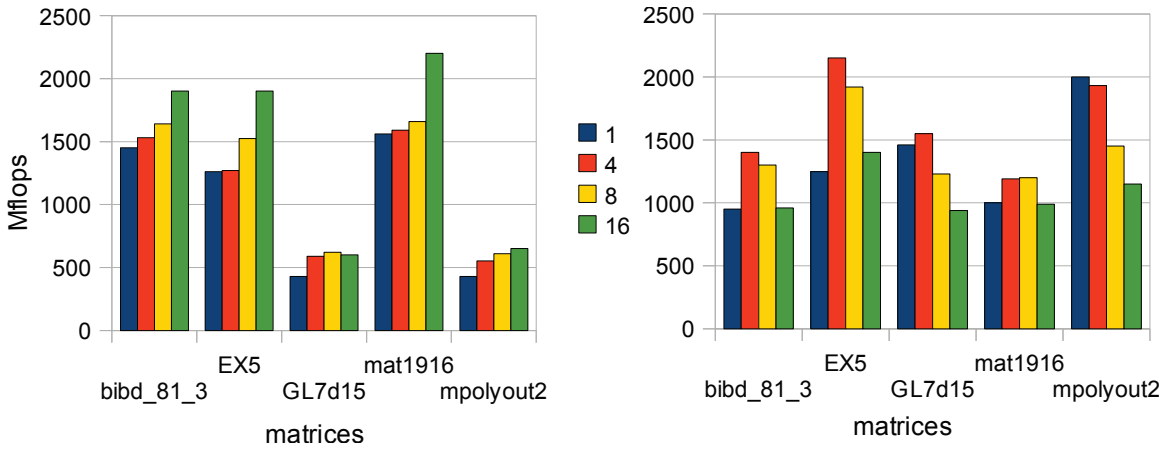

Fig.. 3.16 — Accélération de SpMV sur Palo-Alto@ujfentre CPU (gauche) et GPU (droite) pour les multivecteurs colonne d'abord, avec 1, 4, 8 et 16 vecteurs ; format ELL_R

l'utilisation sur le CPU des vecteurs colonne d'abord représentent un gain non négligeable de vitesse. Au contraire, sur le GPU, l'implémentation ne permet pas des tailles de bloc supérieures à 8. Nous soupçonnons que le problème ne vienne d'une mauvaise utilisation de la mémoire locale. De plus, les très grosses matrices commencent à atteindre les limites de mémoire embarquées sur le GPU utilisé.

3.4.2 Performance et problèmes.

Tout d'abord, nous faisons quelques remarques générales sur l'optimisation d'un appel à SpMV sur GPU. Le kit Cuda ne permettait pas alors des transferts asynchrones transparents entre *host* et *device* qui auraient permis de masquer quelques temps de transferts, notamment lors de l'envoi sur GPU des diverses sous-matrices et lors de la réception des vecteurs. Par contre, nous avons eu la possibilité de

3 — ou profiter des opérations SpMM (multiplication matrice creuse–matrice dense) que certaines librairies implantent...

mettre en général le vecteur **x** dans une mémoire texturée de façon à cacher les accès non réguliers. Nous avons aussi dû faire très attention à la pression sur la mémoire locale (ne pas trop la remplir, les tailles physiques n'étant pas connues alors) et à la pression sur le nombre de *threads* (Cuda préfère être saturé, mais il s'avère que les grilles ou blocs de *threads* ne doivent pas être de n'importe quelle taille ; cela dépend de la carte et doit être défini par tâtonnements, toutes les caractéristiques n'étant pas connues).

L'optimisation sur GPU d'un seul appel à SpMV du point de vue du *host* est surtout très lent du fait du transfert des vecteurs/matrices de/vers la carte (*device*). Dans la figure 3.17, nous illustrons ce problème par le calcul de $y \leftarrow \mathrm{A}^n x$ ou de la suite $\left\{\mathrm{A}^i x\right\}_{i\in[\![0,m]\!]}$ qui sont utilisés dans les techniques boîte noire. Les deux pseudo-codes utilisés sont présentés dans le code 3.6.

```
void smpv_n(y, A, x, n){
        y_d = copy_on_gpu(y);
        x_d = copy_on_gpu(x);
        A_d = copy_on_gpu(A);
        for (i=0 ; i<n ;++i) {
                y_d = A_d * x_d ; // spmv sur GPU
                x_d = y_d;        // copie complète
        }
}

void n_spmv(y, A, x, n){
        A_d = copy_on_gpu(A);
        for (i=0 ; i<n ;++i) {
                y_d = copy_on_gpu(y_i);
                x_d = copy_on_gpu(x_i);
                y_d = A_d * x_d ; // spmv sur le GPU
        }
}
```

Code 3.6 — Pseudo-code pour $y \leftarrow \mathrm{A}^n x$ et n répétitions de $y \leftarrow \mathrm{A}x$ sur le GPU avec x généré aléatoirement sur CPU.

Comme attendu, la figure 3.17 montre clairement qu'il faut limiter les transferts CPU/GPU. Même avec un petit nombre d'itérations, un gain est déjà décelable. *Note :* Les graphiques de ce chapitre in-

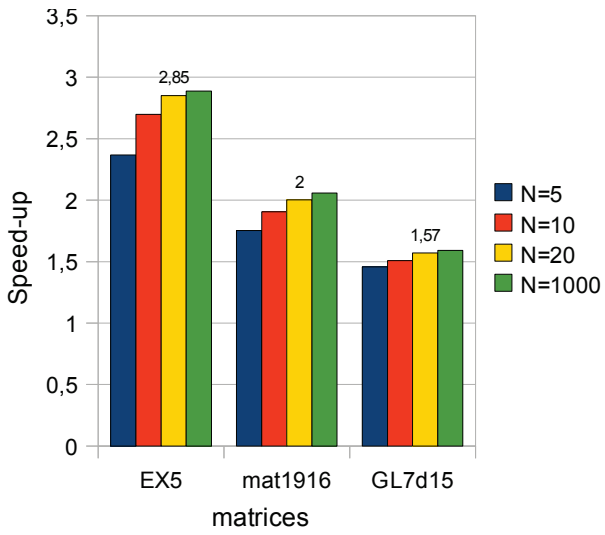

Fig.. 3.17 — Accélération sur GPU (Palo-Alto@ujf) de $y \leftarrow \mathrm{A}^n x$ comparée à n fois $y \leftarrow \mathrm{A}x$, avec $n = 5, 10, 20$; format CSR.

cluent tous les temps de transferts de/vers le GPU pour chaque opération SpMV !

3.5 Prolongements.

Nous avons donc implanté un produit matrice creuse–vecteur dense rapide sur architectures multi-cœurs et GPU. Nous avons été limités par les possibilités de Cuda (qui à l'époque n'était qu'un très fine sur-couche du C++). Un autre inconvénient fut l'impossibilité d'utiliser OpenMP sur le *host* pendant que Cuda faisait une partie des calculs. Il n'était aussi (logiciellement) pas possible de faire du multi-GPU. La gestion de la mémoire locale et des nombres de fils n'a pas été aisée et cela a dû être totalement effectué à la main (et de façon expérimentale). Si les dernières versions de Cuda ne présentent plus ces défauts (notamment une bien meilleure prise en charge des mécanismes de *template*, des noyaux, une gestion automatique de la mémoire locale...) il serait de toute évidence une perte de temps de mettre à jour le code en Cuda-4.1. Entre temps cependant, une librairie cuSPARSE est apparue : en particulier, le produit SpMV y est disponible numériquement. Il serait donc intéressant de l'utiliser (surtout que notre *design* a été conçu pour envelopper au besoin des routines numériques externes). Il serait cependant beaucoup plus intéressant de voir cette bibliothèque utilisable facilement depuis une interface [4] de type OpenCL... Aussi, une version parallèle d'Oski (pOski) vient de sortir. Finalement, et ce n'est pas le travail le plus difficile à réaliser, nous n'avons pas encore utilisé de routines numériques optimisées pour remplacer notre code générique de SpMV numérique. L'idée est que la conception est bonne et donc que cela sera fait directement dans LinBox, qui commence à se voir lié des bibliothèques spécialisées dans les opérations creuses (superlu [5]).

Finalement, tout comme nous avons fait une analyse statistique pour détecter l'intérêt de séparer ou non les ± 1, nous pourrions envisager de prolonger cette étude à la détection de sous matrices plus denses ou plus creuses, avec une forte densité diagonale...

4 — Mentionnons que ce n'est pas une attente irréaliste : AMD propose déjà une implantation OpenCL de leur BLAS entre autres (AMD APPML).

5 — *cf.* http://crd.lbl.gov/~xiaoye/SuperLU/.

Chap. 4 Application au calcul du rang.

Sommaire

4.1 Parallélisation de SpMV sur CPU et GPU. 69
4.1.1 Parallélisation du calcul de la suite matricielle. 70
4.1.2 Parallélisation du calcul des σ-bases. 70
4.1.3 Calcul parallèle du co-degré du déterminant. 72
4.1.4 Performances de la version parallèle de l'algorithme de Wiedemann. . . . 73
4.2 Prolongements. 73

Parmi les applications les plus représentatives dans LinBox requérant une opération SpMV rapide, il y a les méthodes « boîte noire » basées sur les approches de type Lanczos, Krylov. En particulier, les méthodes proposées par Wiedemann [Wie86] et sa version par blocs proposée par Coppersmith [Cop94] peuvent parfaitement mettre en lumière l'efficacité d'une opération SpMV puisque cette dernière en est souvent une pierre d'achoppement. Nous présentons dans ce chapitre une parallélisation du code du rang avec ré-écriture minimale, en parallélisant diverses parties du calcul.

4.1 Parallélisation de SpMV sur CPU et GPU.

En application donc, nous proposons d'améliorer l'implantation du rang de Wiedemann par bloc présenté dans [DGEVU07]. Rappelons brièvement cet algorithme. Nous renvoyons vers [Tur06] pour plus de détails.

Soit $A \in \mathbf{F}^{n\times n}$ une matrice qui satisfait les conditions de pré-conditionnement de [KS91]. Alors l'algorithme peut être décomposé en trois étapes :

i. Calcul de la suite matricielle $S_i = Y^T A^i Y$ pour $i = 0, \ldots, 2n/s + O(1)$, avec $Y \in \mathbf{F}^{n\times s}$ choisi aléatoirement.

ii. Calcul du générateur minimal $F_Y^A \in \mathbf{F}^{s\times s}[x]$ de la série matricielle $S(x) = \sum_i S_i x^i$.

iii. Renvoyer le rang $r = \deg\left(\det\left(F_Y^A\right)\right) - \operatorname{codeg}\left(\det\left(F_Y^A\right)\right)$.

Notre approche consiste à séparer la parallélisation de chaque étape. La première (4.1:i) est clairement reliée à SpMV et nous réutiliserons nos outils présentés dans le chapitre précédent. Pour utiliser ffspmvgpu dans LinBox, une simple sur-couche (*wrapper*) — pour les vecteurs, les matrices, les corps et la fonction *read* — est nécessaire.

Le second est effectué par la parallélisation dans le calcul des σ-bases (cf [BDG10, section 3.2], [DGEVU07, section 2.2]). Finalement, la dernière étape se réduit au calcul du co-degré du déterminant de cette σ-base.

4.1.1 Parallélisation du calcul de la suite matricielle.

La parallélisation proposée dans [DGEVU07] correspondait à envoyer un ensemble indépendant de blocs de vecteurs V vers différents cœurs et de les appliquer en parallèle. Puis les résultats sont rassemblés pour calculer un produit scalaire dense avec le bloc $^{\top}$U. Une alternative est, bien entendu, l'usage de SpMV. Dans la figure 4.1 nous comparons les vitesses de l'implantation SpMV native de LinBox à celle introduite dans ffspmvgpu pour la création de la suite matricielle lors de l'étape 4.1:i.

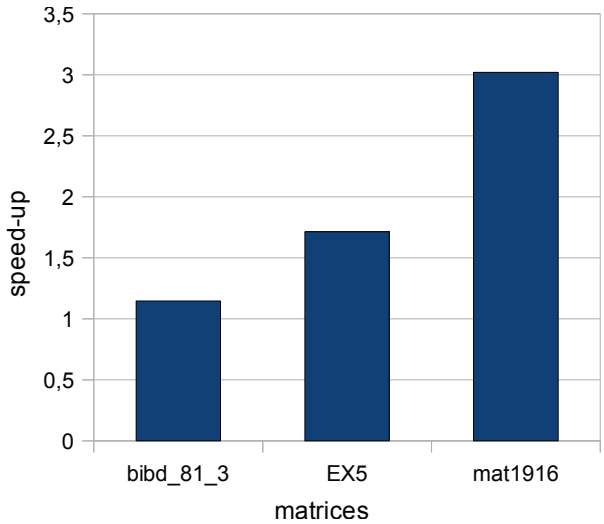

Fig.. 4.1 — Accélération due à ffspmvgpu comparée à l'implantation de LinBox dans la génération de la suite matricielle ($2n$ itérations) sur un cœur de Montpellier@lirm

Nous montrons maintenant une parallélisation du point 4.1:ii. Le choix de conception dans cette sous-section sera de réutiliser efficacement des composants rapides existants dans la bibliothèque, plutôt que d'en créer d'autres *ad hoc*.

4.1.2 Parallélisation du calcul des σ-bases.

Une σ-base peut être efficacement calculée en utilisant l'algorithme `PM-Basis` de [GJV03]. Cet algorithme se réduit essentiellement à la multiplication de matrices polynomiales. Une première approche peut donc consister à paralléliser cette multiplication.

Soient $\mathrm{A}, \mathrm{B} \in \mathbf{F}^{n\times n}[x]$ deux matrices polynomiales de degré d. On peut multiplier A et B dans $\mathbf{F}$ avec une complexité $\mathcal{O}\left(n^3 d + n^2 d \log d\right)$ — lorsque $\mathbf{F}$ a une racine primitive d^{e} de l'unité, *cf.* [CK91]. En supposant que l'on dispose de k processeurs tels que $k \leqslant n^2$, on peut effectuer cette multiplication avec une complexité parallèle de $\mathcal{O}\left(\frac{n^3 d}{k} + \frac{n^2 d \log d}{k}\right)$ opérations dans $\mathbf{F}$. En effet, soit à paralléliser l'algorithme 4.1 rapide et séquentiel de multiplication de matrices polynomiales suivant. Nous notons, dans cet algorithme `DFT`(P, L) l'évaluation multi-points du polynôme P en les points de L et $\otimes$ est le produit point-à-point (Kronecker).

Nous pouvons paralléliser cet algorithme 4.1 ainsi :

- Les étapes 1, 2 et 4 sont des transformées de Fourier rapides sur chaque entrée, c'est-à-dire $n^2 \times \mathcal{O}(d \log d)$ opérations (*cf.* [GG99, Théorème 8.15]). Sur chacun des k processeurs, on peut effectuer une FFT sur $\frac{n^2}{k} + \mathcal{O}(1)$ éléments matriciels. Ceci donne une complexité parallèle en $\mathcal{O}\left(\frac{n^2 d \log d}{k}\right)$ opérations dans $\mathbf{F}$.

Algorithme 4.1 : Multiplication rapide de matrices polynomiales

Entrées : $A, B \in \mathbf{F}^{n\times n}[x]$ de degré d
Entrées : ω une racine d^e primitive de l'unité dans $\mathbf{F}$
Sorties : $A \times B$

1 $\bar{A} := \mathtt{DFT}(A, [1, \omega, \omega^2, \cdots, \omega^{2d}])$
2 $\bar{B} := \mathtt{DFT}(B, [1, \omega, \omega^2, \cdots, \omega^{2d}])$
3 $\bar{C} := \bar{A} \otimes \bar{B}$
4 $C := \frac{1}{2d}\mathtt{DFT}(\bar{C}, [1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-2d}])$
5 **retourner** C

✦ L'étape 3 quant à elle correspond au calcul de $2d$ produits matriciels indépendants, de dimension n, ce qui donne une complexité de $\mathcal{O}\left(n^3 d\right)$ opérations dans $\mathbf{F}$. Il est facile de distribuer ces calculs sur k processeurs tel que chacun effectue $\mathcal{O}\left(\frac{n^3 d}{k}\right)$ opérations.

Nous montrons dans la figure 4.2 les performances de l'implantation de cet algorithme parallèle dans LinBox. Nous constatons dans la figure 4.2 que notre code n'atteint pas les performances paral-

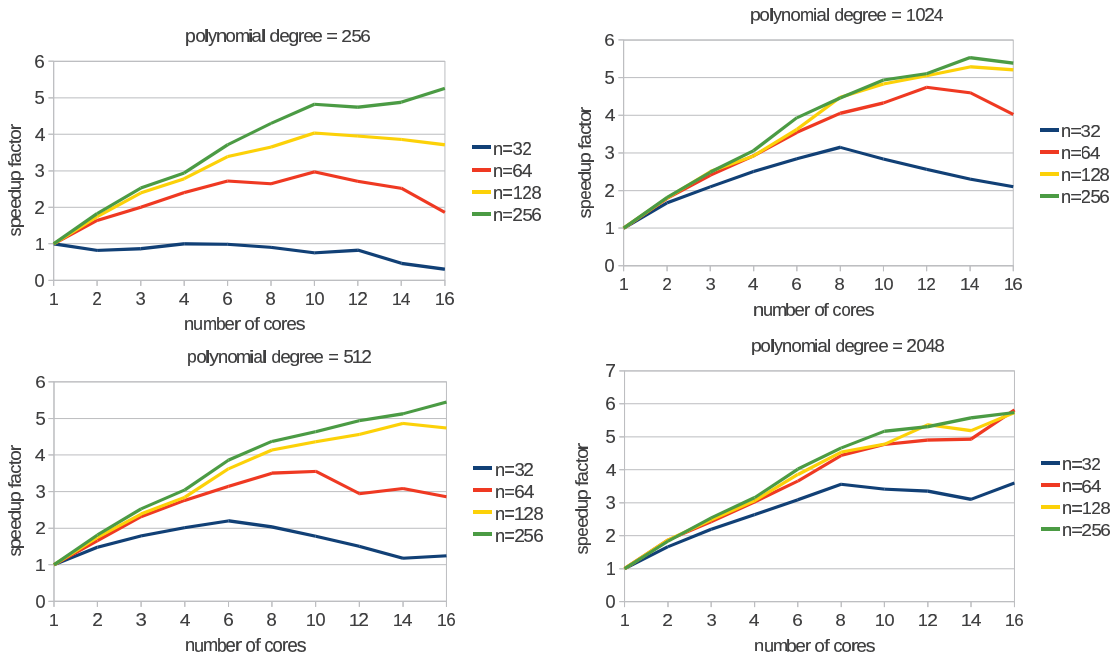

Fig.. 4.2 — Passage à l'échelle d'une multiplication matricielle polynomiale parallèle avec LinBox et OpenMP sur un 16 cœurs (Quad-Core AMD Opteron). La dimension des matrices est n.

lèles attendues. Au mieux, nous n'arrivons qu'à un gain de 5,5. Cependant, sur un sous-ensemble des processeurs disponibles, nous arrivons tout de même à 75% des performances optimales. Ce phénomène est peut-être dû à l'architecture (problèmes de cache selon la localité des calculs et des cœurs). Il serait très intéressant de paralléliser ce code avec kaapi et observer.

Comme attendu cependant, la figure 4.2 montre que la taille croissante des matrices permet une meilleure parallélisation. Probablement, les latences dues aux opérations sur la mémoire/les caches locaux et au traitement des fils par OpenMP sont cachées par le travail plus important sur les grosses matrices.

Implantation parallèle des σ-bases.

D'après la réduction de l'algorithme `PM-Basis` à la multiplication de matrices polynomiales, nous pouvons obtenir une complexité parallèle de $\mathcal{O}^{\sim}\left(\frac{n^3 d}{k} + \frac{n^2 d \log d}{k}\right)$ opérations dans $\mathbf{F}$ (où k est le nombre

de processeurs disponibles, tel que $k \leqslant n^2$. Ainsi, il suffit d'utiliser directement notre algorithme polynomial dans le code de σ-base orignal de LinBox pour en obtenir une implantation parallèle. Nous montrons les performances obtenues dans la figure 4.3. Encore une fois, nous n'avons pas atteint les

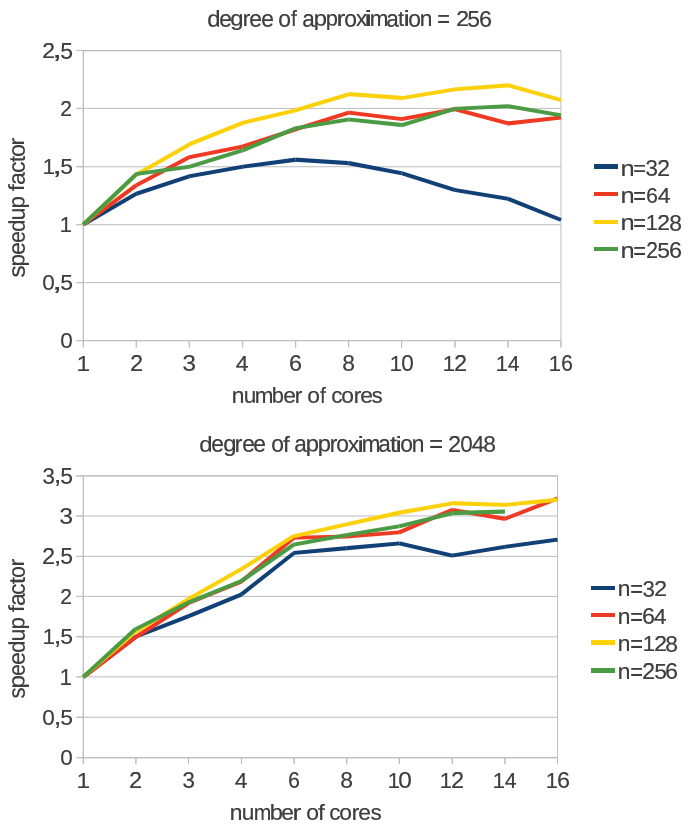

Fig.. 4.3 — Passage à l'échelle lors du calcul parallèle des σ-bases dans LinBox, avec OpenMP sur un 16 cœurs (Quad-Core AMD Opteron). La taille des matrices est n.

performances théoriques de passage à l'échelle parallèle mais nous avons tout de même obtenu une accélération de 3× sur 16 processeurs. Nous remarquons cependant que ces temps sont en accord avec les performances précédemment obtenues (5× dans la figure 4.2).

Une explication quant au légèrement moindre gain obtenu réside dans la structure récursive de l'algorithme `PM-Basis`. En effet, cet algorithme est de type « diviser pour régner » sur le degré de l'approximation (*cf.* [GJV03, théorème 2.4]) : les appels récursifs sont donc effectués sur des approximations de plus en plus petit degré ce qui entraîne une perte d'efficacité de la multiplication en parallèle. Un prolongement de ce travail consisterait à paralléliser aussi l'algorithme `M-Basis` ([GJV03]) qui, lorsque les tailles deviennent petites, est en pratique plus rapide.

4.1.3 Calcul parallèle du co-degré du déterminant.

L'étape 4.1:iii est aisée à paralléliser. Ce calcul se fait par « évaluation/interpolation ». Nous évaluons en parallèle en différents points le polynôme matriciel et calculons les déterminants des matrices obtenues en ces points. Dans notre implantation, nous parallélisons simplement les $s \times \deg\left(\mathrm{F}_{\mathrm{A}}^{\mathrm{Y}}\right) + 1$ points d'évaluation ; nous pourrions aussi paralléliser les calculs de déterminant lors de ces évaluations. L'interpolation s'effectue séquentiellement avec la classe `Poly1CRT` de Givaro. Notons que nous pourrions utiliser une parallélisation de cette étape aussi (*cf.* [DGR10]).

4.1.4 Performances de la version parallèle de l'algorithme de Wiedemann.

Dans le tableau 4.1, nous montrons les performances de notre algorithme sur un processeur 8 cœurs (Montpellier@lirm). La ligne `*-LB` contient les temps pour l'implantation actuelle dans LinBox, les lignes `*-SpMV` correspondent à la bibliothèque ffspmvgpu *wrappée* dans LinBox.

L'accélération pour SpMV entre 1 et 8 cœurs est sensiblement supérieure à 5 pour toute les matrices alors que l'accélération pour l'implantation de LinBox est entre 4 et 4,9. De plus, l'accélération obtenue grâce à SpMV par rapport à LinBox sur le point 4.1:i (suite matricielle) semble passer à l'échelle et s'améliore même dans la version parallèle. En comparant avec la figure 4.1, nous pouvons confirmer que la pierre d'achoppement dans la phase de génération de la suite est bien SpMV.

Tab. 4.1 — Calcul du rang modulo 65 521 avec OpenMP pour l'algorithme parallèle par bloc de Wiedemann sur Montpellier@lirm (temps en secondes)

Matrix	mat1916		bibd_81_3		EX5	
nb. de cœurs	1	8	1	8	1	8
Seq-LB	15,09	3,08	47,73	12,41	84,21	20,22
Seq-SpMV	5,02	0,91	41,28	7,56	49,66	7,36
σ-base	9,02	1,64	18,45	3,63	37,45	8,39
Interpolation	0,37	0,29	1,07	0,82	2,29	1,75
Total-LB	24,48	5,01	67,25	16,86	123,95	30,36
Total-SpMV	14,41	**2,84**	60,80	**12,01**	89,40	**17,50**

4.2 Prolongements.

Si les performances obtenues sont déjà tout à fait encourageantes, nous pouvons faire encore mieux. Ce dernier tableau 4.1 devra en effet être complétée par l'inclusion d'une ligne correspondant au (multi-)GPU [1]. Cela implique d'une part du travail dans LinBox et FFLAS-FFPACK (introduction d'OpenCL dans LinBox, utilisation des BLAS sur GPU, par exemple) et d'autre part du travail dans ffspmvgpu (utilisation de cuSPARSE ou équivalent en OpenCL, probablement devrons-nous en attendant porter notre code écrit en Cuda vers OpenCL).

Du point de vue de la conception, nous pouvons certainement tirer comme conclusion de cette partie qu'il est bénéfique sinon essentiel d'utiliser des standards ouverts (OpenMP, OpenCL) qui sont garants de performances portables et de stabilité plutôt que les solutions propriétaires synonymes de fermeture et de performances ponctuelles. D'autre part, notre conception modulaire nous permet d'obtenir une grande généricité : nous pouvons choisir d'utiliser nos routines aussi bien que des routines spécialisées fournies par d'autres logiciels, et s'assurer ainsi de bonnes performances, sans changer le code supérieur — i.e. au dessus de `apply` — mais simplement un paramètre *template*.

1 — Même si techniquement cela est possible et a été testé, l'utilisation du GPU pour calculer les produits SpMV dans LinBox a donné de très mauvaises performances, liées à des problèmes techniques dans Cuda. Comme le code n'a pas été porté dans les dernières versions de Cuda, nous ne pouvons pas dire si le problème persiste, mais comme l'utilisation des différents niveaux de mémoire a été revue et automatisée, nous pouvons penser que les problèmes ont été réglés.

III

Conception logicielle pour une bibliothèque de calcul.

"Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves."

Alan Kay

Introduction.

Sommaire

Quelques problèmes et défis pour une bibliothèque de calcul. 77
Le choix du langage C++ modèle la bibliothèque. 78
Configuration et installation simplifiées, préalable à une bonne conception. 80
Plan de la troisième partie. 82

De manière résumée, l'*algèbre linéaire* est l'étude des espaces vectoriels sur un *corps* donné. En particulier, on s'intéressera aux éléments de ces espaces, les *vecteurs* et à des *transformations linéaires* qui « respectent » la structure de ces espaces. En dimension finie, la représentation de ces transformations par des *matrices* dans des bases permet des applications informatiques pratiques.

Dans cette introduction, nous allons d'abord cerner et définir les problèmes que nous avons à considérer lors de la conception d'une bibliothèque générique — en particulier LinBox — puis nous introduirons quelques concepts liés au développement du code de cette bibliothèque. Nous remarquons que ces idées sont indépendantes de l'écosystème dans lequel vit LinBox et s'appliquent à toute bibliothèque de calcul qui doit définir des structures, fournir des algorithmes performants et être la plus générale possible. Nous renvoyons par exemple à [FWH04] qui examine la conception templétée de la bibliothèque CGAL où à la conception de MTL (Matrix Template Library) qui utilise les concepts de généricité de la STL et d'optimisations lors de la compilation, *cf.* [SL98]. Nous voulons dans ce chapitre non seulement expliquer des choix de conception pour LinBox mais aussi les rendre généraux et les abstraire de manière à les appliquer dans d'autres domaines.

Quelques problèmes et défis pour une bibliothèque de calcul.

Une bibliothèque d'algèbre linéaire doit donc définir proprement ses corps, vecteurs et matrices avant de proposer d'une part des opérations élémentaires sur ces objets (*cf.* les BLAS ou autres briques de base) et d'autre part des algorithmes plus sophistiqués. Les choix de représentation, de structuration représentent souvent d'importants défis et de véritables problèmes dont les solutions ne sont généralement pas triviales. La réponse apportée par LinBox à ces défis tient en un concept : la *généricité*.

Construire une bibliothèque générique est donc un de nos *challenges* principaux. Nous voulons ainsi être capable, sur un problème donné, de le résoudre génériquement ; par exemple, sur tout type de matrice et de corps nous devons être capable de calculer un rang. Il est fort probable que la méthode

Tab. 4.2 — Évolution du nombre de lignes de code (en milliers) dans LinBox, FFLAS-FFPACK et Givaro (†contient Givaro, ‡contient FFLAS-FFPACK).

LinBox	1.0.0†,‡	1.1.0†,‡	1.1.6‡	1.1.7‡	1.2.0	1.2.2	1.3.0
lignes de code	77,3	85,8	93,5	103,1	107,7	109	111,8
FFLAS-FFPACK	N/A	N/A	N/A	1.3.3	1.4.0	1.4.3	1.5.0
lignes de code	—	—	—	11,6	23,9	25,2	25,5
Givaro	N/A	N/A	3.2.16	3.3.3	3.4.3	3.5.0	3.6.0
lignes de code	—	—	30,8	33,5	39,4	48,3	48,6
total (milliers)	77,3	85,8	124	137	171	182	186

générique ne soit pas la plus efficace dans certaines conditions. Donc, une fois cerné un problème particulier, nous allons le spécialiser : c'est l'autre défi majeur de LinBox, l'*efficacité*.

Nous soutenons que la généricité et l'efficacité sont des concepts qui peuvent aller de pair, même dans une bibliothèque. Les bibliothèques de calcul que nous pouvons utiliser (gmp, BLAS, FLINT...) sont en général très spécialisées — et extrêmement efficaces. Cette idée se retrouve dans la conception initiale de LinBox mais aussi nous trouvons des références comme [GJK+05] qui la théorisent et s'en servent pour détecter des limites aux standards. Donner des réponses à la question combinée de la généricité et de l'efficacité est donc un objectif crucial de cette partie.

D'autre part, la conception, le développement et la maintenance d'un logiciel entraînent des problèmes généralement non triviaux. Dans cette partie, nous allons aussi discuter de solutions et de cadritiels (*framework*) apportés à divers problèmes techniques : comment optimiser et assurer de bonnes performances, comment rendre un code existant plus stable, plus fiable, plus facilement maintenable ?

Car si proposer un code générique et efficace constitue des défis premiers pour LinBox, il ne faut pas pour autant négliger la qualité de l'interface utilisateur et celle de l'écriture du code. En effet, cette bibliothèque ne doit pas se cantonner à une bibliothèque dédiée à la recherche mais aussi à un véritable outil pour implanter facilement et efficacement les algorithmes d'un nouvel utilisateur. La conception d'une bibliothèque dont la sophistication ne nuit pas à son utilisation ni à ses performances ou à son évolutibilité et à la réutilisation des concepts développés est un autre défi, probablement mésestimé, qui mérite néanmoins que l'on s'y attarde.

Nous constatons qu'au cours des années le nombre de lignes de code de LinBox n'a cessé d'augmenter, comme le montre [2] le tableau 4.2. Ce code a été contribué par de nombreuses personnes qui n'ont souvent été actives que ponctuellement. Malgré certaines règles et conventions de codage publiées [3], le code produit est très hétérogène, certaines parties non maintenues ou désuètes, de larges parties du code plus ou moins documentées...

Il existe une large bibliographie sur les questions de standard de codage, de conception logicielle ([Ale01, Gam95, SA05, Str94]), de nombreuses ressources sur Internet et surtout l'expérience acquise par de nombreux projets libres dont la structuration, les façons de procéder sont disponibles à tous et éprouvées. Nous allons tirer profit de ces enseignements et essayer d'apporter quelques pierres à l'édifice.

Le choix du langage C++ modèle la bibliothèque.

Nous présentons dans cette section une introduction sur les styles avec lesquels est écrite la bibliothèque LinBox : son style d'architecture, de fonctions, de gestion de la mémoire. Pour une approche très détaillée nous renvoyons à la thèse [Gio04].

2 — Réalisé avec sloccount, disponible à l'adresse `http://sourceforge.net/projects/sloccount/`.
3 — *cf.* `http://www.linalg.org/developer.html#standards`.

Une bibliothèque d'en-têtes.

La bibliothèque LinBox se présente comme une bibliothèque d'en-têtes contenant des fonctions et classes *template* (patrons) écrits en langage C++. Le choix de ce langage a été fait dès la création de LinBox pour sa généricité, son efficacité et sa prise en charge par de nombreux compilateurs. Souvent, en C, on écrit un code de la manière dont la machine le pense. Le langage C++, certes une « extension » du C, nous permet en outre abstraction et généricité.

Le code est structuré autour de structures de données (p.ex. dans les dossiers `field/`, `matrix/`, `vector/`, `blackbox/`, *etc.*), d'algorithmes (répertoire `algorithms/`) sur ces structures et finalement de solutions (`solutions/`). Si les deux premiers points sont essentiellement destinés aux développeurs, le troisième point fournit une interface avec les utilisateurs, et le choix des algorithmes via des *méthodes*.

```
template <class Blackbox, class Method >
inline unsigned long &rank ( unsigned long   &r
                           , const Blackbox  &A
                           , const Method    &M = Method::Hybrid()
                           );
```

Code 4.1 — Exemple de solution `rank.h`.

Cette solution, en fonction de la méthode donnée et de traits (caractéristiques) sur A, va choisir — de préférence à la compilation — le meilleur algorithme à sa disposition. C'est ce style d'interface qui est recherchée dans LinBox.

Styles de fonctions.

De manière préférentielle, les fonctions de LinBox retournent une référence sur leur type de retour. Cette référence est aussi le premier argument de la fonction. En outre, pour éviter les recopies [4] les arguments sont toujours appelés par référence — les références sont plus faciles à utiliser que des pointeurs, permettent un code plus sûr. Les codes 4.1 et 4.2 en sont des exemples.

```
Matrix& oneFunction (Matrix& B, const Matrix& A) ;

Matrix& someFunction (Matrix& A)
{
        ...
        Matrix B (A.field(), A.rowdim(), A.coldim() );
        oneFunction(B, A);
        ...
        return A ;
}
```

Code 4.2 — Style de programmation de LinBox.

En particulier, tous les arguments doivent être déclarés et initialisés avant l'appel d'une fonction ou la création d'une classe.

Modèle d'allocation.

Finalement, dans la conception de nos objets, nous nous imposons un modèle d'allocation particulier ([DGPS10]) qui est un analogue du style de programmation RAII (*Resource Acquisition Is Initialisation*), introduit par [Str94], dans le sens où la mémoire utilisée par les objets est allouée par le constructeur et libérée seulement à la destruction ; la gestion de la mémoire acquise par l'objet lui est exclusivement réservée. Dans le cas où la taille d'un objet ne peut pas être connue à l'avance (p.ex. po-

4 — Parfois cela ne suffit pas : on évitera l'utilisation des `std::pair` qui recopient leurs arguments et dupliquent alors la mémoire utilisée.

lynômes de matrices, noyau), une fonction `resize` permet de réallouer de la mémoire à l'objet. Ce style de programmation est sûr dans le sens où l'utilisateur n'a pas à se soucier des fuites de mémoire (il n'y en a pas, même en cas de levée d'exception), ni à désallouer explicitement la mémoire, ni à utiliser des systèmes tels le comptage de référence — qui induisent un surcoût arithmétique. Il n'y a pas non plus d'utilisation avant initialisation ou après destruction [5]. En outre, par système de portée (*scope*) la durée de vie des objets peut être assez finement choisie. De plus, nous nous autorisons les *vues* sur les objets alloués. Nous nous contraignons à utiliser des classes différentes pour les objets qui gèrent leur mémoire et ceux qui la partagent, par exemple `class myObjectOwner` et `class myObjectView`. La classe `myObjectView` pourra contenir une référence vers un objet `myObjectOwner` mais il lui sera interdit de modifier sa mémoire allouée. Nous appellerons *mère* une classe du première type et *vue* une du second type. Comme un objet de classe mère est créé et alloué avant toute vue sur lui, la portée des vues est inférieure à celle de la mère et la mère, désallouée seulement en fin de portée ne peut être désallouée avant ses vues.

Configuration et installation simplifiées, préalable à une bonne conception.

Le premier principe que nous pouvons énoncer est : *la bibliothèque doit être facile à configurer, compiler et installer.* En particulier, l'intervention de l'usager pendant ce processus doit être minimale. Nous nous sommes donc fortement penchés sur la question.

En particulier, compiler LinBox, Givaro ou FFLAS-FFPACK doit être aussi simple qu'un `ebuild` [6] de quelques lignes de `Gentoo Linux`. Les dépendances doivent être claires et les phases de préparation ou d'installation tenir en quelques opérations standards seulement. Les personnes en charge du paquetage doivent aussi avoir à fournir un nombre si possible nul de corrections (*patches*).

Ne pas respecter ce principe entraîne une perte d'efficacité lors de la distribution du code et une réaction de rejet de la part des utilisateurs. Nous proposons quelques solutions de base.

Les autotools : un système de construction puissant.

Les bibliothèques LinBox et Givaro sont toutes deux configurées et installées avec les autotools (*GNU build systems* [7]). Ce système est robuste, éprouvé. Il est vrai qu'il faut cependant s'habituer à des syntaxes parfois déroutantes (notamment les macros .m4). Même si d'autres systèmes paraissent plus 'sexy' ou modernes (scons, cmake), la tâche de portage vers un autre système aurait été trop fastidieuse. Au lieu de cela, nous avons préféré d'une part synchroniser les scripts avec Givaro pour une maintenabilité plus aisée et d'autre part respecter plus strictement les conventions m4 en utilisant de préférence leur syntaxe plutôt que celle tirée du bash et sujette à problèmes selon les interpréteurs. Nous avons aussi créé de nouvelles macros m4 pour simplifier la détection de certaines dépendances (IML, fplll, mpfr...), simplifier la détection selon les fournisseurs des BLAS et de LAPACK ou la détection des compilateurs — afin p.ex. d'adapter finement les options de compilation. Ainsi, nous fournissons un système facile à utiliser (`./configure --help`, `make install`) et assez simplement maintenable, réutilisable directement dans chacun des projets.

Une fois cette homogénéisation bien réglée, il restait le problème du développement de FFLAS-FFPACK, sous forme d'une liste d'en-têtes qu'un script pouvait copier dans les répertoires de LinBox adéquats, ces fichiers étant aussi versionnés dans LinBox. Cette bibliothèque était donc très difficilement maintenable car doublement versionnée : des changements pouvaient facilement s'opérer sur chacune de ces systèmes de révision. Nous avons donc décidé de donner une autonomie à la bibliothèque FFLAS-FFPACK. Aussi, nous avons créé à partir de la liste d'en-têtes de FFLAS-FFPACK une véritable bibliothèque basée sur ces mêmes autotools, avec ses exemples et ses tests propres. Au passage, nous avons extrait de LinBox tout ce qui appartenait à FFLAS-FFPACK. Il est ainsi plus facile de compiler et utiliser la bi-

5 — De préférence, nous utilisons la pile et non le tas, de sorte qu'une mère n'est jamais désallouée avant ses vues, ce qui ne serait pas garanti avec des `new`, `delete`.

6 — Un court script, *cf.* `http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2`

7 — *cf.* `http://www.gnu.org/software/automake/`.

bliothèque FFLAS-FFPACK et surtout de la tester et la documenter. Nous apportons ici une deuxième recommandation simple : garder séparés les projets dont les utilisateurs peuvent être distincts de manière à laisser à ces projets une liberté d'évolution.

Nous conclurons ce paragraphe sur ce *build system* des autotools avec la remarque qu'une installation ou désinstallation propre sont elles aussi importantes (notamment, ne pas oublier de fichiers à désinstaller et ne pas retourner une installation sur un code erreur [8]) ! Tester la bonne installation et désinstallation permet aussi de faciliter la diffusion du code !

Gestion de version des projets et inter-compatiblité clarifiée.

Ensuite, nous avons dû faire des choix sur l'évolution et les dépendances entre LinBox d'une part et FFLAS-FFPACK ou Givaro d'autre part. L'évolution des versions de ces bibliothèques se faisait souvent sur le dernier chiffre, quelle que soit l'étendue de l'évolution du code. Nous avons choisi de numéroter les versions selon le schéma plus classique `version_majeure.version_mineure.correction_de_bogue`. Cela permet d'une part d'être plus clair sur les évolutions apportées à LinBox (p.ex. dans le `ChangeLog` ou les `News` sur le site), mais aussi de pouvoir fixer des bogues sur des versions antérieures et assurer une maintenance des précédentes versions. Par exemple, Sage utilise encore la version `1.1.6` publiée il y a plusieurs années sans que l'on puisse inclure leur corrections (*patches*) autrement que dans une numérotation très lourde `1.1.6-rX`, puisque la prochaine version majeure publiée était la `1.1.7`. La bibliothèque LinBox n'évolue pas suffisamment vite pour nécessiter quatre niveaux de numérotation.

Le même système est appliqué à Givaro et FFLAS-FFPACK. Les dépendances se font de telle sorte que la publication `1.2.x` de LinBox doit être compatible avec la publication `3.4.y` de Givaro et `1.4.z` de FFLAS-FFPACK par exemple. De cette manière, nous devons assurer une certaine stabilité dans les API de Givaro ou FFLAS-FFPACK, notamment un changement d'API, même mineur, implique un changement de version (mineure !). Cela permet d'assurer une stabilité générale entre ces trois bibliothèques et des dépendances entre les versions bien définies, tout en laissant une certaine liberté à leur développement et leur maintenance.

Finalement, du côté du développeur, un script `incremente-version.sh` aide à incrémenter facilement les numéros de versions de ces projets — et en parallèle les *soname* des bibliothèques.

Test automatique de l'installation.

Afin de bien être sûr que les versions de Givaro ou FFLAS-FFPACK concordent avec celles demandées par LinBox, non seulement la macro `givaro-check.m4` ou `fflasffpack-check.m4` émettra un message clair en cas de désaccord, mais aussi un script `auto-install.sh` à la racine du source de LinBox donne explicitement les dépendances. Ce script récupère, configure et installe localement Givaro, FFLAS-FFPACK et ensuite configure et installe LinBox automatiquement ; il utilise bien sûr la chaîne des autotools afin de ne pas diverger du système de configuration et d'installation de LinBox. Des options supplémentaires permettent de s'adapter aux configurations non courantes — notamment des BLAS hors des chemins standards ou des vendeurs de BLAS particuliers.

La création d'un tel script permet, pour le développeur, après un `make dist` de vérifier que la version distribuée compile et s'installe parfaitement et permet aussi à l'utilisateur d'installer facilement LinBox et ses dépendances [9] :

```
sh auto-install.sh
```

Code 4.3 — Configuration et installation automatique de LinBox.

8 — Ce sont des remarques qui peuvent paraître triviales, mais certains paqueteurs vérifient les installations et les désinstallations : une fabrication de RPM échouera si `make install` renvoie un code d'erreur. Si nous prenons ce code d'erreur comme une blague « erreur, l'installation s'est bien passée » des personnes plus strictes y verrons un vrai problème.

9 — Ce script et de simplifier la tâche de l'utilisateur.

Diminution des temps de compilation.

Dans le même souci de faciliter l'expérience de l'utilisateur face à une bibliothèque d'en-têtes, il est important de produire une bibliothèque qui soit efficace à installer et à utiliser, nous parlons ici d'efficacité au sens de rapidité. La cible `make install` est extrêmement rapide car la plus grande partie du code de LinBox n'est pas compilée à l'installation. La bibliothèque LinBox est en effet essentiellement une bibliothèque d'en-têtes mais les mécanismes avancés de *template* utilisés rendent généralement la compilation coûteuse en temps en en mémoire. Pour des raisons d'efficacité, indépendamment des questions de durée et d'utilisation de la RAM, nous compilons les exemples avec des paramètres poussés d'optimisation (au minimum `-O2` [10]) afin que leur exécution soit efficace. Par contre, ce n'est pas nécessaire pour les tests : nous avons de nombreux tests et il faut trouver un juste milieu entre temps de compilation et temps d'exécution. Dans les tableaux 4.3 et 4.4, nous comparons les temps mis pour effectuer les tests basiques de LinBox avec diverses options d'optimisation — des plus basses `-O0` aux plus élevées `-O3` ou `-Ofast` en passant par `-Os` qui essaie de produire de petits exécutables. D'après

Tab. 4.3 — Temps de 'make check' avec LinBox-1.3.0, sur Joran@imag, avec gcc-4.7

options	-O0	-O1	-Os	-O2
temps (min)	**2,38**	5,28	4,45	7,14

Tab. 4.4 — Temps de 'make check -j2' avec LinBox-1.3.0, sur Joran@imag, avec gcc-4.7

options	-O0	-O1	-Os	-O2
temps (min)	**1,33**	3,12	2,37	4,2

les temps des tableaux 4.3 et 4.4, nous choisissons `-O0` comme option de compilation par défaut pour compiler et exécuter rapidement les tests. Cette remarque se reproduit pour Givaro ou FFLAS-FFPACK et vérifie que les tests effectués ne sont pas gourmands en ressources — le temps étant principalement passé lors de la compilation.

Plan de la troisième partie.

Dans cette introduction, nous avons fait une présentation rapide de la bibliothèque LinBox. Nous allons dans cette partie nous pencher sur des questions de généricité, d'efficacité, de robustesse, de fiabilité, d'évolutibilité.

- Dans un premier temps (chapitre 5), nous allons présenter un exemple de solution, la solution `mul` qui permet de multiplier des matrices ou des vecteurs. Nous introduirons d'abord des classes de vecteurs et de matrices générales avant de proposer divers algorithmes (en cascade, adaptatifs) dont la solution `mul` peut faire usage et qui fournissent de bonnes performances en pratique.
- Nous allons ensuite développer et exposer des techniques et manières de coder qui permettent de rendre efficace la bibliothèque sans pertes de généricité (chapitre 6).
- Dans le chapitre 7, nous développerons quelques problématiques liées à la qualité du code : comment le rendre plus sûr, plus stable, plus compréhensible, comment effectuer efficacement ce travail de supervision si le code ne respecte pas certains critères en premier lieu ?

10 — En principe, LinBox résiste très bien à un niveau d'optimisations élevé, mais nous sommes toujours à la merci de code mal généré par le compilateur lors d'optimisations trop poussées

"We share a philosophy about linear algebra: we think basis-free, we write basis-free, but when the chips are down we close the office door and compute with matrices like fury."

Irving Kaplansky

Chap. 5 Conception d'une multiplication matricielle efficace et générique : la solution `mul`.

Sommaire

5.1 Matrices et Vecteurs dans LinBox : simplification et homogénéisation. 83
5.1.1 Nouvelles matrices et vecteurs dans LinBox. 84
5.1.2 Opérateur `rebind` et modèle d'allocation. 86
5.1.3 Description des matrices BLAS et creuses. 86
5.1.4 Permutations. 89
5.1.5 Prolongements. 90
5.2 La solution mul. 90
5.2.1 Multiplication générique : système contrôleur–modules. 91
5.2.2 Nouvelles routines BLAS dans FFLAS. 92
5.2.3 Multiplication de matrices entières. 95
5.2.4 Conclusion, prolongements. 97

Une multiplication matricielle efficace et générique est une opération non triviale à implanter. Jusqu'à présent, la situation dans LinBox était la suivante : les multiplications matrices–matrices ou matrices–vecteurs étaient effectuées à l'intérieur de domaines [1] et bien souvent l'algorithme utilisé était une triple boucle sans que son choix soit paramétrable. Des problèmes d'interopérabilité entre ces classes, ainsi que des API très restrictives nous ont poussé à revoir ces choix. Par exemple, l'utilisation de structures de données de m4ri aurait conduit d'une part à la création d'une nouvelle classe de matrices, p.ex. `MariMatrix`, et à des modifications profondes dans les deux classes de domaines pré-citées, voire un nouveau domaine `MariDomain`. Nous pensons que cette approche n'est pas toujours judicieuse et à la vue de l'importance de l'opération *multiplication* en algèbre linéaire, nous avons décidé d'en faire une solution — libérée de tout domaine — et de développer des algorithmes spécifiques.

Avant de présenter la solution `mul` dans la section 5.2, nous allons définir proprement ses opérandes, à savoir matrices ou vecteurs dans la section 5.1.

5.1 Matrices et Vecteurs dans LinBox : simplification et homogénéisation.

Nous appelons *matrice dense* un tableau d'éléments d'un corps. Dans la série LinBox-1.1.x, on trouvait par exemple de nombreuses classes différentes pour les matrices denses (`BlasMatrix<_Element>`,

1 — classes qui implantents des opérations sur une certaine classe de matrice, par exemple les classes `BlasDomain`, ou `MatrixDomain`.

DenseSubmatrix<_Element> et DenseMatrixBase<_Element>, ou les BoîteNoire DenseMatrix<_Field> et BlasBlackbox<_Field>, voire DenseRowsMatrix non utilisée, *etc.*). Chacune de ces classes compte environ 200 lignes et souvent il est nécessaire de convertir une matrice dense vers une autre classe, éventuellement avoir à changer de domaine. Par exemple, la convention dans LinBox veut qu'une matrice soit une BlackBox (*cf.* code 5.3, [DGG+02]). Pour les matrices denses, prenons donc une DenseMatrix<Modular<double> > A. Si nous voulons bénéficier d'une multiplication rapide, il faut utiliser un BlasDomain (un MatrixDomain, compatible avec A, utilisera une triple boucle) et donc convertir A vers une BlasMatrix<double>, effectuant possiblement une copie profonde.

Nous avons voulu simplifier et ordonner tout ceci afin de n'avoir à effectuer de conversions que si les formats de données ou les corps changent et en profiter pour rendre certaines interfaces plus simples. Idéalement, nous ne voudrions utiliser que des Matrix ou des Vector et éventuellement subMatrix ou subVector pour les types de matrices qui sont effectivement des conteneurs (p.ex. matrices denses, creuses, structurées) par opposition aux matrices abstraites (Compose, Sum...).

5.1.1 Nouvelles matrices et vecteurs dans LinBox.

La première simplification que nous apportons consiste à *templéter* toutes les matrices et vecteurs par un corps au lieu du type de ses éléments. Cela permet aux matrices d'être indépendantes — en ne connaissant que ses éléments, une matrice ne sait pas faire d'opérations dessus — mais aussi une unification de l'interface de toute classe de matrice.

La seconde simplification consiste à décrire le format de la matrice dans un paramètre *template* [2], dont un défaut pourra être donné.

```
namespace matrixStorage {
struct dense {}
struct blas       : public dense { ... };  // stockage type BLAS
struct vectOfRows : public dense { ... };  // vecteurs de vecteurs
struct mari       : public dense { ... };  // structure de matrices de m4ri
...
} // matrixStorage
```

Code 5.1 — Exemple de formats de matrices denses.

Nous notons que nous préférons utiliser des noms d'espaces pour lister les traits plutôt que des classes car cela nous laisse plus de libertés — en particulier la liberté pour une personne utilisant LinBox de définir par ailleurs son propre format de matrice. L'implantation des matrices se fait alors par spécialisation. Nous pouvons donner des valeurs par défaut ou faire des alias, comme le montre le code 5.2.

```
template<class _Field, class _Storage = matrixStorage::blas>
class Matrix ;

template<class _Field>
class Matrix<_Field, matrixStorage::blas> {
public :
        typedef _Field                Field ;
        typedef matrixStorage::blas   storageType ;
        ...
};

template<class _Field>
class Matrix<_Field, matrixStorage::mari> {
```

2 — Notons l'analogie avec la STL : template < class T, class Allocator > class vector ;

```
        ...
};
```

Code 5.2 — Classes de matrices dans LinBox.

Interface commune des matrices.

En outre, nous forçons à ce que *toutes* les matrices partagent l'interface d'une BoîteNoire (code 5.3). En effet, cette interface regroupe la moindre des choses que l'on puisse demander à une matrice : une BoîteNoire correspond à une approche application linéaire d'une matrice ; seules les opérations matrices–vecteurs sont fournies ainsi que son corps de base et ses dimensions. Nous rajoutons un opérateur `rebind` qui permet d'effectuer un morphisme de corps — nous en reparlerons dans la section 5.1.2.

```
template< ... >
class Matrix {
public:
        /* constructeurs */
        ...

        /* y <- A.x ou y <- A^T.x */
        template<class _in, class _out>
        _out& apply(_out &y, const _in &x) const;
        template<class _in, class _out>
        _out& applyTranspose(_out &y, const _in &x) const;

        /* opérateur rebind */
        template<typename _Tp1>
        struct rebind ;

        /* dimensions */
        size_t rowdim() const;
        size_t coldim() const;

        /* accès au corps */
        const Field& field() const;
protected:
        /* interne */
        ...
};
```

Code 5.3 — Interface d'une BoîteNoire

Nous proposons donc cette architecture pour toutes les matrices dans LinBox. Nous remarquons que les membres `apply` et `applyTranspose` ne sont plus seulement templétés par des vecteurs mais peuvent tout aussi bien prendre en argument des matrices (utiles p.ex. dans les algorithmes par blocs). C'est une nouvelle interface de `apply` qui laisse plus de libertés.

Nous remarquons que cette interface n'est pas maximale. Par exemple, nous pourrions rajouter un membre `getEntry` — car l'élément (i, j) d'une matrice A est l'élément i du vecteur `A.apply(e_j)` où `e_j` est le j^e élément de la base canonique. Cependant, nous préférons ne pas fournir cette méthode par défaut car elle n'est pas performante. Dans les cas où elle n'est pas présente, une solution `getEntry` permet toutefois de récupérer cette fonctionnalité. Cela force le concepteur d'algorithmes à utiliser de préférence des `apply`, plus efficaces.

Vecteurs.

Une commodité largement utilisée dans LinBox consiste à utiliser les vecteurs fournis par la STL. Cependant, ils ne sont alors templétés que par des éléments, non un corps. Nous avons donc proposé de créer une classe unique `Vector` (code 5.4), sur le même modèle que `Matrix`.

```
namespace vectorStorage {
class blas           : public dense {};   // dense avec des strides
class mari           : public dense {};   // m4ri
class ratdense       : public dense {};   // couple vecteur/dénominateur commun
class sparseSeq      : public sparse {};  // creux
class canonicalBasis : public sparse {};  // base canonique
...
}
template<class _Field, class _Storage = VectorStorage::blas() >
class Vector ;
```

Code 5.4 — Vecteurs unifiés dans LinBox.

L'implantation n'est encore que partiellement effectuée mais elle est capitale dans le sens où une implantation propre des matrices impose une définition propre des vecteurs. Les formats principaux de vecteurs développés sont des formats denses (éventuellement avec des *strides*), des formats creux et des formats externes (comme m4ri).

5.1.2 Opérateur rebind et modèle d'allocation.

Nous rappelons que dans notre modèle d'allocation, nous avons une dichotomie entre les objets qui possèdent et gèrent leur mémoire et ceux qui la partagent avec d'autres. Notre modèle d'allocation nous impose en particulier de faire attention à l'opérateur `rebind` (*cf.* code 5.5). Cet opérateur est classique dans les allocateurs des structures standards de la STL : il permet au moyen d'un `typedef` de changer le type de base. Nous l'utilisons pour changer le corps de base d'une matrice, i.e. faire un morphisme de corps. Le `typedef ... other` doit donc se faire vers un type de matrice mère. Un `operator()` permet d'effectuer la conversion.

```
template <class _Matrix>
template<class _OtherField>
struct someMatrix< _Matrix >::rebind {
        // typedef typename _Matrix::template rebind<_OtherField> Rebinder
        typedef someMatrixOwner<Rebinder::other > other;

        void operator() (other & Ap, const Self_t& A) {
                // Rebinder () (...);
        }
};
```

Code 5.5 — Exemple d'opérateur `rebind`.

Ce mécanisme de `rebind` s'éloigne du standard mais permet une grande généricité lors des conversions. La classe `someMatrixOwner<T>` est une classe qui contient une copie de `T` ; l'appel à son opérateur `rebind` permet de construire effectivement `Ap`.

5.1.3 Description des matrices BLAS et creuses.

À l'opposé d'une BoîteNoire, une matrice sous format dense ou creux est l'approche « tableau de nombres » d'une matrice : chaque élément peut être atteint et modifié. Pour ce type de matrice, nous

pouvons fournir une interface plus riche (code 5.6). Notamment, nous avons des accès rapides en lecture/écriture aux éléments, et des itérateurs (sur les lignes, les colonnes, les éléments).

```
template<class _Element>
class Matrix {
public:
        /* constructeurs et  dimensions */
        ...
        /* resize */
        void resize (size_t m, size_t n);
        /* E/S */
        template <class Field>
        std::istream &read (std::istream &file, const Field &F);
        template <class Field>
        std::ostream &write (std::ostream &os, const Field &F) const;
        /* Éléments */
        void setEntry (size_t i, size_t j, const Element &a_ij);
        Element &refEntry (size_t i, size_t j);
        const Element &getEntry (size_t i, size_t j) const;
        Element &getEntry (Element &x, size_t i, size_t j) const;
        /* utils */
        Matrix &transpose (Matrix &M) const;
        /* Rebind */
        /* Row, Col... iterators */
};
```

Code 5.6 — Ancienne interface d'une matrice

Nous proposons une implantation des matrices au format BLAS (codes 5.7 et 5.8) dans notre modèle d'allocation :

```
template<class _Field>
class Matrix<_Field, matrixStorage::blas>
public :
        typename _Field::Element Element;
        ...
        void resize(size_t m, size_t n);
protected :
        size_t _row,_col ;
        std::vector<Element> _rep ;
};
```

Code 5.7 — Matrice dense (format BLAS)

```
template<class _Field>
class subMatrix<_Field, matrixStorage::blas> > {
        typename _Field::Element Element;
        ...
        // pas de resize ici
protected :
        size_t  _row, _col ;
        size_t  _r0, _c0 ;
        size_t  _stride ;
        Base_t& _ref ;
};
```

Code 5.8 — Sous-matrice dense (format BLAS)

Une Matrix en représentation BLAS alloue donc une matrice pleine sous forme d'un vecteur. Il lui est possible de modifier sa taille. Si cette matrice est de taille $m \times n$, alors ce vecteur aussi et l'élément (i, j) est à la place $i \times$ _col $+ j$. Dans une vue sur cette matrice BLAS, l'élément (i, j) est à la place $(i +$ _r0$) \times$ _stride $+ \big(j +$ _c0$\big)$.

À cette interface, nous avons rajouté notamment des membres getRow, refRow, setRow et leurs analogues pour les colonnes (Col) qui sont l'équivalent pour des vecteurs des fonctions existantes sur les entrées (*Entry).

Matrices creuses.

L'implantation des matrices creuses dans LinBox se faisait avec des vecteurs creux :

```
template <class _Element, class _Row>
class SparseMatrixBase<_Element, _Row, VectorCategories::SparseSequenceVectorTag
    >;
```

Code 5.9 — Matrice creuse dans LinBox-1.1.7.

Nous proposons, pour le format de stockage COO (figure 3.1), la classe suivante, plus générale, dans la même veine que les matrices denses. Nous utilisons un corps comme patrons et nous identifions les API (*cf.* code 5.10).

```
namespace matrixStorage {
        struct sparse {} ;
        struct COO : public sparse {} ;
        struct CSR : public sparse {} ;
        struct ELL : public sparse {} ;
        struct ELLR: public sparse {} ;
        struct HYB : public sparse {} ;
        ...
};

template<class _Field >
class Matrix<_Field, matrixStorage::COO> {
        //spécialisation par traits pour COO
};
```

Code 5.10 — Matrice creuse sous formats standards.

Toutes les matrices creuses sous un format donné portent le même nom Matrix, le patron _Storage permet de spécifier les formats de stockage. L'interface est la même que celles des Matrix sous format BLAS. Les membres getEntry, setEntry et refEntry peuvent nécessiter une recherche, souvent dans un tableau rangé par ordre croissant. Contrairement au cas dense, ces accès s'effectuent donc en $\mathcal{O}^{\sim}(1)$ au lieu de $\mathcal{O}(1)$. Par contre, setEntry doit insérer un élément et refEntry doit créer une référence sur un élément qui peut très bien être nul : il faut donc éventuellement insérer un élément nul. Cela peut provoquer divers problèmes. Dans certains cas (p.ex. ELL) on peut éventuellement utiliser le 0 comme un marqueur de fin de ligne, ce qui est alors faux ; ou alors l'insertion peut être très coûteuse — dans le cas du format ELL, si le nombre de colonnes maximal est dépassé, il faut en général ré-allouer un tableau avec une colonne de plus : on double la mémoire nécessaire. Les matrices creuses nécessitent beaucoup d'attention quant à la manipulation de leurs éléments (et leurs itérateurs) : il faut privilégier leur interface BoîteNoire.

Une autre différence se situe dans read et write où nous introduisons le support de différents formats :

- Le format sms dont le nombre d'éléments non nuls est inconnu et la lecture se fait jusqu'à l'apparition d'un triplé `0 0 0`, nécessitant une numérotation à partir de 1 et non de 0. Chaque ligne est un triplé `ligne colonne entrée`.
- Le format smf dont une majoration du nombre d'éléments non nuls nbnz est connu dès la première ligne et la lecture se fait jusqu'à la fin du fichier ou de nbnz lignes du fichier. La numérotation peut se faire alors à partir de 0.
- Ces mêmes formats de fichier, mais sous un format de donnée type CSR. Le format est identique aux formats sms et smf à ceci près qu'une ligne de données est constitué soit d'un couple formé par un numéro de colonne et une entrée soit d'un singleton -1. Chaque singleton correspond au saut d'une ligne (figure 5.1). Ce format permet de gagner environ 30% d'espace, vu que chaque ligne possède en

```
lig col nbnz
c1 e1
c2 e2
-1
c3 e3
⋮  ⋮
```

Fig.. 5.1 — Format de fichier CSR.

général deux entrées au lieu de trois.
- Il est prévu de supporter les formats MatrixMarket [3] et d'utiliser des commentaires en début de fichier pour compléter les spécifications répondant à nos attentes (types d'entrées plus vastes, autres formats supportés...).

Ajouts à l'interface générique.

Dans le cas des matrices denses, le problème du format de stockage ne se pose pas vraiment. Les matrices denses sont sous format BLAS, ligne d'abord. Dans LinBox, les matrices denses ne sont en pratique jamais représentées comme vecteurs de vecteurs ou colonne d'abord. Par contre, pour les matrices creuses, selon l'emploi que l'on veut en faire, le format du format stockage peut être un problème de base. Nous proposons des conversions de format avec des fonctions `import` et `export`. Le minimum requis pour ces fonctions est la conversion de/vers le format CSR. Au besoin, pour éviter les doubles conversions, des conversions particulières (p.ex. COO → ELL) peuvent être spécialisées.

La distinction entre éléments nuls ou non étant importante, nous avons des fonctions `clean` qui suppriment les 0 inutiles ou une fonction fonction `clearEntry` qui permet de spécialiser `setEntry(i, j, F.zero)`.

5.1.4 Permutations.

Un type très particulier de matrice est celui des permutations. Ces matrices n'ont pas directement besoin de corps puisque leurs entrées sont des naturels sur lesquels nous n'avons a priori pas d'opération à effectuer. Nous avons créé deux classes pour ces permutations :

```
template<class _UINT, class _Storage>
class PermutationMatrix ;

template<class _UINT, PermutationStorage::LAPACK>
class PermutationMatrix ;
template<class _UINT, PermutationStorage::Standard>
class PermutationMatrix ;
```

3 — *cf.* http://math.nist.gov/MatrixMarket/.

Code 5.11 — Matrices de permutations dans LinBox..

- La première est la représentation compressée des permutations que l'on retrouve dans de multiples routines LAPACK (alors couramment appelé ipiv). Un vecteur ipiv représente la décomposition de $\sigma \in \mathbf{S}_n$ en produit de transpositions $(1, \texttt{ipiv[1]}) \cdot (2, \texttt{ipiv[2]}) \cdots (r, \texttt{ipiv[r]})$. Cette décomposition est économique dans le sens où l'application de cette transposition à une matrice est l'itération d'un nombre limité d'échanges de lignes ou de colonnes et que l'application de la permutation inverse consiste simplement à appliquer les transpositions dans le sens inverse.
- La seconde correspond à la représentation naturelle : un vecteur s de taille n représente $\sigma \in \mathbf{S}_n$ tel que $s_i = \sigma(i)$.

Nous avons implanté ces classes en se basant sur l'interface unifiée des matrices. Nous avons préféré distinguer Matrix et PermutationMatrix car les corps de base sont très différents. À la limite, pourrions-nous considérer un corps UnparametricField<_Uint> ou un indexDomain pour unifier ces deux types de matrice. Dans la convention LAPACK, certaines fonctions comme transpose ou getEntry peuvent être très coûteuses. Pour y remédier, nous construisons à la demande une représentation standard, ce qui facilite certaines opérations. Nous y gagnons en complexité arithmétique — par exemple sur les fonction getEntry ou transpose avec un simple coût en n sur la mémoire.

5.1.5 Prolongements.

Ce travail de conception a notamment été pensé pendant les deux précédentes réunions autour de LinBox : DublinBox et RaleighBox. L'implantation pratique est en cours et s'effectue de manière incrémentale. Dans LinBox-1.3, les matrices sont toutes templétées par un corps, respectent toutes l'interface d'une BoîteNoire et respectent le modèle d'allocation vue/mère. Terminer ce travail est un objectif majeur avant LinBox-2.0.

Nous avons donc réussi à unifier toutes les matrices de LinBox dans une notation assez simple. Le prochain pas à franchir entraîne la création d'un apply performant. Nous proposons pour cela de créer une solution, nommée mul, pour les matrices–conteneurs. Sa conception constitue l'objet de la section suivante.

5.2 La solution mul.

Nous nous attachons dans cette section à décrire comment implémenter génériquement et efficacement une solution qui permet les multiplications entre matrices et avec des vecteurs. Implémenter une solution mul, c'est fournir une solution de niveau BLAS ; c'est une routine de base d'algèbre linéaire et dans cet esprit, nous plaçons la multiplication matrice–vecteur dans un espace de noms blas2 [4] et la multiplication matrice–matrice dans blas3. Ces espaces de noms apportent du sens — favorisent la compréhension du relecteur — au code qui les utilisent. Cependant, contrairement aux routines BLAS dont les spécifications sont très strictes, notre solution mul doit s'adapter à divers structures de données et corps de base.

Nous avons donc voulu créer une solution mul.h dont l'interface est une simple fonction *templétée* :

```
/* C = AB */
template<class _Matrix1, class _Matrix2, class _Matrix3, class _Method>
_Matrix1& mul ( _Matrix1& C, const _Matrix2& A, const _Matrix3& B
              , const _Method& myMehtod = mul3Method::default() ) const ;
```

Selon la méthode choisie (triple boucle, Strassen, restes chinois, *etc.*), la nature du corps (entier, modulaire, *etc.*) et le type de matrice (BLAS, m4ri, creuse, *etc.*) cette solution appelle divers algorithmes.

4 — par analogie avec les niveaux 2 et 3 des spécifications BLAS.

Le seul pré-requis est que `_Matrix1` possède un membre `setEntry`. En particulier, l'interface des `BoîteNoire` actuelle peut alors prendre la forme, au lieu du traditionnel domaine plus contraignant :

```
template<class _outVector, class _inVector>
OutVector &apply (_outVector &y, const _inVector &x) const
{
        //return _MD.vectorMul (y, *this, x);
        return blas2::mul(y, *this, x);
}
```

Code 5.12 — Implantation d'`apply` sur un vecteur par solution `mul`.

5.2.1 Multiplication générique : système contrôleur–modules.

Nous allons maintenant donner un patron d'algorithme générique et efficace que nous appliquons dans le cas de la multiplication dense. Ce modèle permet d'obtenir des performances au moins aussi bonnes que celles du meilleur algorithme à disposition.

Seuils et Cascades.

Supposons que nous implémentions un algorithme récursif dont nous puissions choisir les cas de bases. Comment implanter génériquement un tel algorithme dont l'efficacité soit garantie ?

Nous proposons un modèle, dans lequel les appels récursifs se font tous dans une procédure séparée (les modules) et appellent tous en retour une routine de base (le contrôleur). Cette façon de coder présente un double intérêt ; le contrôleur choisit toujours la meilleure routine qu'il a à sa disposition et les modules fonctionnent comme des routines spécialisées, voire peuvent être considérées comme système de greffons (*plug-in*). Nous schématisons ce patron dans la figure 5.2.

À notre connaissance, il n'existe pas de patron de conception qui gouverne cette construction. Le patron *stratégie* s'en rapproche le plus. Nous proposons de faire une analogie avec des contrôleurs pour systèmes dynamiques — une fois que le contrôleur a envoyé une correction au système, il reçoit de celui-ci une nouvelle mesure qui lui permet de faire à nouveau une correction.

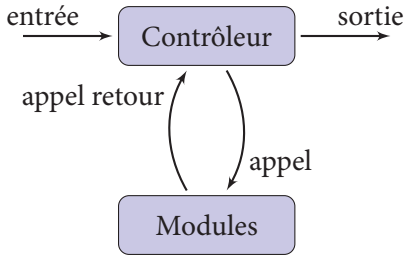

Fig.. 5.2 — Patron de conception contrôleur–modules

Dans la figure 5.3, nous illustrons cette idée sur un exemple de multiplication matricielle dense utilisant un algorithme récursif de type Winograd (tableau 1.1) et p.ex. un cas de base type `BLAS`.

Nous prouvons dans les figures 5.4 à 5.6 le bien fondé d'un tel algorithme en *cascade*. L'utilisation de ces seuils permet en effet de diminuer les complexités spatiales et arithmétiques des divers algorithmes. Nous traçons les complexités relatives théoriques pour les trois algorithmes suivants, en utilisant les formules de la section 1.2 :

— l'algorithme naïf seul (classique, triple boucle) ;
— l'algorithme de Winograd seul (tableau 1.1) ;
— l'algorithme 5.1 avec pour `BaseCase` l'algorithme naïf et pour `RecursiveCase` l'algorithme de Winograd.

Algorithme 5.1 : `AlgoSeuil` : contrôleur

Entrées : A et B, denses, de dimensions resp. $n \times k$ et $k \times n$.
Sorties : $C = A \times B$
si $\min(m, k, n) < t$ **alors**
  BaseCase(C,A,B) `/* BLAS rapide */`
sinon
  RecursiveCase(C,A,B,t)
fin
retourner C ;

Algorithme 5.2 : `RecursiveCase` : module récursif

Entrées : A, B, C et t comme ci-contre.
Sorties : $C = A \times B$
Découpe A,B,C en S_i, T_i ⋯
...
P_i = `AlgoSeuil`(S_i, T_i, t)
...
retourner C

Fig.. 5.3 — Conception d'un algorithme récursif contrôlé

Nous remarquons que pour la complexité algorithmique, dès un seuil de 300, on a théoriquement un gain. Par contre, en complexité spatiale, il faut limiter le nombre d'appels récursifs pour qu'il y ait un réel intérêt. En revanche, le nombre d'allocations est drastiquement réduit dans l'algorithme en cascade.

Cette méthode tout à fait générale de conception contrôleur–module permet d'une part la réutilisation des modules et d'autre part assure automatiquement l'efficacité. Par exemple, une technique de cascade permet de fournir un algorithme global toujours plus rapide que les modules particuliers. En outre, le contrôleur a une totale liberté de choix de modules, pouvant par exemple s'adapter à l'architecture (*cf.* benchmarks), aux ressources (notamment la mémoire) et aux modules disponibles. La conception d'un contrôleur doit toutefois respecter une règle : les cycles processeurs utilisés pour faire les choix doivent être réduits, p.ex. limités à quelques branchements ou `switches`.

5.2.2 Nouvelles routines BLAS dans FFLAS.

Dans l'élaboration d'une multiplication générique, nous nous retrouvons souvent devant des algorithmes très spécifiques à implanter. Un exemple provient de la création rapide de matrices aléatoires de rang donné pour laquelle nous avons choisi de multiplier une matrice triangulaire inférieure inversible avec une matrice triangulaire supérieure de rang donné. Utiliser une routine type `ftrmm` qui prend en argument une matrice triangulaire et une matrice dense n'est pas optimal : nous avons donc proposé des algorithmes en place, qui permettent de multiplier plus rapidement des matrices triangulaires entre elles.

5.2.2.1 Multiplication de matrices triangulaires.

Il n'existe pour le moment pas de multiplication rapide de matrices triangulaires autre que ftrmm (*Triangular matrix-matrix multiplication*) pour la multiplication d'une matrice triangulaire et d'une matrice dense. Cette opération fait partie des spécifications `BLAS` avec la signature `(x)trmm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB)`. L'opération est faite en place dans la matrice rectangulaire dense B. Nous cherchons donc à créer une opération (nommée `ftrtrm`) qui permet de multiplier entre elles deux matrices triangulaires A et B, en place dans la seconde B. La signature proposée est :

```
ftrtrm(side, uploA, trans, diag, uploB, n, alpha, A, ldA, B, ldB) ;
```

Code 5.13 — `ftrtrm`

L'opération effectuée est $B \leftarrow \alpha op(A) \times B$ ou $B \leftarrow \alpha B \times op(A)$ selon le paramètre `side`. La diagonale de A est implicitement unitaire selon le paramètre `diag`. Selon le paramètre `trans`, soit $op(A) = {}^{T}A$ soit $op(A) = A$. Les matrices A et B sont considérées triangulaires inférieures ou supérieures selon les paramètres `uploA` et `uploB`.

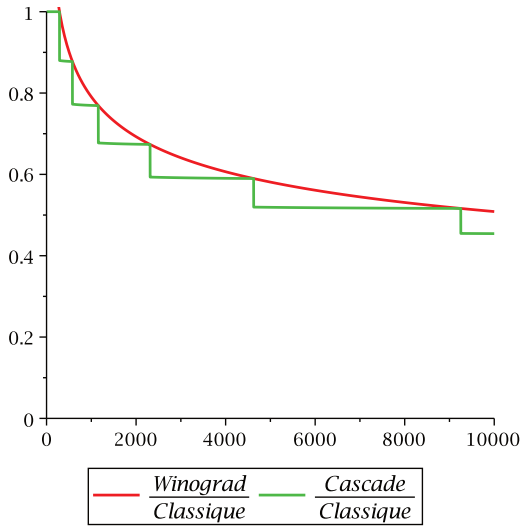

Fig.. 5.4 — Complexité algorithmique de la cascade Winograd-BLAS

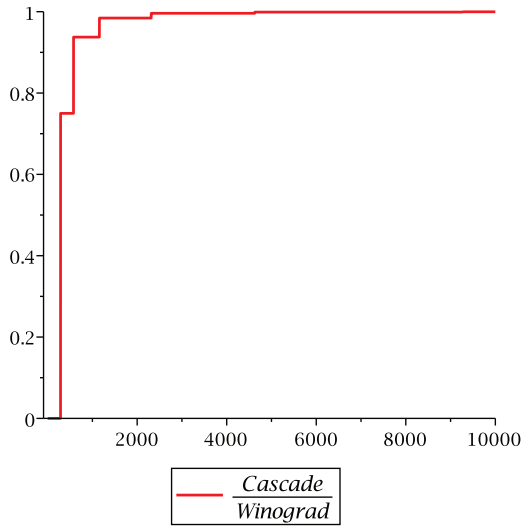

Fig.. 5.5 — Complexité spatiale de la cascade Winograd-BLAS

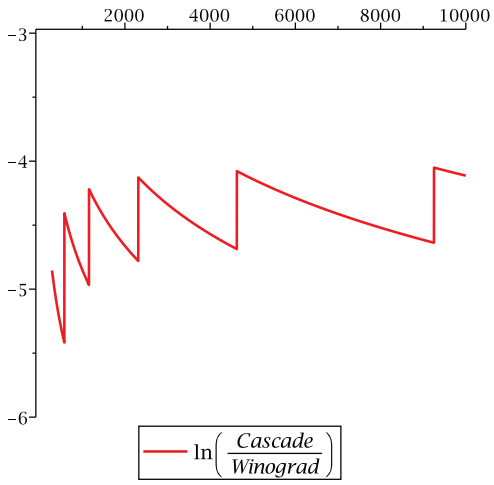

Fig.. 5.6 — Nombre d'allocations total pour la cascade Winograd-BLAS

La difficulté algorithmique provient du fait que les données à l'extérieur du tableau triangulaire de taille $n \times n$ dans la matrice (B, `ldB`) ne doivent pas être modifiées. Par exemple, lorsque les matrices triangulaires inférieures et supérieures sont superposées dans un même tableau de données, le résultat, représenté par la matrice entière, doit se retrouver calculé correctement en place.

Nous découpons les matrices comme suit.

$$L = \left(\begin{array}{c|c} \begin{matrix} * & \\ L_1 & \end{matrix} & * \\ \hline M & \begin{matrix} * \\ L_2 \end{matrix} \end{array}\right) \qquad U = \left(\begin{array}{c|c} \begin{matrix} U_1 & \\ * & \end{matrix} & N \\ \hline * & \begin{matrix} U_2 \\ * \end{matrix} \end{array}\right)$$

Nous proposons les algorithmes suivants, pour lesquels nous supposons que nous avons une fonction `fgemm`(`uplo`, A, `uplo`, B, C) qui effectue l'opération $C = C + \mathtt{op}(A) \times \mathtt{op}(B)$. De plus, dans la suite, les matrices triangulaires inférieures ou supérieures en argument seront notées L ou U et la matrice résultat, notée B plus haut, est découpée en quadrants :

$$B = \left(\begin{array}{c|c} B_{NW} & B_{NE} \\ \hline B_{SW} & B_{SE} \end{array}\right).$$

Dans l'algorithme 5.3, nous proposons un ordonnancement pour l'opération `ftrtr(left, istrans, L, U)`, i.e. $U \leftarrow \mathtt{op}(L) \times U$. Nous notons que la copie effectuée à l'étape 3 n'est utile que lorsque L et U ne sont pas stockées dans la même matrice (i.e. en arithmétique de pointeurs `L+n/2 !=U`)

Algorithme 5.3 : `ftrtr(left, istrans, L, U)`

	$B_{SE} \leftarrow$ `ftrtr(left, istrans,` L_2, U_2`)`	`/* recursive */`
1	$B_{SE} \leftarrow$ `gemm(uplo, istrans, M, notrans, N)`	`/* dense accumulation */`
2	$B_{NW} \leftarrow$ `ftrmm(left, istrans,` L_1`, N)`	`/* ftrmm */`
3	$B_{SW} \leftarrow M$	`/* copy */`
4	$B_{SW} \leftarrow$ `ftrmm(right, istrans,` U_1`, M)`	`/* ftrmm */`
5	$B_{NW} \leftarrow$ `ftrtr(left, istrans,` L_1, U_1`)`	`/* recursive */`

Les autres combinaisons de paramètres pour `ftrtrm` sont des variantes de l'algorithme 5.3. Nous montrons dans l'algorithme 5.4 comment adapter cet algorithme pour obtenir par exemple `ftrtr(left, istrans, U, L)`.

Algorithme 5.4 : `ftrtr(left, istrans, U, L)`

$B_{NW} \leftarrow$ `ftrtr(left, istrans,` U_1, L_1`)`	`/* recursive */`
$B_{NW} \leftarrow$ `fgemm(N, istrans, M, notrans,` B_{NW}`)`	`/* dense accumulation */`
$B_{NE} \leftarrow$ `ftrmm(right, istrans, N,` L_2`)`	`/* ftrmm */`
$B_{SW} \leftarrow M$	`/* copy */`
$B_{SW} \leftarrow$ `ftrmm(left, istrans,` U_2`, M)`	`/* ftrmm */`
$B_{SE} \leftarrow$ `ftrtr(left, istrans,` U_2, L_2`)`	`/* recursive */`

De même, des ordonnancements adéquats fournissent aisément les algorithmes correspondant à `ftrtr(*, *, L, L)` et `ftrtr(*, *, U, U)` (p.ex. algorithme 5.5).

Algorithme 5.5 : `ftrtr(right, istrans,` $L^{(1)}, L^{(2)}$`)`

$B_{SW} \leftarrow$ `ftrmm(right, istrans,` $L_1^{(1)}, M^{(2)}$`)`	`/* trmm */`
$B_{SW} \leftarrow B_{SW} +$ `ftrmm(left, istrans,` $L_2^{(2)}, M^{(1)}$`)`	`/* trmm dans` L_1 `*/`
$B_{NW} \leftarrow$ `ftrtrm(right, istrans,` $L_1^{(1)}, L_1^{(2)}$, M, `notrans,` B_{NW}`)`	`/* Recursive */`
$B_{SE} \leftarrow$ `ftrtrm(right, istrans,` $L_2^{(1)}, L_2^{(2)}$, M, `notrans,` B_{NW}`)`	`/* Recursive */`

Rappel des complexités

Nous rappelons les complexités des algorithmes pour les ordonnancements proposés telles que calculées dans [DGP08], et en utilisant une multiplication rapide en $\mathrm{M}(n) = 6n^{\omega}$:

$$\mathrm{W}_{\texttt{ftrmm}}(n) = \frac{2}{2^{\omega} - 4}\mathrm{M}(n).$$

La complexité de l'algorithme `ftrtrm`, que la copie soit effectuée ou non effectuée, vérifie :

$$\mathrm{W}_{\texttt{ftrtrm-LU}}(n) = \frac{2^{\omega}}{(2^{\omega} - 2)(2^{\omega} - 4)}\mathrm{M}(n).$$

Dans le cas non symétrique, nous avons :

$$\mathrm{W}_{\texttt{ftrtrm-LL}}(n) = \frac{4}{(2^{\omega} - 2)(2^{\omega} - 4)}\mathrm{M}(n).$$

Cet algorithme peut ainsi être utilisé dans la solution `mul` lorsque les matrices A et B sont triangulaires denses — diminuant le terme constant devant n^{ω} de 4 à 2,8 ou 1,6 pour la multiplication rapide de Winograd.

5.2.2.2 Multiplication modulaire sur types entiers.

Les routines BLAS ont d'abord été créées pour les numériciens. Les types de données supportés sont les réels en simple et double précision (`float` et `double`) et les complexes (simple et double précision aussi). En particulier, il n'existe pas de routines pour les types entiers (ni pour la quadruple précision). Avec les nouvelles instructions SIMD sur les types entiers (SSE, AVX), il est raisonnable d'envisager tester les performances de petits noyaux tels `fgemm` sur les entiers. En outre, les opérations sur les types entiers non signés se font en fait modulo 2^{32} ou 2^{64} : pour ces moduli particuliers, la réduction modulaire n'est alors pas nécessaire et il est possible d'en tirer partie par exemple dans un cas de base d'un schéma de restes chinois.

5.2.3 Multiplication de matrices entières.

La multiplication de matrices entières apporte diverses complications par rapport à celle sur les corps finis. En particulier, la taille de la matrice résultat n'est pas connue à l'avance et généralement bien plus importante que celle des matrices opérandes. Aussi, la multiplication atomique n'est pas aussi rapide que celle sur les types natifs et les structures de données représentant les entiers longs sont plus compliquées. Toutes ces remarques vont nous causer bien des problèmes. Pour nous attaquer à ce problème, nous considérons divers algorithmes : l'algorithme naïf, les restes chinois et celui de Strassen–Winograd avec comme cas de base l'un des deux précédents.

5.2.3.1 L'algorithme triple boucle.

Cet algorithme est le plus facile à implanter et utilise directement les entiers en multi-précision de gmp. Une simple triple boucle `ijk` ou `jik` fournira généralement de bonnes performances sur les petites matrices ou petits entiers. Au prix de quelques efforts (pré-transposition de sous-matrices de B ou spécialisation de l'opération atomique `mul` selon la taille des opérandes sur chaque coefficient de A et de B comme dans FLINT, *etc.*) il est possible d'améliorer sensiblement les performances d'une triple boucle naïve. La parallélisation via OpenMP de cette triple boucle est aussi assez aisée. La complexité d'un tel algorithme est $2n^3 l(b)$ si $l(b)$ est la complexité pour multiplier ou additionner deux entiers de taille b en entrée et si n est la taille d'une matrice (carrée pour simplifier).

5.2.3.2 Restes chinois.

Il existe de nombreuses implantation des restes chinois — avec ou sans terminaison anticipée, ou des systèmes de résidus (RNS, *residue number system*), des bases combinées (*mixed radix*) — que l'on re-

trouve en particulier dans LinBox, Givaro, IML ou FLINT. Nous renvoyons par exemple à [DGR10, CS05]. La méthode des restes chinois est très efficace pour les grands entiers et est parallélisable (notamment le calcul des $\left(A \mod p_i\right) \times \left(B \mod p_i\right)$. Si nous supposons que nos entrées sont de taille b et que nous utilisons une multiplication de matrice rapide sur les résidus $\mathbf{Z}/p_i\mathbf{Z}$ en $M(n) = 6n^{\omega}$, la complexité de l'algorithme des restes chinois est $\mathcal{O}(kM(n) + kbn^2)$ où k est le nombre de nombres premiers requis, par exemple si chacun a une taille 2^{β}, alors $k = \frac{2b+\log_2(n)}{\beta}$.

5.2.3.3 Algorithme de Winograd.

Il existe toujours la possibilité d'implanter un algorithme de Winograd. Dans ce cas, il faut faire attention à la mémoire supplémentaire utilisée car les tailles des temporaires sont arbitrairement grandes, à la grande différence de la multiplication sur un $\mathbf{Z}/p\mathbf{Z}$ où les coûts des opérations atomiques sont unitaires. Si $M(n, b)$ est le coût d'une multiplication entière dans le cas de base, alors la complexité de l'algorithme de Winograd — *cf.* algorithme 0.2 avec 8 pré-additions, 7 multiplications et 7 post-additions — avec un seul niveau de récursion est $8{}^{n}\!/_{2}{}^{2}l(b) + 7M\left({}^{n}\!/_{2}, \mathcal{O}(b)\right) + 7{}^{n}\!/_{2}{}^{2}l\left(2b + \log_2\left({}^{n}\!/_{2}\right)\right)$.

5.2.3.4 Construction d'un algorithme de multiplication entière.

L'efficacité d'une multiplication entière dépend beaucoup de la taille des coefficients. Mis à part les petites tailles de matrice, c'est donc un paramètre nécessaire à calculer avant la multiplication à proprement parler afin de choisir l'algorithme adéquat — sauf éventuellement pour un algorithme à terminaison anticipée.

Tab. 5.1 — Comparaison de divers algorithmes pour la multiplication matricielle entière (normalisé par $2n^3 \max(b_A, b_B)/sec. \times 10^{-9}$) sur Joran@imag.

matrices			algorithme					
n	b_A	b_B	naïf	CRA	FLINT	Wino–naïf	Wino–CRA	Wino–FLINT
100	10	10	0,7	1,3	15	0,6	0,58	5,0
100	1 000	1 000	4,1	2,4	7,5	4,5	1,4	4,5
100	1 000	5 000	4,7	2,7	4,6	5,4	1,8	5,4
10	5 000	5 000	1	0,1	1,8	1,2	0,03	1,5
16	5 000	5 000	1,3	0,2	1,1	1,4	0,08	1,3
50	5 000	5 000	1,4	0,8	1,4	1,5	0,5	1,6
500	500	500	4,8	10	22	5,4	6,7	18
1 000	10	10	0,3	10,5	14,8	0,3	5,8	11,8
1 000	100	100	2	20	29	2	12	25
2 000	50	50	1,5	26,7	39	1,5	17	36
3 000	5	5	—	16,5	12,8	—	8,4	11,3

Nous constatons que l'implantation des restes chinois de FLINT est en général significativement plus performante que celle de LinBox et que les « petits » cas sont souvent mieux traités aussi. En outre, l'utilisation de l'algorithme de Winograd peut entraîner des améliorations — avec les algorithmes utilisés, cela apparaît sur les petites tailles de matrices et grandes tailles en entrée. L'efficacité de la multiplication dépend ici autant de la complexité que de l'implantation. Nous proposons donc de fournir tous les algorithmes à disposition de manière à ce qu'un autre mécanisme (*cf.* section 6.1) permette de choisir le meilleur dans un cas donné. En effet, prenons l'exemple de FLINT pour lequel le choix de basculement entre l'algorithme triple boucle et les restes chinois se fait à partir de constantes indépendantes de l'environnement. Il se trouve qu'entre les lignes 4 et 5 du tableau, l'algorithme de Winograd avec comme

cas de base la routine de FLINT prend le pas sur la routine de `fmpz_mul` de FLINT. Un meilleur seuil dans cette dernière aurait pu éviter cela et rendre `fmpz_mul` inconditionnellement plus rapide [5].

Une bonne implantation de multiplication sur $\mathbf{Z}$ doit donc déjà prendre en compte la taille des matrices et des entrées mais aussi de l'environnement et des performances des algorithmes de base. Nous mettrons en œuvre (chapitre 6) des techniques pour parvenir à s'adapter à l'environnement et aux algorithmes à disposition, tandis que les choix des algorithmes seront effectués essentiellement par traits.

Implantation par traits.

Finalement, l'implantation de chaque algorithme (code 5.14) se fait donc par *traits* [6] et la solution, selon les dimensions des matrices, la taille des entrées et les paramètres d'optimisation (*cf.* plus bas) utilisera l'algorithme choisi.

```
namespace Protected {
        template<class Matrix1, class Matrix2, class Matrix3>
        Matrix1& mul(Matrix1                                & C,
                     const Matrix2                          & A,
                     const Matrix3                          & B,
                     const RingCategories::IntegerTag       & tag,
                     const MMMethod::CRA                    & meth)
        {
                ...
        }
}

template<class Matrix1, class Matrix2, class Matrix3, class Method>
Matrix1& mul(Matrix1& C, const Matrix2& A, const Matrix3& B, const Method& meth)
    {
        typedef typename FieldTraits<typename Matrix1::Field> matField ;
        return Protected::mul(C,A,B,matField::categoryTag(), meth);
}
```

Code 5.14 — Sélection des algorithmes par traits.

Cette méthode de programmation permet d'effectuer des choix à la compilation et offre une facilité d'écriture : le nom de la fonction appelante est constant, seuls les traits changent, c'est un exemple typique de conception par blocs de base.

5.2.4 Conclusion, prolongements.

Nous avons vu dans cette section diverses approches pour effectuer une multiplication matrice–matrice. Nous avons conçu des algorithmes :

— en cascade qui permettent d'avoir de très bonnes complexités ;

— des algorithmes spécifiques aux propriétés des matrices.

Nous proposons donc un ensemble d'algorithmes, de briques élémentaires que la solution `mul` adapte en fonction des caractéristiques des matrices et, comme nous le verrons dans un prochain chapitre, en fonction de l'environnement.

5 — Après, à nous développeurs de LinBox de partir de cette situation pour améliorer nos restes chinois, FLINT n'utilisant même pas de routines BLAS pour leurs cas de base... !

6 — *tag dispatching*,*cf.* `http://www.boost.org/community/generic_programming.html`, aussi communément utilisé dans la STL.

Formes de Hermite.

Nous généralisons cette approche à d'autres solutions. Par exemple, nous avons introduit dans LinBox le calcul de la forme de Hermite d'une matrice dense[7]. Nous en avons donc implémenté divers algorithmes. Tout d'abord, il y a l'algorithme classique de pgcd (voir par exemple l'algorithme ClassicHermite, [Sto94, chapitre 3]). Nous avons aussi implanté l'algorithme de Kannan–Bachem ([KB79]) amélioré par Chou–Collins ([CC82]) ainsi que l'amélioration de l'algorithme de Micciancio–Warinschi ([MW01]) proposée par [PS10] (et implantée par les auteurs dans Sage et partiellement dans LinBox). Parmi les algorithmes avec réduction modulaire, nous avons implanté une variante de l'algorithme classique par pgcd et, dans ses cas d'utilisation, utilisé la routine NTL::HNF de NTL. Dans le cas où nous cherchons à réduire la taille des coefficients de U, nous avons aussi implanté l'algorithme de Havas–Majewski–Matthews ([HMM98]). Finalement, nous avons modifié l'algorithme de [GJS01] dans le cas où la partie essentielle est supposée petite — et fait donc la simplification que lors du premier appel récursif, nous pouvons supposer que nous n'obtenons aucun autre élément diagonal que l'unité.

L'implantation d'un algorithme de Hermite a été faite avec le prototype du code 5.15. Selon le paramètre template, la matrice unitaire est, ou n'est pas, calculée. La méthode fait partie d'un espace de nom (*namespace*) nommé hermiteMethod. Nous avons quatre choix d'implantation : la matrice H est triangulaire inférieure ou supérieure, la matrice U est appliquée à droite ou à gauche (lignes ou colonnes). La fonction hermiteInBase fait partie des algorithmes, les détails étant placés dans l'espace de noms Protected habituel.

```
template< class Ring
        , bool withU >
BlasMatrix<Ring> &
hermiteInBase( BlasMatrix<Ring>                & H
             , BlasMatrix<Ring>                & U
             , const hermiteMethod::hermiteGCD & meth
             , const implementedLeftLower      &
             )
```

Code 5.15 — Implantation de l'algorithme Hermite.

Lorsqu'une méthode n'est pas implantée, une routine hermiteSwitcher effectue les transformations suivantes qui permettent de passer d'une décomposition à droite à une décomposition à gauche et d'une décomposition selon les lignes à une décomposition selon les colonnes :

— transpose(hermite(A, right)) = hermite(transpose(A, left));

— si J est une matrice anti-diagonale, J.hermite(A, left).J = hermite(J.A.J, right).

```
template< class Ring
        , enum LinBoxTag::Side s
        , enum LinBoxTag::Shape t
        , class myMethod >
BlasMatrix<Ring>&
hermiteIn( BlasMatrix<Ring>& H
         , const myMethod  & meth = hermiteMethod::hermiteGCD()
              )
{
        BlasMatrix<Ring> U(H.field(), 0, 0);
        const bool withU = false;
        Protected::hermiteSwitcher<Ring, s, t, withU, myMethod>(Z, H, U, meth);
        return H;
```

7 — L'effort a été porté sur les matrices denses. En effet, des techniques, comme par exemple celle présentée dans [BM10], permettent de réduire le problème sur des matrices creuses à des matrices denses — une élimination heuristique est effectuée jusqu'à ce que la matrice soit suffisamment densifiée pour lancer un algorithme dense rapide.

```
}

...

template< class Ring
        , enum LinBoxTag::Side s
        , enum LinBoxTag::Shape t
        , class myMethod >
BlasMatrix<Ring>&
hermite( BlasMatrix<Ring>       & H
       , const BlasMatrix<Ring> & A
       , const myMethod         & meth = hermiteMethod::hermiteGCD()
{
        H = A ;
        hermiteIn<Ring, s, t>(H, meth);
        return H;
}
```

Code 5.16 — Implantation de la solution Hermite

Nous obtenons un ensemble d'implantations de la forme de Hermite pour des matrices denses sur les entiers et, avec une interface commune, les diverses définitions de décompositions de Hermite que l'on peut trouver dans la littérature.

Résolution de systèmes entiers.

Nous nous sommes aussi intéressés à la résolution de systèmes entiers, pour lesquels nous avons à disposition des implantations de l'algorithme de Dixon ([Dix82]) — dans LinBox ([Gio04]), mais aussi IML ([CS05]) — et proposons leur adaptation avec l'algorithme de Moenck et Carter ([MC79]). Nous montrons que nous pouvons directement utiliser les routines d'IML dans LinBox, sans conversions de données — à la différence de FLINT — , et bénéficions ainsi directement de ces routines.

```
BlasMatrix<PID_integer> A(PID_integer(), m, n);
mpz_t * mp_A = preinterpret_cast<mpz_t*>(A.getPointer());
...
IML::nonsingSolvLlhsMM(IML::RightSolu,B.rowdim(),1, mp_A, mp_B, mp_N, mp_D);
```

Code 5.17 — IML wrapping

D'autre part, il existe des algorithmes numériques–symboliques qui sont nettement plus rapides dans leur domaine d'utilisation (entrées de taille en générale inférieure à 40 bits), citons par exemple [Wan06, SWY11], implantés par leurs auteurs dans LinBox. Le travail de la solution `solve`, là encore consistera à choisir en fonction des bons paramètres le meilleur algorithme à disposition parmi toutes les briques offertes.

"Breakage is the price you have to pay for advancements"

Martin Daleski

Chap. 6 Optimisation des performances.

Sommaire

6.1 Garantir des performances par leur mesure (*benchmarking*). 102
6.1.1 Des *benchmarks* pour mesurer, tester, paramétrer. 102
6.1.2 Une *framework* de *benchmarks* partagée par les utilisateurs et l'optimiseur. 103
6.1.3 Prolongements. 106
6.2 Amélioration des performances : conception par briques de base. 107
6.2.1 Optimisations, micro-optimisations : défi ou temps perdu ? 108
6.2.2 Conception générique par *building blocks* (briques modulables). 109
6.2.3 Conclusion. 112

De nombreux logiciels et bibliothèques sont optimisés lors de leur installation. L'un des plus célèbres parmi ceux que nous utilisons est certainement ATLAS : *Automatically Tuned Linear Algebra Subroutines*, *cf.* [WPD01]. Citons aussi par exemple NTL. Plus récemment, une option `-pgo` (*profile guided optimisations*) a été introduite pour le compilateur `gcc` qui permet de définir les codes générés les plus rapides sur l'environnement de compilation. Le point commun à tous ces exemples est une machinerie permettant, avant l'installation, de découvrir les meilleurs paramètres pour une architecture donnée.

Dans la bibliothèque FFLAS-FFPACK en particulier, il existe notamment une option de configuration `--enable-optimizations` qui permet de détecter le meilleur seuil de basculement entre les algorithmes de Winograd et ceux des BLAS pour la multiplication matricielle sur des `double`. Nous voulons aller plus loin. Cependant, à la différence des exemples sus-cités, nos choix d'optimisations peuvent se situer à des niveaux très divers, et compliquer donc notre tâche :

- Au niveau purement algorithmique, nous pouvons écrire de meilleurs algorithmes avec de meilleurs complexités, implanter des algorithmes plus efficaces dans certains cas particuliers, détecter les meilleurs algorithmes en fonction de divers paramètres.
- Au niveau des routines de base, nous avons souvent le choix entre diverses implantations, dont les performances peuvent significativement varier selon les architectures de processeurs par exemple.

Nous allons donc apporter (section 6.1) des propositions d'auto-optimisation qui seront assez génériques pour recouvrir nos besoins, du choix de la réduction modulaire en fonction du type d'argument au choix de la méthode de résolution de système en fonction des caractéristiques de la matrice en entrée, en passant par la sélection des meilleurs algorithmes de restes chinois... Nous voulons fournir une

framework claire et générale pour permettre non seulement des mesures de performances, mais aussi leur *comparaison*.

D'autre part, nous profitons de ce système de mesures pour permettre à l'utilisateur de représenter les performances de la bibliothèque. D'autres logiciels implantent aussi des moyens de mesurer leurs propres performances (`%timeit` dans Sage, `profiler.c` dans FLINT...) ; nous proposons d'associer les deux objectifs : des *benchmarks* (banc d'essai comparatif) pour les fins de l'utilisateur et pour l'optimiseur automatique.

Ensuite, dans la section 6.2 nous proposons des modèles de conception modulables qui permettent d'assurer des performances tout en gardant le code tout à fait générique.

6.1 Garantir des performances par leur mesure (*benchmarking*).

La création d'un répertoire `benchmark/` répond à des attentes multiples (*cf.* section 6.1.1). Les problèmes qui se posaient alors étaient de :

— s'assurer des performances et de leur reproductibilité ;
— vérifier les performances et détecter les régressions ;
— créer des outils pour mesurer effectivement les performances.

Nous proposons ensuite (section 6.1.2) d'en dériver des outils et de créer des pratiques pour optimiser automatiquement notre bibliothèque en déterminant, à l'installation, les meilleurs choix d'algorithmes sur une architecture donnée.

6.1.1 Des *benchmarks* pour mesurer, tester, paramétrer.

Dans un premier temps, nous décrivons précisément nos motivations et nos objectifs. Ils sont triples : détecter les régressions, fournir un système de mesure de performances à l'utilisateur, créer une infrastructure (*framework*) d'optimisation automatique du code.

6.1.1.1 Tester et cibler les régressions.

Initialement, cette *framework* a été créée pour vérifier les performances de `fgemm`, notamment en fonction des paramètres `alpha` et `beta`, mais aussi en fonction des corps : un `ModularBalanced<T>` n'était pas plus rapide qu'un `Modular<T>` alors qu'on s'attendait au contraire. Aucun test ne permettait de vérifier cela. Il est critique de vérifier que les routines de base sont aussi efficaces qu'attendu et, dans le cas contraire, il n'est pas forcément aisé de détecter, lors de l'exécution d'un algorithme complexe, d'où proviennent certaines pertes de performances. Mesurer les performances en tant que *test* a donc été un point de départ.

Considérons par exemple la création d'une matrice aléatoire sur un certain corps par la classe `RandomMatrix`. Nous disposons d'un algorithme de référence — utilisant p.ex. simplement une fonction `rand()` — et d'une tolérance — disons $\tau = 5\%$. Le test de *benchmark* passera si, à la tolérance près, la classe `RandomMatrix` n'est pas au moins aussi rapide que la routine triviale, sur toute la plage de valeurs. De cette manière, nous sommes parvenus à détecter de mauvaises implémentations de la fonction `init`, qui seraient par ailleurs passées probablement encore longtemps inaperçues.

6.1.1.2 Mesurer les performances.

Ensuite, il découle pour l'utilisateur d'un outil pour représenter graphiquement les performances de tel ou tel algorithme et de les comparer. Par exemple, en fonction de divers paramètres (méthode, domaine...) il est aisé de découvrir et comparer les combinaisons qui produisent les meilleures performances. Les *benchmarks* lui permettent aussi de *comparer automatiquement* les algorithmes de LinBox avec ceux provenant d'autres bibliothèques (IML, NTL...), ou de comparer les performances de son nouvel algorithme en fonction de ce qui se fait déjà dans la bibliothèque.

Aussi, nous rajoutons la contrainte qui demande que la création d'un tel graphique soit *aisée* pour l'utilisateur et réclame un minimum d'investissement. Finalement, proposer un système de mesure de performances permet de s'assurer de la *reproductibilité* des performances mesurées.

6.1.1.3 Établir des paramètres par défaut.

Finalement, nous voulons utiliser cette architecture pour améliorer les performances de la bibliothèque. C'est l'un des défis les plus relevés de ce système de mesure. En effet, une étude exhaustive des performances des divers algorithmes qui constituent les méthodes d'une solution permet de choisir les méthodes par défaut et au besoin de les changer. C'est en particulier utile pour vérifier que les méthodes hybrides ou sélectives par défaut font les bons choix (i.e. les plus efficaces). C'est aussi utile pour découvrir à l'installation les meilleurs paramètres sur un environnement donné, exactement comme `--enable-optimisation` lors de la configuration de FFLAS-FFPACK permet de trouver, pour `fgemm`, le meilleur seuil entre la multiplication rapide par Winograd et celle avec les BLAS.

Ainsi, le développeur trouvera un outil pour tester automatiquement si une modification qu'il a apportée quelque part dans un code entraîne une régression ; l'utilisateur vérifiera que son installation est correcte et aura une vue d'ensemble des performances accessibles sur sa machine.

Nous allons maintenant présenter l'architecture mise en œuvre pour mesurer automatiquement et efficacement les performances.

6.1.2 Une *framework* de *benchmarks* partagée par les utilisateurs et l'optimiseur.

Comme nous l'avons déjà mentionné, un objectif de ce système de *benchmarks* est de pouvoir produire facilement des tableaux de données de qualité et de les exploiter aisément, notamment par des créations de graphes ; il est facilement extensible par l'utilisateur — c'est une forme de généricité.

6.1.2.1 Production de graphiques.

Un de nos buts est donc la production simplifié de graphes de qualité représentant des mesures de performances. Nous allons décrire la conception des classes qui permettent de répondre à cette question.

Abstraction de gnuplot.

Tout d'abord, le logiciel libre gnuplot [1] permet de créer des graphiques de qualité à partir de tableaux de données et d'un langage de script : ce logiciel est donc tout à fait pertinent quant à nos besoins. Ensuite, nous avons voulu permettre à l'utilisateur de pouvoir l'utiliser sans avoir à connaître sa syntaxe ou ses options, simplement à travers une classe–interface en C++ : pour cela une classe `PlotData` (code 6.1) permet d'accumuler les données tandis qu'une classe `PlotStyle` permet de définir aisément le style de graphiques — légendes (abscisses, ordonnées, titres), les styles de tracé, les formats de sortie, les échelles, *etc.* Une classe `PlotGraph` combine les deux pour créer un graphe combinant le style et les données. Ces classes sont bien documentées et elles sont faciles à utiliser pour des besoins courants. Dans une utilisation plus pointue, elles laissent néanmoins la possibilité à l'utilisateur averti de personnaliser à loisir ses graphes.

En résumé, une structure de données permet d'enregistrer des points, une autre permet en quelques appels de fonctions de faire des choix simples et courants de formatage et enfin une structure de données combine et articule ces deux dernières pour créer un graphe.

```
template<class T>
class PlotData {
        vector<vector<double> >  data_y ;
        vector<string>           names_y ;
        vector<T>                data_x ;
```

1 — Disponible à www.gnuplot.info/.

```
public :
        PlotData (size_t n_x, size n_y =1) ;
        void setEntry (size_t i, size_t j, double val) ;
        void setAbsciName (size_t j, T name);
        void setSerieName (string name);
        ...
        double getEntry (size_t i, size_t j) ;
        ...
};
```

Code 6.1 — Structure de données des benchmarks.

Un fichier d'image (eps, png, jpeg, pdf, svg...) est créé. On peut aussi choisir un simple tableau en format texte brut, html ou LaTeX. Nous montrons dans la figure 6.1 un exemple de mesures effectuées sur la routine `fgemm` en caractéristique 13 sur divers corps. Un corps de réels (`UnparametricField`) sert de référence. Nous constatons que pour un petit modulo, l'utilisation des `float` est plus avantageuse que celle des `double` et que les représentations centrées sont un peu plus efficaces que les représentations classiques des corps. Nous constatons que le fait de convertir automatiquement les `int32_t` en `float` permet de rendre l'opération `fgemm` rapide sur ce type de données. Nous pouvons aussi noter que lorsque la caractéristique est petite, la routine `fgemm` aurait généralement intérêt à convertir vers des `float`.

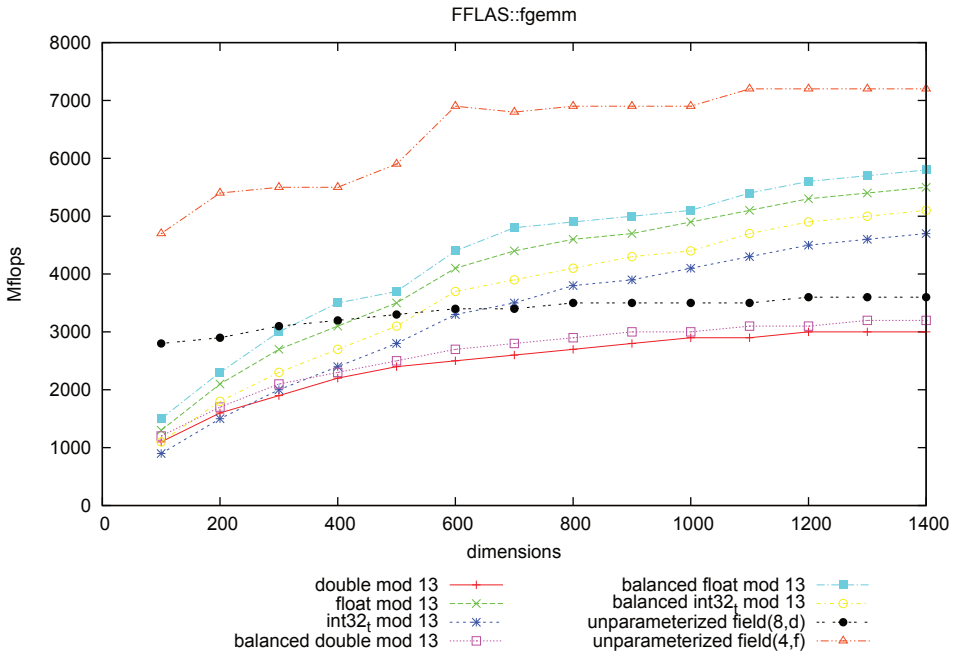

Fig.. 6.1 — Exemple de sortie de `benchmark` sur Perso@home : `fgemm`.

Une nouvelle cible `make benchmarks` permet actuellement de créer un certain nombre de graphes que l'utilisateur peut consulter, comme celui de la figure 6.1.

Adaptabilité des mesures à l'environnement.

Non seulement nous voulons pouvoir produire des graphiques, mais aussi voulons-nous pouvoir les comparer, p.ex. entre deux versions de LinBox différentes. Nous voulons aussi que la création des graphiques dépende de l'environnement. Nous ne pouvons pas demander de tester inconditionnellement par pas de 20 toutes les tailles de matrices entre 1 et 10 000 car certaines machines mettraient des heures à effectuer ces tests tandis que d'autres échoueraient faute de mémoire disponible pour les grandes tailles... Il nous faut donc laisser une certaine *adaptabilité* à la création des mesures.

Nous devons être capable de faire des mesures depuis les petits cas jusqu'aux limites des machines, de manière transparente pour l'utilisateur et sa machine.

Pour cela, il est possible de contrôler précisément le temps passé sur la création de ces *benchmarks*. Chaque itération est répétée un nombre de fois suffisant pour qu'un point corresponde à une durée minimale représentative d'exécution, mais sans dépasser un temps et nombre de répétitions maximaux. Cela permet p.ex. de s'adapter aux machines lentes ou rapides, disposant de plus ou moins de mémoire. Typiquement, un test sur `fgemm` pourra voir testées des matrices plus ou moins grosses selon l'environnement. Une progression est d'autre part affichée sur le terminal, au cas où la création d'un *benchmark* dure plus de temps que prévu, afin de ne pas surprendre l'utilisateur.

6.1.2.2 Optimisation automatique via les mesures de performances.

Nous avons un moyen de fournir à l'utilisateur des réponses à des questions telles « sur quel corps `fgemm` est-il plus rapide ? », « sur des entiers de 1 000 bits, quel est le meilleur algorithme de reste chinois ? »... Nous voulons maintenant que, lorsqu'une bibliothèque que nous développons est configurée avec le paramètre `--enable-optimizations`, ce soit le script `configure` qui réponde à ce genre de questions et que toute la bibliothèque profite de ces optimisations.

Il existe divers types d'optimisation à pratiquer. Tout d'abord, nous trouvons les optimisations de paramètres. Par exemple, l'optimiseur découvre le paramètre `LINBOX_WINO_THRESHOLD_DOUBLE` ou `LINBOX_SOLVE_INTEGER_METH` et dans le code, on retrouve des passages comme dans les codes 6.2 et 6.3.

```
if ( m < LINBOX_WINO_THRESHOLD_DOUBLE )
      mul_blas(...);
else
      mul_wino(...);
```

Code 6.2 — Paramètre par défaut (multiplication).

```
template<class Vector, class Matrix, class Method>
Vector & solve(Vector &x, const Matrix &M, const Vector &b,
               const RingCategories::IntegerTag & tag,
               const Method & M = LINBOX_SOLVE_INTEGER_METH)
```

Code 6.3 — Paramètre par défaut (systèmes entiers).

Ensuite, il existe les optimisations des fonctions simples ou de base dont il peut exister plusieurs versions. Nous présentons dans le code 6.4 un exemple d'implantation. La méthode utilisée est la suivante. Le script `configure` découvre les paramètres de l'environnement et les définit dans un fichier `linbox-config.h`. Les choix par défaut (avant l'optimisation) se trouvent dans un fichier `linbox-configuration.h`. Ce fichier remplace exactement `linbox-config.h`, qui devait être inclus dans tout fichier de LinBox. Ce fichier `linbox-configuration.h` inclut lui-même un fichier `linbox-optimise.h` qui est créé par le script `configure`. Si les optimisations sont activées, les définitions de macros dans ce fichier prévalent sur les défauts de `linbox-configuration.h`. Finalement, le fichier `linbox-optimise.h` est créé incrémentalement, de sorte que l'optimisation n° k bénéficie des paramètres découverts lors des optimisations n° $1, \ldots, k-1$. Ainsi, nous parvenons à créer proprement et lors de la configuration un système de configuration par défaut qui s'adapte à l'environnement.

```
// file : linbox/linbox-configuration.h
#include "linbox/linbox-config.h"
#include "linbox/linbox-optimise.h"
#ifndef LINBOX_MOD_DOUBLE
#define LINBOX_MOD_DOUBLE((a),(p)) fmod((double)a,(double)p)
```

```
#endif

// file : linbox/field/Modular/modular-double.h
template<>
class Modular<double> {
    ...
    Element &init(Element &x, Element&y) const
    {
        x = LINBOX_MOD_DOUBLE(x,y) ;
        ...
     }
     ...
};
```

Code 6.4 — Fonction par défaut (fmod).

Finalement, et c'est certainement le plus difficile, il existe les optimisations qui dépendent de fonctions compliquées ou de divers paramètres. Par exemple, selon la taille des entrées en argument, certains algorithmes pour résoudre des systèmes linéaires sur les entiers se comportent très différemment (*cf.* section 5.2.4, §2). Dans ce cas particulier, il faut découvrir ces combinaisons : pour un certain nombre d'algorithmes testés en fonction de certains paramètres (ici tous les algorithmes de résolution de systèmes sur les entiers en fonction de la taille maximale des coefficients de la matrice) l'optimiseur découvre à partir des tableaux de performances les algorithmes les plus rapides et les seuils associés.

```
// file : linbox/linbox-optimise.h (autogenerated)
#define LINBOX_SOLVE_INTEGER_THRES0 45
#define LINBOX_SOLVE_INTEGER_THRES1 78
#define LINBOX_SOLVE_INTEGER_ALG0 solveMethod::WanTrait
#define LINBOX_SOLVE_INTEGER_ALG1 solveMethod::Dixon
#define LINBOX_SOLVE_INTEGER_ALG3 solveMethod::IML
#define LINBOX_SOLVE_INTEGER (x,A,b,l) {
if (l< LINBOX_SOLVE_INTEGER_THRES0) {
        solve(x,A,b,LINBOX_SOLVE_INTEGER_ALG0());
}
else if (l < LINBOX_SOLVE_INTEGER_THRES1) {
        solve(x,A,b,LINBOX_SOLVE_INTEGER_ALG1());
}
else {
        solve(x,A,b,LINBOX_SOLVE_INTEGER_ALG2());
}
```

Code 6.5 — Seuils et fonctions par défaut.

6.1.3 Prolongements.

Comme cette architecture de *benchmark* est toute récente, il existe encore diverses pistes à explorer. Nous citons quelques futurs axes de travail.

- Idéalement, tous les *algorithmes* et les *solutions* devraient être présents dans ce répertoire. Ce sera encore le fruit de longs efforts.
- Cette architecture de *benchmark* pourrait être portée dans Givaro ou dans FFLAS-FFPACK.
- Nous nous sommes axés sur les performances liées au temps (les unités de l'axe des y sont essentiellement les secondes, les mffops...) mais pas à l'espace. Par exemple l'outil `massif` de valgrind permettrait de vérifier automatiquement l'utilisation mémoire de certains algorithmes et d'en tirer des graphes.

✦ L'implantation de `--enable-optimization` lors de la configuration pour choisir et adapter des seuils, des morceaux de codes ou des choix d'algorithmes dont l'optimisation dépend de l'architecture n'est pas complète à ce jour...

Nous avons découvert *a posteriori* la bibliothèque *template* C++ eigen [2] d'algèbre linéaire numérique qui propose deux systèmes de *benchmarks*, l'outil final pour la création de graphes étant aussi gnuplot :

— L'un est à base de scripts non génériques récoltant les résultats d'exécutables. Cela permet en particulier de pouvoir mesurer des performances entre divers compilateurs ou avec diverses options de compilation. Le seul moyen pour y parvenir dans notre système de *benchmark* est de remplir le fichier `.dat` avec divers exécutables (par exemple compilés par divers compilateurs) et de construire ensuite nos structures de données à partir des valeurs dans ce fichier (avec un constructeur `PlotData(const std::string & datafile)`).

— L'autre système est dérivé d'une version significativement modifiée de BTL [3] ([PH08]). Il est intéressant de noter que la bibliothèque BTL est construite pour être générique à travers un mécanisme d'encapsulation de fonctions (actions) pour diverses implantations (noyaux) qui permet de mesurer les performances de ces diverses implantations et même de les tester automatiquement par rapport à une routine supposée sûre.

Dans la bibliothèque eigen, ces *benchmarks* sont purement destinés à des fins de mesures et de tests, et non destinés à l'auto-optimisation. Cependant, la bibliothèque BTL prévoit comme nous — mais elle ne la distribue pas à notre connaissance — une synthèse automatique des meilleurs algorithmes pour une fonction donnée. Leur problème est cependant plus général que le nôtre car ils prévoient par exemple de changer de vendeur de BLAS (implantations différentes mais interfaces identiques) au sein d'une même routine, d'où des problèmes de lien (utilisation de `dlopen` à la place). Ceci n'est ni prévu par LinBox ni démontré en pratique dans BTL. Bien que le code `bench<timer_t, action_t< implem_t > > > puisse être considérée comme très générique car laissant à l'utilisateur la flexibilité d'écrire sa propre classe `action_t` pour des implantations données, nous pensons donner plus de libertés à l'utilisateur en lui permettant d'implanter lui-même sa fonction. Cela permet par exemple la liberté du constructeur de données aléatoires (avec ou sans certaines propriétés, depuis un fichier...), de pouvoir mesurer les performances de deux implantations différentes d'une même fonction dans une même bibliothèque... En outre, nous offrons la liberté de créer via une interface simple, en C++, et dans le même fichier, des graphiques correspondant aux données (liberté étant laissée à l'utilisateur de retravailler son fichier `.dat`) : contrairement à BTL nous fournissons via un effort minimum la possibilité de formater son graphe de sortie sans avoir à relancer un script non paramétrable bash après la création du fichier brut de données `.dat`.

Le projet BTL n'est donc pas directement utilisable dans LinBox mais cet exemple de bibliothèque de *benchmarks* générique cautionne [4] nos objectifs de cette section ; les idées se rejoignent : prévoir un système de mesures proposant à la fois une production de graphes et de code (hybride) optimal, tout en étant « générique [5] ». Aussi, la bibliothèque BTL met en avant, à juste titre, l'importance du choix du chronomètre et permet de *templéter* ses *benchmarks* par un *timer*, c'est une piste que nous devrions aussi suivre. C'est typiquement le genre de mécanisme qui permet aussi une meilleure portabilité entre différents systèmes d'exploitation.

6.2 Amélioration des performances : conception par briques de base.

Selon l'adage de Donald Knuth, « on devrait oublier les petites optimisations locales, disons, 97% du temps : l'optimisation prématurée est la source de tous les maux ». Nous n'allons pas chercher à optimiser

2 — Disponible sur `http://eigen.tuxfamily.org`.

3 — Benchmark Template Library `http://projects.opencascade.org/btl/`.

4 — Il n'est pas fréquent dans la littérature ou dans la vie des logiciels de rencontrer ce genre de problèmes et d'y apporter des solutions.

5 — en fait les actions proposées par BTL ne sont pas si génériques et contrairement à notre système de *benchmark* moins flexibles quant aux possibilités de représentations et de comparaisons de données.

le code de nos bibliothèques tel quel. Au lieu de cela, nous cherchons à rendre le code modulable, bâti à partir de briques de bases (*building block*) performantes. Nous démontrons pourquoi dans cette section.

Le code de LinBox étant essentiellement *templété*, lors de l'inclusion d'un entête, le compilateur voit tout le code. C'est un atout pour le compilateur, mais un désavantage si la taille de l'exécutable explose. Par exemple, la bibliothèque Givaro étant très peu *templétée*, nous avons restructuré de nombreux fichiers pour permettre [6] d'*inliner* un maximum de code afin que le compilateur puisse effectuer de meilleurs choix. Ceci se fait au détriment du temps de compilation — et de l'utilisateur — car une majorité du code de Givaro est compilable. Nous laissons à l'utilisateur le choix, à la configuration, entre code pur et code pré-compilé. Cette question de généricité et de performance a soulevé de nombreuses études. Nous citons en particulier [UEKM96] qui étudient la possibilité de fournir un code générique tout en proposant dans son source divers types de polymorphisme ; [TWBS09] qui proposent des techniques et une surcharge au langage C++ afin de minimiser les dépendances entre les membres d'une classe générique et avec ses paramètres *template* pour alors diminuer la taille des exécutables. Nous notons que nous pouvons aussi utiliser, pour les combinaisons les plus communes, un système de compilation séparée ([DGPS10]).

6.2.1 Optimisations, micro-optimisations : défi ou temps perdu ?

Lorsque ces modules arrivent à un bas niveau, le compilateur saura généralement fournir un code performant. Néanmoins, nous devons garder un œil dessus (le profiler, tester son efficacité) pour s'assurer des meilleurs performances à tous les niveaux. Une autre option utilise des routines spécifiques, optimisées et sûres provenant d'autres bibliothèques — p.ex. BLAS, NTL, IML... Une dernière option consiste à créer dans LinBox du code optimisé (*cf.* SpMV, partie II).

Laisser les optimisations au compilateur.

Bien entendu, nous laisserons généralement au compilateur la lourde tâche d'optimiser le code de bas niveau et éviterons les optimisations spécifiques, non toujours portables, comme les jeux d'instructions SSE ou MMX par exemple. Nous laisserons aussi le choix au compilateur de dérouler ou d'optimiser les boucles — dans les cas où cela est possible, on préférera cependant prendre l'habitude de la pré-incrémentation histoire de ne pas créer de temporaires inutiles, surtout lorsque cette opération est coûteuse. On cherchera au maximum à guider le compilateur en favorisant les choix effectués à la compilation (utilisation de traits) ou en spécifiant le mot clef `const` par exemple — plus un compilateur a d'informations, plus il pourra effectuer d'optimisations agressives.

Voici par exemple comment nous pouvons essayer d'optimiser une addition. Cette opération est en général lente car la réutilisation des données est nulle [7] et elles sont transportées seulement à la vitesse du bus... Une simple boucle `for` bien optimisée fournira donc de bonnes performances. Nous avons cependant essayé d'utiliser des routines SSE (code 6.6).

```
void add_sse(double *c, const double *a, const double *b, size_t n)
{
        __m128d *av, *bv, *cv;
        av = (__m128d*)a; // alignés
        bv = (__m128d*)b;
        cv = (__m128d*)c;
        size_t i = 0 ;
        for ( ; i < n/2  ; ++i)
                cv[i] = _mm_add_pd(av[i], bv[i]);
        if (n&1)
                c[n-1] = a[n-1] + b[n-1];
}
```

6 — Ces codes peuvent néanmoins aussi se retrouver pré-compilés dans la bibliothèque, c'est une option de configuration.
7 — en effet, l'opération add n'est pas dans les spécifications BLAS...

Code 6.6 — Utilisation d'instructions SSE pour additionner deux vecteurs.

Il est très difficile de mesurer précisément la rapidité de cette routine pour les n petits. Pour des tests où l'on répète n fois cette routine sur des vecteurs de taille n, (typiquement une sous-matrice dont les pointeurs sont alignés), l'accélération obtenue se retrouve généralement entre 1,2 et 2. Par contre, lorsque les pointeurs ne sont pas alignés (sur 16 bits), la version non alignée de ce code (utilisant alors `_mm_loadu_pd` et `_mm_storeu_pd`) est généralement plus lente qu'une boule naïve optimisée par le compilateur. Il faut donc connaître l'alignement d'un pointeur à l'entrée dans cette fonction (sous peine de *segfault* ou de pertes de performances), ce que ne prévoit pas FFLAS-FFPACK... Bref, cette optimisation en particulier n'apporte rien, si ce n'est, éventuellement, des bogues. Nous garderons par contre les fonctions add, sub et leurs dérivées qui apportent plus de sens et de clarté à la lecture d'un code que des doubles boucles (*cf.* section suivante) et autorisent éventuellement les optimisations locales — p.ex. une addition sur des mots compressés, sur $\mathbb{Z}/2\mathbb{Z}$.

L'opposé de cette approche consiste à fournir un code non nécessairement optimisé avec lequel le compilateur fera du mieux qu'il peut. Nous proposons une conception différente, à la fois générique et performante qui permet de laisser le choix entre l'utilisation d'un code hyper-optimisé ou d'un code de base, tout en laissant la liberté d'une optimisation « entre les deux extrêmes ».

6.2.2 Conception générique par *building blocks* (briques modulables).

Nous proposons dans cette sous-section une approche de conception de la bibliothèque par *briques de base*. Nous voyons dans la généralisation de cette approche divers avantages. Cerner des briques de base, c'est décrire des unités dans le code et leur donner sens. C'est aussi permettre d'interchanger ces briques, ou de pouvoir les optimiser une à une, séparément. C'est surtout ne pas perdre en performances tout en fournissant un code qui reste générique et modulable. Nous donnons des exemples à divers niveaux : au niveau de concepts, au niveau sémantique, au niveau des fonctions.

Exemples d'optimisations des corps.

Les opérations sur les corps sont des opérations de base, souvent atomiques, et donc leurs performances doivent être optimisées. Un corps est un des *building block* essentiel dans notre bibliothèque, comme tout concept paramètre *template* d'une majorité de fonctions le serait dans une bibliothèque générique quelconque.

Par exemple, pour des soucis d'efficacité, nous avons rajouté dans l'interface de chaque anneau les éléments `zero`, `one` et `mOne`, car ces éléments existent toujours et sont souvent utilisés, donc inutiles à re-créer [8]. Nous avons donc extrait une opération parfois coûteuse (`Element one ; F.init(one, 1) ;`) en une opération optimale et pleine de sens (`F.one`).

Dans le but d'optimiser les opérations sur les corps, nous avons fait divers tests, profilé diverses opérations. Notamment, nous avons implanté une version de `Modular` et `ModularBalanced` qui compte elle-même ses opérations (nombre d'additions, de multiplications, de *cast*, de divisions...). Cela nous procure des informations plus précises qu'un profilage de code avec `-pg`. Nous en avons tiré de nombreux enseignements et petites optimisations, par exemple que l'opération `fmod` peut être moins rapide qu'espéré :

```
// x = fmod (double(y), modulus);    // lent
x = (double)((int64_t)y%lmodulus);   // rapide
```

Code 6.7 — Opérateur réduction modulaire `%` ou `fmod`.

8 — Cela impose cependant au corps de savoir construire ses éléments : en particulier, cela nous a conduit à séparer du corps `UnparametricField` ses opérations et initialisations (p.ex. `UnparametricOperations`). En effet, un anneau qui hériterait de `UnparametricField<K>` ne saurait pas forcément construire son élément `zero`...

Tab. 6.1 — Comparaison des opérations axpy et `.+=.*.` (µs) sur Joran@imag

Corps	axpy	a+=b*c
`Givaro::integer`	0.039	0.17
`LinBox::integer`	0.039	0.19
`mpz_class`	0.033	0.18
`mpz_t`	0.027	—

Nous avions pensé que décentrer la représentation d'un `ModularBalanced` vers des valeurs positives permettrait de trouver des compromis entre taille maximale de l'anneau et vitesse. La classe `ModularCrooked` l'implémente et montre bien un compromis.

Dans le code 6.8, nous donnons un exemple d'utilisation cruciale de brique atomique qui rend le code significativement plus rapide.

```
// test-integer-speed.C
Integer a,b,c ;
...
/* lent : a += b*c ; */
Integer::axpyin(a,b,c) ; // rapide
```

Code 6.8 — Utilisation des blocs génériques *vs.* les opérateurs.

En effet, comme Givaro ne fait pas d'évaluation paresseuse et de surcharge de certains opérateurs, le code `axpyin` qui combine l'opération '+' et'*' sera plus rapide que les deux opérations successives sur la ligne commentée. Dans la même veine, nous avons implanté les opérations `axmy` et `maxpy` et leur variante en place (suffixée de `in`) — et propagé cette notation à d'autres domaines, notamment matriciels.

En général, nous avons donc essayé d'envelopper (*wrapper*) les fonctions spécialisées de gmp pour notre usage, et de faire une veille des nouvelles fonctionnalités, ajouter de nombreuses fonctions (résidus, opérations combinées, nombres aléatoires, nouvelle classe `Rationnel`[9]...

Classes ou noms de d'espaces ?

L'utilisation de classes ou structures avec constructeur (implicite) trivial et fonctions statiques ou de noms d'espace permet de regrouper du code sous une même autorité, lui donnant ainsi une unité et un sens. Nous avons choisi de placer Givaro, FFLAS, FFPACK, ainsi que les librairies externes IML ou fplll dans des noms d'espace plutôt que d'utiliser des classes. La raison en est d'abord structurelle : il est plus agréable de travailler dans un nom d'espace car il y a moins de contraintes de programmation. Par exemple, il est très facile de rajouter une fonction ou une classe à un nom d'espace en dehors de là où il a été premièrement défini, alors que c'est impossible pour une classe, où toutes les fonctionnalités doivent être groupées ensemble — on évite ainsi en particulier les `#ifndef`. Il est généralement agréable d'utiliser des noms d'espace pour séparer clairement les fonctionnalités et les attributs de chaque identité de code. La raison provient aussi des performances : il n'est pas moins efficace[10] d'utiliser des noms d'espace.

Fonctions spécialisées.

Prenons quelques nouvelles fonctions apportées à FFLAS-FFPACK : les fonctions `fzero`, `fcopy`, `fmove`, `fadd`, *etc.* Ces fonctions ne sont pas bien difficiles à implanter, généralement l'affaire d'une simple

9 — Nom temporaire en attendant de supprimer l'ancienne classe `Rational` qui tirait mal parti de gmp.

10 — En fait, depuis gcc-4.5, une fonction provenant d'un nom d'espace est autant optimisée qu'un membre statique dans une structure.

Tab. 6.2 — Optimisations de applyP : calcul de $P\left(P^{-1}AP\right)P^{-1}$ (secondes).

dimension	2 000	5 000	10 000
FFLAS-FFPACK-1.3.3	0,18	1,24	6,6
FFLAS-FFPACK-1.5.0	0,08	0,54	2,3

Tab. 6.3 — Optimisations de fgemm : calcul de $C = 0AB + \beta C$ (secondes).

dimension	2 000	5 000	8 000
FFLAS-FFPACK-1.3.3 ($\beta = 1$)	0,03	0,20	0,56
FFLAS-FFPACK-1.3.3 ($\beta \neq \pm 1$)	0,03	0,21	0,55
FFLAS-FFPACK-1.5.0 ($\beta = 1$)	0,00	0,00	0,00
FFLAS-FFPACK-1.5.0 ($\beta \neq \pm 1$)	0,03	0,22	0,56

boucle, voire double. Pourtant, il y a de réels bienfaits à cela. D'une part, extraire les fonctions réutilisables et les nommer de manière appropriée permet de faciliter la compréhension et la réutilisation du code, de diminuer sa taille, facilite parfois le ciblage des problèmes lors de débogage ou de profilage. D'autre part, extraire des fonctionnalités d'un code permettra de s'attaquer de manière « atomique » à ses performances et de le rendre modulable. Bien souvent, la routine par défaut utilisée par fcopy sur un vecteur sera une simple boucle ou un std::copy. Cependant, sur des doubles, nous pouvons aussi utiliser la fonction standard memcpy, voire une routine BLAS ! Nous rendons notre fonction spécialisée plus générique et plus performante.

C'est ce style de programmation que nous essayons de mettre en œuvre. Étudions par exemple dans la bibliothèque FFLAS-FFPACK quelques routines : applyP et fscal. Dans la première nous constatons de nombreux défauts de cache dus à la fonction fswap : nous écrivons donc deux sous-fonctions selon que l'on permute les lignes ou les colonnes, le parcours des éléments est alors différent. Nous constatons dans le tableau 6.2 le bienfondé de ce choix.

Dans le tableau 6.3 suivant, nous avons réécrit la fonction fscal (code 6.9) de manière à la généraliser aux matrices et à traiter les cas particuliers via des retours rapides. Cette façon de faire permet de ne pas perdre de performances dans le cas général (à quelques branchements près mais nous voyons qu'ils n'interviennent pas dans les temps mesurés) tout en bénéficiant des cas particuliers. En outre, la partie de code de fgemm l'utilisant devient plus claire. Lorsque le paramètre α dans fgemm vaut 0, l'opération se réduit alors à $C = \beta C$, soit une opération fscal, nous comparons donc les temps mis par la routine fgemm avec le paramètre β à 1, mais aussi avec un paramètre β non trivial. Nous avons un gain de performances maximal pour $\beta = 1$ et pas de perte de performances dans le cas général, ce qui valide notre conception du bloc fscal.

```
template<class Field>
void
fscal (const Field& F, const size_t m , const size_t n,
       const typename Field::Element alpha,
       typename Field::Element * A, const size_t lda)
{
        typedef typename Field::Element Element ;

        if (F.isOne(alpha))
                return ;
        else {
                if (lda ==  n)
                        fscal(F, n*m, alpha, A,1);
```

```
                else {
15                      for (size_t i = 0 ; i < m ; ++i)
                                fscal(F, n, alpha, A+i*lda,1);
                }
                return;
        }
20 }
```

Code 6.9 — Optimisation de `fscal` : retours rapides.

6.2.3 Conclusion.

Nous avons montré que la construction par blocs à tous les niveaux ne sont pas de « petites optimisations locales », elles sont totalement nécessaires dès la conception de l'algorithme, assurant à la fois généricité et efficacité. La conception par blocs permet aussi de mieux organiser le code — en regroupant certaines fonctionnalités (classes et noms d'espace), et en extrayant d'autres (fonctions, membres de classes) — tout en facilitant sa compréhension. C'est cependant souvent avec du recul, avec de nouveaux besoins, après de nombreuses « refactorisations » de code ou après un profilage attentif que l'on arrive à bien cerner et extraire les briques de base les plus essentielles. Le contrainte que nous essayons de décrire ici est d'éviter une conception du code « à la demande » et de la remplacer par une conception du code « par brique » qui seront réutilisables, interchangeables et optimisables. C'est d'ailleurs tout à fait le style de codage que nous prévoyons pour intégrer un parallélisme générique dans LinBox-2.0 où cette conception par *building block* conduira peut-être à un langage de programmation *ad hoc*.

"Any code of your own that you haven't looked at for six or more months might as well have been written by someone else."

Eagleson's Law

Chap. 7

Amélioration de la qualité du code.

Sommaire

7.1 Comment documenter une bibliothèque générique ? 114
7.1.1 Annoter proprement et simplement le code. 114
7.1.2 Documenter simultanément et efficacement la bibliothèque pour l'utilisateur et le développeur. 115
7.1.3 Prolongements. 116
7.2 Rendre le code plus sûr. 117
7.2.1 Tester et vérifier une bibliothèque générique. 117
7.2.2 Normalisation du code : robustesse et pérennisation. 119
7.3 Conclusion. 125

La qualité d'un code est une notion subjective. Elle est cependant certainement liée à la qualité du logiciel développé. Un code bien structuré, bien écrit, bien documenté est un code plus facile à maintenir, à faire évoluer, plus facile à pérenniser et à déboguer. Cependant, la tentation est souvent forte de rajouter un petit hack deci-delà, oublier de commenter sa nouvelle classe, ne pas propager complètement certaines modifications, remettre à plus tard l'écriture des tests... Dans ce chapitre, nous allons proposer des moyens pour rendre des codes hétérogènement écrits plus homogènes et plus faciles d'accès.

En effet, prenons la version 1.1.7 de LinBox ; nous constatons une diversité dans la qualité de certains fichiers. Nous pouvons expliquer cela par la variété des développeurs de LinBox, le maintien inégal de leurs contributions... Par exemple, compilons les tests de cette version de LinBox avec `CXX=g++-4.4` d'une part et le compilateur `CXX=icpc` d'autre part, avec les options de compilation par défaut. Dans le premier cas, nous obtenons 370 lignes d'avertissements tandis que dans le second, nous en sommes à 486 000 ! Une leçon à tirer impose de forcer la compilation avec le plus possible de niveaux d'avertissements de manière à simplement ne pas gêner l'utilisateur [1].

Afin de s'assurer de l'homogénéité d'un code, il existe la solution de le soumettre à une revue. Cette revue peut être communautaire, effectuée par des pairs, comme p.ex. dans Sage ou effectuée par quelques personnes en charge du code, par des chefs de projets (*leaders*). Le nombre de développeurs de LinBox et des projets associés est trop petit pour que ces solutions aient un sens. Au lieu de cela, nous préférons avoir un accès libre et responsable aux modifications du source. Il nous faut donc

1 — Imaginons en plus que parmi ces Mo de messages d'avertissement l'utilisateur cherche la source d'un véritable message d'erreur !

superviser le code produit, établir des règles, pouvoir les vérifier et les corriger — automatiquement dans la mesure du possible.

Comment pouvons-nous assurer, vérifier et maintenir une certaine qualité de code ? Nous proposons d'abord (section 7.1) des méthodes pour documenter efficacement un code générique puis dans la section 7.2 nous mettons en œuvre différentes techniques afin de proposer un code plus stable, plus robuste.

7.1 Comment documenter une bibliothèque générique ?

Pour la continuité, la maintenance ou l'utilisation d'un logiciel, une documentation claire, précise, facilement disponible et utilisable est indispensable. Un premier problème que nous rencontrons pour documenter une bibliothèque générique consiste à présenter et décrire les concepts ainsi que leurs relations avec le code et les algorithmes. En effet, LinBox étant une bibliothèque de mathématiques qui utilise des techniques de programmation et de conception non triviales, il est facile de s'y perdre sans un bon fil rouge — de bonnes explications, documentations, références.

Nous avons choisi de proposer une documentation située à deux niveaux : au niveau du code lui-même pour le rendre clair et lisible (section 7.1.1), mais aussi au niveau des fonctionnalités, de sa structuration (section 7.1.2). Il s'avère aussi qu'un utilisateur et un développeur ne réclament pas le même niveau de documentation : là aussi nous fournissons des niveaux de documentations adéquats.

7.1.1 Annoter proprement et simplement le code.

Il n'est dans cette sous-section rien de très novateur : nous proposons simplement des techniques pour clarifier et structurer la documentation interne du code. En effet, le mot *commenter* peut avoir une signification ambiguë. Nous utilisons ce verbe dans le sens d'annoter, expliquer, décrire. Commenter un code est en quelque sorte une documentation interne du code, dédiée seulement à sa compréhension interne, non à l'explication des fonctionnalités. Nous préférons aussi le terme *désactiver* dans le sens de cacher un morceau de code — que l'on a pas envie de voire disparaître pour diverses raisons, parfois affectives souvent paresseuses. N'oublions pas enfin que le première façon de commenter un code, c'est de l'écrire avec des classes aux noms évocateurs, des noms de fonctions et de variables bien choisis...

Styles proposés d'annotation.

Commenter un code constitue une tâche difficile à effectuer *a posteriori* si elle n'a pas été proprement faite en premier lieu. Globalement, lors de la relecture de code, un effort a été fait pour mettre les commentaires importants en multi-lignes (`/* comment */`), résistants à l'insertion et au saut de ligne et les commentaires très précis ou locaux en fin de ligne (`// comment`). La désactivation de code, quant à elle, est effectuée via le pré-processeur :

```
#if 0 /* pour quoi ce code est inconditionnellement désactivé */
        dead_code();
#endif
```

Code 7.1 — Désactivation de code par pré-processeur.

Un avantage réside dans le fait que tout code (même du code imbriquant des commentaires) peut être exclu avec quelques frappes au clavier, sans rien changer par ailleurs. Certains éditeurs de texte (p.ex. vim) comprennent cette désactivation et permettent un repli de code et sa coloration syntaxique. Un désavantage réside néanmoins dans le fait que certains éditeurs ne le prennent pas en compte et alors la lecture du code est *bien* moins aisée qu'en rajoutant devant chaque ligne un commentaire style C++ (`//`). Nous remarquons que la désactivation permet, avec un simple `grep`, de retrouver le code mort pour l'auditer et le supprimer totalement ensuite.

Supervision et audit.

Un effort certain a été effectué pour simplifier et nettoyer des commentaires du code. Cependant, afin de faciliter la maintenance des divers codes écrits par diverses personnes, qui ne sont plus forcément toujours actives, ce travail de *supervision* est totalement nécessaire — un code écrit est toujours destiné un jour ou l'autre à être maintenu par une autre personne. Il n'y a pas de mesure que l'on puisse apporter à ce travail, ni de mesure qui permette de quantifier la qualité du commentaire d'un code. La relecture d'un code par une autre personne, un travail de *revue* [2] permet cependant de valider ou non le travail d'annotation : s'il y a des problèmes de compréhension, c'est alors souvent au relecteur (supervision) d'apporter les commentaires pertinents manquants.

Nous concluons simplement ce paragraphe par la remarque qu'un code bien structuré, clair, lisible et annoté est le premier pas vers un code sain, facile à déboguer et à faire évoluer : c'est une marque de qualité.

7.1.2 Documenter simultanément et efficacement la bibliothèque pour l'utilisateur et le développeur.

Nous exposons dans cette sous-section le modèle de documentation que nous avons utilisé pour LinBox et introduit dans Givaro et FFLAS-FFPACK. Nous proposons une méthode pour discerner aisément deux niveaux de documentation : celle adressée à l'utilisateur général et celle adressée à l'utilisateur averti ou au développeur. Nous montrons aussi comment nous parvenons à documenter la structure du projet, ses fonctionnalités, ses objectifs et les concepts développés et utilisés.

7.1.2.1 Le logiciel doxygen pour documenter.

Nous ne voulons pas d'une documentation caricaturale qui renvoie par exemple à une bibliographie ou n'explicite pas les inter-connections au sein de la bibliothèque. Au contraire, le choix avait été fait d'utiliser un véritable outil de documentation, doxygen [3] et de s'en servir pour structurer la documentation. Ce logiciel est utilisé par de nombreux projets (Qt, KDE, Adobe Open Source, GNU STL...) pour leur documentation et permet, grâce à une syntaxe assez aisée, de documenter facilement et efficacement des codes mêmes complexes. Le résultat peut en particulier être un document `html`. Il n'est aussi pas très difficile de personnaliser son rendu.

7.1.2.2 Que documenter ?

Une difficulté lors de la documentation d'un code provenant d'en-têtes par rapport à des fonctions dans une bibliothèque ou une interface est la notion d'accessibilité et de visibilité de certaines fonctions aux utilisateurs. Pour cela, nous avons créé une documentation *orientée utilisateur* et une autre *orientée développeur*. Il est facile de basculer entre ces deux arborescences (utilisateur et développeur) car elles sont placées côte à côte sur la page principale de la documentation et sont similairement construites (à un suffixe _dev pour le chemin et à des conventions de couleur près).

Le mot clef `@internal` permet simplement de rendre un paragraphe de documentation visible seulement pour le développeur et ainsi de différentier les deux documentations. L'idée est de ne fournir que l'essentiel des fonctionnalités à l'utilisateur et de tout décrire au développeur. L'un cherchera une approche synthétique, l'autre une approche en profondeur, dans les détails. Principalement, les algorithmes, les solutions, les fonctions et classes non spécialisées sont destinées à la documentation utilisateur. Les fonctionnalités internes (par exemple tout ce qui réside dans le nom d'espace `Protected::` ou certaines spécialisations de fonctions ou membres obscurs de classes) sont quant à elles seulement documentées dans la partie développeur.

Nous créons ainsi, grâce à de simples fichiers « `makefile` pour doxygen », une documentation `html` interactive et automatiquement structurée qui permet à l'utilisateur débutant comme à l'utilisateur

2 — ou plutôt *review* : rapport, critique, audit.
3 — Disponible à l'adresse `http://www.stack.nl/~dimitri/doxygen/`.

avancé de s'y retrouver facilement. Le développeur de la bibliothèque trouve quant à lui un travail réduit à effectuer — ne pas oublier `@internal` quand la description qu'il apporte devient trop technique ou pointue.

7.1.2.3 Comment documenter efficacement ?

Nous cherchons déjà à produire une documentation qui permette une vision synthétique et efficace des fonctionnalités de notre bibliothèque. Pour cela doxygen fournit automatiquement des documentations arborescentes et thématiques (classées par structure, par espace de noms, par fichier, par répertoire, par liste d'exemples) qui permettent d'atteindre rapidement les documents ciblés et de naviguer entre eux. Faut-il encore pouvoir les ordonner, les regrouper et les interconnecter clairement.

Un des problèmes de doxygen est qu'il ne regroupe pas par défaut la documentation des fonctions « humainement ou visiblement proches ». Afin d'éviter trop de répétitions et de lourdeurs, les fonctions qui partagent un même nom et des signatures égales au type prêt peuvent avoir une documentation factorisée en utilisant par exemple les délimiteurs `//@{` et `//@}`. Il est aussi possible de grouper de la même manière des ensembles de fonctions par « thèmes » afin de clarifier et simplifier la documentation et son rendu.

Documentation automatique par modules.

Dans cette idée de classements par thème, il existe les groupes — modules — que l'on peut organiser selon une arborescence. Ceci se fait avec les commandes `@defgroup`, `@ingroup`. Par exemple, `algorithms.doxy` définit le groupe `algorithm` qui appartient au groupe `linbox` et explique ce que sont les algorithmes dans LinBox. Au dessous, il y a p.ex. le groupe `CRA` auquel appartiennent toutes les classes de restes chinois et qui explique comment faire des restes chinois avec LinBox. Chaque groupe possède un texte explicatif. Les classes concernées apparaissent dans ce groupe. Cela crée une vue d'ensemble et arborescente des fonctionnalités de la bibliothèque tout en expliquant les concepts (que sont les algorithmes, quels restes chinois avons-nous implantés, qu'est-ce qui les différencie ?).

Le développeur peut aussi créer lui-même des pages arborescentes (`@page`, `@subpage`) découpées en sections (`@section`). C'est ainsi que nous avons regroupé sur la page d'accueil (`@mainpage`) les tutoriels, explications d'installation, *etc.* auparavant écrits en html pur et donc moins facilement maintenables et moins bien intégrés visuellement que dans une approche de style `css` « tout doxygen ». Notamment, nous pouvons bénéficier gratuitement des colorations syntaxiques des extraits de code, *etc.* D'autre part, des listes et pages orphelines sont regroupées dans un onglet *pages liées*. Nous y avons entre autres introduits la liste des bogues, des codes dépréciés, des choses à faire, des fichiers de tests et des références bibliographiques, qui sont autant de *points d'entrée* différents dans la bibliothèque.

Les mots clefs `@todo` (choses à faire) et `@bug` (bogues et problèmes) `@deprecated` permettent de bien cibler certains problèmes. Bien qu'apparue *a posteriori* et timidement dans la version `1.7.5` de doxygen, nous avions créé une commande `@cite` afin de pouvoir ajouter à la documentation une bibliographie (référencée par ailleurs dans une page spéciale) permettant de citer les articles clefs, la littérature de base, *etc.*

7.1.3 Prolongements.

De manière plus pragmatique, il s'avère que commenter et maintenir une documentation sont des tâches souvent pénibles et fastidieuses — et que l'on procrastine aisément. Cette documentation par thèmes peut s'avérer initialement laborieuse et nécessite une véritable vue d'ensemble de la bibliothèque — dans le cas de LinBox, une connaissance des algorithmes implantés, des mathématiques associées mais aussi des techniques de programmation et concepts associés. Le plus difficile est de la garder à jour : s'il est facile de rajouter un algorithme dans un fichier, l'expérience montre qu'il est beaucoup plus difficile de décrire clairement et précisément ses fonctionnalités et utilisations ou de le replacer dans un contexte général dans le fichier `*.doxy` qui définit un groupe auquel il appartient.

Cette *framework* a été reproduite pour les projets Givaro et FFLAS-FFPACK. Comme la documentation de ces projets est très lacunaire, il reste de gros efforts à effectuer pour la rendre complète.

7.2 Rendre le code plus sûr.

Lors de la distribution d'une bibliothèque de calcul, nous voulons nous assurer de son bon fonctionnement et ceci dans toutes les configurations sous lesquelles elle pourra être utilisée. Pour une bibliothèque dont le code est générique, cela pose des problèmes additionnels. De plus, à l'introduction de nouveaux compilateurs, de nouveaux environnements, de nouveaux standards, nous voulons minimiser le temps passé à ajuster le code. Nous mettons en œuvre des techniques pour répondre à ces défis.

7.2.1 Tester et vérifier une bibliothèque générique.

Tester le bon fonctionnement d'une bibliothèque générique est un problème délicat ; p.ex. nous ne pouvons pas certifier que toutes les combinaisons possibles de paramètres *template* soient testées. En effet, nous devons avant tout proposer une architecture de tests simple, efficace et rapide et et l combinatoire des possibles nous empêche de les tester entièrement. Nous proposons deux approches combinées. D'une part, à l'intérieur même du code, nous nous assurons que certaines propriétés ou pré-requis sont vérifiés, à l'exécution ou — mieux — à la compilation. C'est une programmation par contrat ([MJ07]). Un exemple de contrat des plus simples préconise l'utilisation du mot clef `const` pour qualifier certains membres, argument ou variables. Il existe de nombreux autres contrats. D'autre part, l'exécution des programmes de tests externes (dans les répertoires `tests` ou `benchmark` par exemple) permet classiquement de vérifier le bon fonctionnement de certains algorithmes et solutions.

7.2.1.1 Tester le bon fonctionnement des solutions.

Les tests, les exemples [4] et les benchmarks dans notre bibliothèque permettent d'une part de vérifier les bons fonctionnements et compilations des algorithmes et d'autre part de vérifier que la bibliothèque est bien installée. C'est très classique. Afin de rendre ces tests plus conviviaux pour les utilisateurs et efficaces pour les développeurs, B. David Saunders a créé la cible `make fullcheck` où « tout » est testé et a modifié la cible `make check` afin qu'elle teste seulement l'intégrité de l'installation.

Comme notre bibliothèque est une bibliothèque générique, et même si le contraire est tentant, il est nécessaire de tester ces algorithmes sur divers corps, avec divers types de matrices et sur des cas particuliers. Par exemple, dans LinBox, s'il est facile de choisir un `Modular<double>` pour tester son algorithme sur un corps $\mathbf{Z}/p\mathbf{Z}$, il peut aussi être intéressant de le tester sur un corps de NTL ou de Givaro ou même sur un `Modular<int>` ou `ModularBalanced<float>`... Ce travail de vérification de la généricité a révélé de nombreuses lacunes, incompatibilités, voire des erreurs. Désormais, lors de la découverte d'un code non testé une annotation par `pragma` est incluse afin de ne pas surprendre l'utilisateur et de marquer le code pour revue — car si tester une fonction ne représente pas une garantie ultime quant à son bon fonctionnement, ne pas la tester ouvre grandement la voie aux codes faux ou non compilables lors des évolutions de LinBox [5].

La suite valgrind [6] a permis, via son outil `memcheck` de colmater de bien nombreuses fuites de mémoire dans les tests et les exemples. Un script présent dans le répertoire des `tests` automatise ce débogage. Cependant, dans une bibliothèque d'en-têtes, vérifier les fuites de mémoire ne peut se faire que s'il y a instantiation : cela illustre encore, dans les tests et les exemples, la nécessité de recouvrir une majorité (de la combinatoire) des fonctions de notre bibliothèque générique.

Finalement, nous proposons un moyen de vérifier certaines méthodes et fonctions utilisées. Parfois, au lieu d'utiliser l'algorithme attendu, le compilateur choisira — à raison — une méthode différente.

4 — servant aussi d'exemples simples d'utilisation de la bibliothèque.
5 — Morale : de grands coups `grep/sed` ne font généralement pas tout !
6 — Disponible à `http://valgrind.org/`.

Afin de tester l'utilisation de la méthode correcte, une solution consiste en un journal (*log*) au format XML des opérations effectuées — par exemple selon le schéma d'un `commentator`. Nous proposons qu'à l'entrée et à la sortie de certaines fonctions, en mode débogage, nous ouvrions et fermions des balises avec attributs, dans un flux XML. Ainsi, nous créons un moyen automatique de vérifier que les chemins suivis à l'exécution sont effectivement les bons : il suffira de *parser* le fichier obtenu — bien plus facilement qu'un rapport de `commentator` car structuré.

7.2.1.2 Vérifier le code par lui-même.

Nous utilisons aussi des moyens introspectifs pour vérifier le code, en mode débogage. Les techniques utilisées sont par exemple les fonctions `*_check()` similaires à des assertions (`assert.h`), inclure[7] des tests entre des `#ifndef NDEBUG`, *etc.* En effet, tant que ces directives représentent un coût de compilation et d'exécution nuls en mode « distribution » (*release*), elles sont autant de garde-fous, d'aide au débogage et de sécurités en mode *debug*. Dans la même veine, et récemment, le nouveau standard `c++11` (ISO/IEC 14882:2011) apporte une nouvelle fonctionnalité — qu'il fallait auparavant maladroitement émuler[8] — qui permet d'effectuer des tests à la compilation : `static_assert`.

```
#ifdef __LINBOX_SUPPORT_CXX11
static_assert(MatrixTraits<Matrix>::hasGetEntry::value == true,
              "you need <Matrix> to have a getEntry method here.");
#endif
```

Code 7.2 — `static_assert` : assertion lors de la compilation.

D'autres méthodes que nous n'avons pas encore explorées existent pour découvrir des erreurs dès l'instanciation, notamment [SL00].

Cependant, bien souvent certains codes ne fonctionnent que dans des cas particuliers, n'interagissent pas comme on l'aurait souhaité, ou ne font pas ce qu'on attend — p.ex. les opérateurs `mod` et `operator%` suivent des conventions différentes dans les entiers de Givaro, ou la fonction `gcd` qui n'avait pas la même signature entre LinBox et Givaro... Pour cela, les champs de documentation `@warning` (paragraphe attirant l'attention) et `@pre` (pré-requis pour la fonction) révèlent déjà une utilité. Bien entendu, un avertissement dans la documentation `html` du code d'une fonction est totalement inutile lorsque cette fonction est enfouie dans un arbre d'appels (i.e. très indirectement appelée). C'est dans ces cas-là par exemple que les tests d'erreurs (à la compilation ou à l'exécution) sont primordiaux, mais seulement en mode de débogage. Nous pourrons encore aussi utiliser, certes avec parcimonie, des annotations de code invisibles en mode `release` sous forme de `#pragma message "msg..."`.

Le langage C++ prévoit aussi un système d'exceptions et leur récupération, mais nous éviterons de les utiliser en cas d'erreurs. En effet, nous préférons les garder comme alternative au système de *return status*, où nous levons une exception sur une possibilité connue et attendue d'échec d'un algorithme. Par exemple, nous préférons un test statique à faux que de lever une exception `NotImplementedYet` qui donnerait à l'exécution la surprise : `*** Error! *** \n in file 'xxx', at line yyy: the function 'zzz' is not implemented yet *** .`

7 — avec parcimonie car nous voulons éviter les directives pré-processeur qui nuisent à la bonne lecture du code.
8 — Par exemple :

```
#define STATIC_ASSERT(X, Y) ({ \
int __attribute__( error("assertion failed: '" #X "' (" #Y ")") )  static_assert(); \
( (X)?0:static_assert() ); })
```

7.2.1.3 Conclusion, prolongements.

Après une bonne maturation dans LinBox, la cible check de make a été portée dans Givaro et FFLAS-FFPACK. D'une part, nous allégeons ainsi la tâche de vérification de LinBox qui n'a besoin alors que de tester le bon interfaçage vers ces bibliothèques ; d'autre part, cela apporte des garanties sur le bon fonctionnement de ces bibliothèques elles-mêmes.

Bien sûr, il manque encore de nombreux efforts d'implantation, notamment de nombreux tests correspondant à des algorithmes non testés ou des configurations de template non instanciées, ou connues pour échouer.

Il manque aussi des tests qui vérifient que les bibliothèques que nous développons échouent « gracieusement » là où elles le devraient, par exemple, en vérifiant qu'une (la bonne) exception a été levée.

Il existe cependant des vérifications qui sont pour le moment impossible, par exemple vérifier automatiquement la bonne utilisation de la fonction espérée. Soit par exemple l'erreur suivante. Une classe `LinBox::myField` hérite d'une autre, `FFPACK::myField` qui redéfinit `init` avec un argument `Givaro::Integer`, sans rappeler la fonction init de base (oubli de `using Father_t::init;`). Bien que nécessaire à la conception générique d'un corps, cette fonction est cependant environ dix fois plus lente qu'un `init` sur un `double` dans `FFPACK::Field`. Comment remonter systématiquement jusqu'à ce genre d'« erreurs » ? Il faut déjà s'en rendre compte ! Pour cela, nous avons proposé un système de *benchmarks* (section 6.1). Autre solution, il faudra profiler « à la main » le code. Pour cela, on peut utiliser gprof avec l'option `-pg` passée à g++ (passée automatiquement si LinBox est configurée avec `--enable-profiling`). On peut aussi utiliser l'outil `callgrind` de valgrind et une interface graphique comme KCachegrind pour se repérer plus facilement dans l'arbre des appels.

7.2.2 Normalisation du code : robustesse et pérennisation.

Cette section s'attache à décrire les choix effectués pour rendre le code d'une bibliothèque plus robuste. Les buts affichés sont de la rendre plus facile à *packager* pour les distributions Linux, le code plus facile à appréhender ou à modifier, plus stable et plus portable. Nous avons besoin d'un code standard, structuré homogènement selon certaines conventions. D'une part, nous devons les décrire clairement, d'autre part pouvoir s'assurer qu'elles sont respectées — nous devons cependant éviter que ces règles ne deviennent trop contraignantes en en imposant par exemple un minimum possible et en les rendant supervisables automatiquement.

7.2.2.1 Standardisation du code.

La standardisation du code que nous avons entreprise affiche principalement deux objectifs :

— permettre une meilleure *portabilité* du code (entre architectures) ;

— rendre le code plus *robuste*.

Rendre du code portable, c'est respecter les normes du C++ tout en prenant en compte les spécificités des divers compilateurs. Pour cela, une première étape consiste à utiliser les mécanismes d'avertissements des compilateurs. Pour commencer, par défaut, seul `-Wall` est activé, en mode `--enable-warning=yes`, nous rajoutons pour gcc les options `-Wextra -Wno-unused-parameter` et en mode `--enable-warning=full`, les options `-Wuninitialized -Wconversion -Wcast-qual -ansi -pedantic -Wshadow -Wpointer-arith -Wcast-align -Wwrite-strings -Wno-long-long`. Avec icc, on rajoutera par exemple `-Wcheck`. Cela a entraîné un impressionnant nombre d'avertissements qu'il a fallu réduire à (presque) zéro. Dans le cas de LinBox, comme cette bibliothèque inclut directement du code de Givaro et FFLAS-FFPACK, le même traitement a été appliqué à ces bibliothèques. Cela a conduit à une factorisation des `configure.ac` et des macros `*.m4` dans le répertoire `macros/` et ainsi donc à une meilleure maintenance de ces mécanismes. Au passage, ce même type d'avertissements a été appliqué au système de `Makefile`, ce qui a permis de mettre à jour les conventions obsolètes, de moderniser une partie du code lié aux autotools et de supprimer bon nombre de messages d'avertissement lors de la configuration par exemple.

Ensuite, tester ce code avec différents compilateurs rend le code encore plus robuste. Cela permet aussi de détecter les erreurs produites par les compilateurs. Par exemple, le compilateur gcc-4.4.5 ne définit pas un `long int` comme un type standard : dans ce cas-là, on singularisera le code pour cette version particulière de gcc, grâce à des macros pré-processeur. Nous montrons des exemples dans le code 7.3.

```
// matrix/blas-matrix.h
#ifdef __GNUC__
#   ifndef __x86_64__
#       if ( __GNUC__==4 && __GNUC_MINOR__==4 && __GNUC_PATCHLEVEL__==5 )
// code for gcc-4.4.5 on non x86_64 machines where long is not an intX_t type.
template<class T>
BlasMatrix (const _Field &F, const long &m, const T &n) ;
#       endif
#   endif
#endif
```

Code 7.3 — Spécialisation du code pour certains compilateurs.

Dans le tableau 7.1 nous montrons en exemple l'évolution de la prise en charge [9] des compilateurs au fil des évolutions récentes de LinBox (et Givaro, FFLAS-FFPACK) sur des architectures x86 et x86_64. Le support de nvcc nécessite de sérieuses modifications (renommer les `.C` en `.cpp` ou `.cc`, passer certaines options avec le switch `-Xcompiler`, *etc.*).

Tab. 7.1 — Évolution de la prise en charge des compilateurs dans LinBox.
✔ les tests compilent, ✗ incompatibilité profonde, ✍ légères adaptations (patches) nécessaires.

environnement			compilateur							
LinBox	Givaro	FFLAS-FFPACK	gcc-4.1 à 4.3	gcc-4.4 à 4.5	gcc-4.6	icc-11 et 12	gcc-4.7	nvcc 3.0, 4.1	clang 2.9, 3.0	ekopath (2012)
1.1.6	3.2.16	—	✔	✍	✍	✗	✗	✗	✗	✗
1.1.7	3.3.3	—	✔	✔	✍	✔	✗	✗	✗	✗
1.2.0	3.4.2	1.4.0	✔	✔	✔	✔	✗	✗	✗	✗
1.2.2	3.5.0	1.4.4	✔	✔	✔	✔	✍	✗	✗	✗
1.3.0	3.6.0	1.5.0	✔	✔	✔	✔	✔	✗	✔	✔

Finalement, nous rendons le code plus robuste en obéissant aux notations standards. Prenons par exemple le cas des types entiers. La bibliothèque LinBox propose les entiers arbitrairement longs via Givaro (qui eux-mêmes sont *wrappés* depuis gmp) ainsi que des entiers machines, signés ou non. Le fichier `stdint.h` permet d'utiliser des types standards qui remplacent `int32` ou `int64` précédemment définis dans `integer.h` et construits à la compilation. On utilise donc les types standards `UintX_t` avec $X \in \{8, 16, 32, 64\}$ et `U`=' ' ou 'u'. De plus, on généralise les entiers non signés `size_t` dans une utilisation en tant que taille — avec éventuellement sa variante signée `ssize_t`.

Remarque. Comme dans la bibliothèque ffspmvgpu, on pourrait aussi utiliser un type indice `index_t` qui est défini comme `typdef uint32_t index_t`. Cela permettrait notamment de ne pas utiliser des indices trop grands (64 bits) alors que les tailles des vecteurs sont rarement plus grandes que $10\,000 \times 10\,000$.

9 — Selon les versions de LinBox, le compilateur gcc-4.2 peut produire du mauvais code.

7.2.2.2 Structuration du code : quelques conventions.

Pour donner une unité à notre bibliothèque de calcul (ici les projets LinBox, FFLAS-FFPACK ou Givaro), une structuration des fichiers sources a été engagée. Cela passe par des conventions, des styles d'en-tête et de pied de page... Nous pensons qu'un style de fichier propre et cohérent permet de faciliter la maintenance et la lisibilité du source.

7.2.2.3 Les licences.

Tout d'abord, ces bibliothèques sont publiées dans le domaine public sous une licence libre : la LGPL-2.0 jusque LinBox-1.2.2, FFLAS-1.4.3 puis LGPL-2.1 [10] ou ultérieure pour les versions suivantes de LinBox et FFLAS. La bibliothèque Givaro est quant à elle sous une licence CeCILL-B [11].

Suite à des problèmes [12] de paquetage, un effort a été fait pour rendre LinBox et FFLAS-FFPACK en règle par rapport à leur licence. Les problèmes majeurs étaient : des fichiers sans licence, sans auteur, sans copyright ou avec des licences diverses. Il a donc dû falloir faire un « grand ménage ». Pour ne pas avoir à répéter ce travail très fastidieux, nous avons décidé de rendre homogène les en-têtes de *tous* les fichiers en suivant les recommandations des licences GNU [13]. Tous les en-têtes sont donc similaires au code 7.4.

```
/* Copyright (C) <+years+> the members of the LinBox group
 * Written by <+someone+> < <+her mail+> >
 *
 * This file is part of the LinBox library.
 *
 * ========LICENCE========
 * LinBox is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * LinBox is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 * ========LICENCE========
 *
 * other mentions
 */
```

Code 7.4 — Mention longue de la licence de LinBox.

Chaque fichier publié mentionne donc un copyright, le nom de la licence, la mention « *any later version* » et les fichiers `COPYING` et `COPYING.LESSER` contiennent les textes des licences nécessaire à leur compréhension. Un délimiteur `========LICENCE========` permet essentiellement de faciliter le travail d'un script lors d'un éventuel changement de version et surtout de vérifier que *tous* les fichiers distribués portent une licence !

10 — Voir http://www.gnu.org/licenses/lgpl-2.1.html.

11 — Voir le texte à l'adresse http://www.cecill.info/licences/Licence_CeCILL-B_V1-fr.html. Cette licence se rapproche d'une licence BSD et ne serait pas compatible avec la GPL. Sa déclinaison 'C' est par contre compatible avec la LGPL.

12 — par exemple le bogue https://bugzilla.redhat.com/show_bug.cgi?id=529466

13 — *cf.* http://www.gnu.org/licenses/licenses.html.

Remarque. Ce texte est long et gaspille de la place — au moins pour les éditeurs qui ne plient pas automatiquement ce code. Nous avons donc proposé de changer la mention entre les délimiteurs par le code 7.5.

```
 * ...
 * This file is part of the LinBox library.
 *
 * LinBox is free software.
 * <+LinBox is licenced under the GNU LGPL 2.1 or any later version. See the
 * files COPYING and COPYING.LESSER in the source directory for more
 * information about the licence.+>
 */
```

Code 7.5 — Licence raccourcie.

Il ne reste alors qu'à remplacer le texte entre les marqueurs <+ et +> lors de la création des sources à distribuer (`make dist`) par l'en-tête traditionnel recommandé (code 7.4). Cet en-tête de fichier est moins distrayant.

7.2.2.4 Architecture des fichiers.

Toujours dans un souci d'homogénéité, nous avons proposé le style de fichier suivant (code 7.6).

```
/* Copyright */
/* Licence */

/*! @file nom-unique.h
 *  @ingroup monmodule
 *  @brief   courte description
 *  @details description plus precise ici
 */

#ifndef __LINBOX_monmodule_nom_unique_H
#define __LINBOX_monmodule_nom_unique_H

#endif //__LINBOX_monmodule_nom_unique_H
/* vim/emacs formatting lines */
```

Code 7.6 — Modèle d'un fichier source `.h`.

Les gardes sont la concaténation du préfixe `__LINBOX_`, d'un identifiant unique en minuscule avec comme séparateur une espace soulignée _ et du suffixe `_H`, `_INL` selon l'extension. Ces gardes ont une double utilité. D'une part ce préfixe unique assure l'inclusion du fichier d'en-tête car il n'est pas défini dans une autre bibliothèque. Ce fut par exemple le cas du trop général `blackbox/transpose.h` qui définissait simplement un `TRANSPOSE_H`, défini par ailleurs. D'autre part, pour rendre la garde facilement *greppable*, bien compréhensible et unique, le fichier `linbox/solutions/solve.h` aura par exemple [14] comme garde `__LINBOX_solutions_solve_H`. Cela permet par exemple de différentier les fichiers `blackblox/dense.h` et `matrix/dense.h`.

14 — Pour mémoire, le petit script suivant répond à cette règle :

```
tr '[:upper:]' '[:lower:]' | tr -cs '[:alnum:]' '_' |
                sed 's/\([a-z]*\)_\(.*\)_\([a-z]*\)_/_\U\1\E_\2_\U\3\E_/g'
```

Remarque. Aussi, pour éviter les définitions multiples, nous avons choisi de nommer une définition locale `__LBX_XXX` (et à la fin de chaque fichier, on trouve un `#undef __LBX_XXX`) alors qu'une définition globale est préfixée par `__LINBOX_`. De cette manière, toutes les définitions globales sont préfixées de la même manière.

Finalement, en début de code, le commentaire doxygen (`@file...`) apporte une description du fichier pour en avoir une vue d'ensemble : quelle utilité, quelles classes et fonction fournies, pour quel usage, *etc.* ?

7.2.2.5 Style de codage.

Un style de codage est proposé depuis des années sur le site internet de LinBox dans la section développement, mais de nombreux fichiers ne le respectaient pas. Il n'est pas très important en soi que certaines accolades ne soient fermées sur la bonne ligne, on pourrait même dire que de rechercher à appliquer strictement les règles de placement des accolades est une perte de temps, voire une erreur. Cependant, il est important que le code en général respecte des règles établies et que tous les programmeurs « jouent le jeu ». Des bonnes règles de style de codage donnent une unité à un projet et facilitent sa lecture.

Pour suivre donc les recommandations de style de codage et rendre ce style homogène dans la bibliothèque toute entière, tous les fichiers se terminent par les lignes suivantes (code 7.7), compréhensibles par `vim` et `emacs` en configuration standard. Elles permettent un (re-)formatage [15] automatique du code.

```
// Local Variables:
// mode: C++
// tab-width: 8
// indent-tabs-mode: nil
// c-basic-offset: 8
// End:
// vim:sts=8:sw=8:ts=8:et:sr:cino=>s,f0,{0,g0,(0,\:0,t0,+0,=s
```

Code 7.7 — Styles de codage pour Vim et Emacs.

Il existe une multitude de styles de fichier et de conventions [16], nous montrons ici quelques exemples du formatage voulu, très proche du style Linux (K&R avec huit espaces d'indentation). Globalement, nous essayons de coder avec du bon sens, sans religion stricte ; nous proposons cependant quelques règles. Nous évitons les indentations trop profondes, les instructions multiples sur une ligne, les morceaux de codes difficiles à lire. Les accolades rendent les codes plus sûrs et lisibles (sauf éventuellement pour les instructions tenant sur une ligne) et facilitent le pliage de code. Il vaut mieux rajouter des espaces pour ne pas rendre le code trop dense à lire. Le parenthésage implicite est source d'erreurs. Les variables utilisent de préférence plusieurs caractères, leur nom est évocateur. Les fonctions sont de préférence notée en style *camelCase* [17] ou approché et les variables du pré-processeur seront en majuscules, sur plusieurs caractères (éviter surtout les `_[A-Z]` qui peuvent être définis par ailleurs, notamment sur les architectures solaris !). Dans les énumérations, les listes, *etc.* un alignement sur les symboles `&` ou `=` par exemple peut les rendre plus lisibles.

Un fichier `linbox/template.h` a été créé pour résumer entre autres ces conventions dans un exemple. Il peut servir de modèle pour la création d'un nouveau fichier.

Si comme nous l'avons dit plus haut, nous n'allons pas chercher toutes les accolades mal fermées ou ouvrir pour réindenter tous les fichiers du projet, nous avons cependant écrit de petits scripts qui permettent de corriger automatiquement, ou du moins détecter, certaines erreurs (code 7.12).

15 — Cependant, bien que très proches, ces formats ne sont pas tout à fait équivalents.

16 — p.ex., http://www.kernel.org/doc/Documentation/CodingStyle ou http://wiki.qt-project.org/Coding_Style et suivants (`Coding_Conventions`, `API_Design_Principles`...)

17 — chatMot, mots en minuscules, sans espace ni espace souligné, débutant par une majuscule, sauf le premier

```
namespace LinBox {
␣␣␣␣␣␣␣␣template<class T>
␣␣␣␣␣␣␣␣int function (T& toto)
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣int i=1,j;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣code(toto);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if (toto.bad())
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return 1;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return 0;
␣␣␣␣␣␣␣␣}
} // LinBox
```

Code 7.8 — Style de codage : fonctions

```
template< class T >
class A {
public:
␣␣␣␣␣␣␣␣A() :
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a_(0)
␣␣␣␣␣␣␣␣{
␣␣␣␣␣␣␣␣}

␣␣␣␣␣␣␣␣void b();

private :
␣␣␣␣␣␣␣␣a_ ;
}; // A
```

Code 7.9 — Style de codage : classe

```
if (a) {
␣␣␣␣␣␣␣␣a_ok();
␣␣␣␣␣␣␣␣//...;
}
else { /* cas non a */
␣␣␣␣␣␣␣␣non_a();
}

for (i = 0 ; i < 10 ; ++i) {
␣␣␣␣␣␣␣␣faire(i);
␣␣␣␣␣␣␣␣//...;
}
```

Code 7.10 — Style de codage : if-else

```
switch (b) {
case 1 : {
␣␣␣␣␣␣␣␣x();
␣␣␣␣␣␣␣␣break;
␣␣␣␣␣␣␣␣}
case 2 :
␣␣␣␣␣␣␣␣y();
␣␣␣␣␣␣␣␣break;
default : {
␣␣␣␣␣␣␣␣z() ;
␣␣␣␣␣␣␣␣}
}
```

Code 7.11 — Style de codage : switch

```
#!/bin/bash -
# Use Vim to indent properly a (list of) file(s)
vim -E -s $1 <<-ENDMSG
        :args $1 | argdo execute "normal gg=G" | update
ENDMSG
```

Code 7.12 — Exemple d'automatisation de respect des règles.

7.3 Conclusion.

Dans ce chapitre, nous avons d'une part montré comment documenter aisément, grâce au programme doxygen, une bibliothèque générique : documentation transversale par groupes et modules, structurations de la documentation en fonction des utilisateurs... Il est en effet essentiel de documenter un code efficacement de manière à ce que le nouvel utilisateur se retrouve facilement, comprenne la structuration et les dépendances dans la bibliothèque, cerne ses fonctionnalités. Une bonne documentation, ainsi qu'une annotation propre du code est une première étape vers un code maintenable, évolutif et débogable. D'autre part, nous avons montré comment combiner diverses techniques afin de s'assurer d'une qualité de code. Cela passe par un ensemble de tests hors et à l'intérieur du code, résidant sur l'instanciation effective du code mais aussi lors de son exécution. En amont, la qualité du code est assurée par une normalisation et des règles pour lesquelles nous avons des scripts qui vérifient leur bon usage et assurent homogénéité et robustesse du code. Indirectement, documentation et robustesse du code le rendent stable et pérenne.

En pratique, dans LinBox, FFLAS-FFPACK et Givaro nous avons restructuré la documentation et augmenté très sensiblement la couverture de code en termes de documentation et de tests.

Conclusion.

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

Martin Golding

Conclusion et perspectives.

Après quatre années de thèse, qu'avons-nous réellement apporté et quel prolongements à ces travaux pouvons-nous raisonnablement attendre ? Nous décrivons dans les prochains paragraphes quelles réponses nous apportons et quelles perspectives nous proposons.

Tout d'abord, nous avons conçu, développé et utilisé des outils qui nous ont permis de répondre à la question « comment ordonner efficacement un algorithme de multiplication rapide à la Strassen ? » Grâce au développement de nombreux outils et d'études précises de complexité, nous avons permis de réaliser une multiplication rapide en place. Puis nous avons essayé plus loin : nous avons développé des outils pour rechercher de meilleurs algorithmes que celui de Winograd. Malgré la puissance des ordinateurs à disposition, la situation n'a guère changé dans les dernières décennies ([BDEZ12, OM10, CBH11]). Si nous n'avons pas non plus permis de trouver de meilleurs algorithmes, pour les petites tailles au moins, nous avons cependant dévoilé la perspective d'utiliser des formules approchées pour accélérer la routine `fgemm`. En outre, une nouvelle perspective qui s'ouvre dans le développement de LinBox est la création de solutions dont une méthode est `inPlace` et qui cherche à réduire « au maximum », tout en gardant une complexité « raisonnable », la mémoire supplémentaire allouée lors de l'exécution de cette solution, ces problèmes restant à définir.

Si la multiplication de matrices denses est essentielle pour la famille des algorithmes « par blocs », la multiplication de matrices creuses et de (blocs de) vecteurs denses est elle essentielle pour les algorithmes BoîteNoire. Nous avons étudié les divers formats de stockage pour ces matrices creuses et en avons extrait de nouveaux formats hybrides performants sur les anneaux $\mathbf{Z}/m\mathbf{Z}$. Nous avons aussi exploré des techniques de parallélisation (multi-fils sur multicœurs, massivement multi-fils sur cartes graphiques) qui nous ont permis d'accélérer significativement l'opération SpMV et en avons démontré la puissance dans l'exemple du calcul du rang. Une introduction générique et transparente de ces *accélérateurs* est une question qui reste grandement à explorer dans l'optique de LinBox-2.0.

La remarque triviale consistant à dire que cette opération de multiplication est l'opération de base en algèbre linéaire nous a permis de mieux centrer LinBox autour de ses problèmes d'API en faisant de cette opération une solution, un bloc de base. Nous avons alors dégagé divers principes — conception de la bibliothèque par briques de base, modulables et interchangeables — et ceci dans le but non seulement de conserver voire accroître, si cela est quantifiable, la généricité de nos bibliothèques mathématiques, mais aussi d'améliorer leurs performances. Cette modularisation permet aussi de bâtir des bibliothèques avec le principe *concevoir pour changer* : changer d'algorithmes, d'implantations, de modèle de calcul...

Dans le projet du développement de la bibliothèque LinBox vers sa version 2.0, nous avons abordé divers problèmes de conception :

— définition claire et structurée des matrices et des vecteurs ;
— standardisation, normalisation du code pour sa pérennisation ;
— agencement et approfondissement de la documentation ;
— meilleure couverture des tests pour une bibliothèque d'en-têtes ;
— système de mesure de performances permettant une auto-optimisation de la bibliothèque et une visualisation de ses performances.
— ...

La route est encore longue vers LinBox-2.0 et un gros travail d'ingénierie et de recherche est encore nécessaire mais nous avons contribué à en poser les bases et à faciliter la transition vers nos nouvelles attentes. Nous voulons certes essentiellement concevoir une utilisation transparente et modulaire des architectures parallèles via des blocs PBB (*parallel building blocs*, *cf.* kaapi, Intel® TBB [18], Thurst [19], le projet ANR hpac [20]). Mais si ces accélérations via les nouvelles architectures parallèles sont tentantes, nous pouvons aussi nous pencher sur des questions telles la réduction de la taille des exécutables ([BJ11]), ou une meilleure auto-optimisation de nos bibliothèques ([DMV+08, LDT09]).

18 — *cf.* http://threadingbuildingblocks.org/.
19 — *cf.* http://code.google.com/p/thrust/.
20 — description et avancement sur http://hpac.gforge.inria.fr/.

Index

Index des mots clefs de cette thèse. Les pages en caractères gras correspondent aux entrées principales.

A

algorithmes
- Hermite **98**
- multiplication matricielle . . . . . *voir* multiplication matricielle
- restes chinois **95**
- résolution de systèmes **99**

ATI Close to Meta/Stream . . . . . *voir* langages

B

benchmarks **102–107**

bibliothèques & logiciels
- cuSPARSE **52**, **67**, *73*
- doxygen **115–116**, *123*, *125*
- FFLAS-FFPACK **4**, *5*, **13**, *51*, *57*, *59*, *73*, **78**, **80–82**, **92–95**, **101**, **106**, **109**, **110**, *110*, **111**, **115**, **116**, **119–121**
- ffspmvgpu **51**, **53–67**, *73*, *120*
- FLINT *4*, *78*, **95**, **96**, *99*, *102*
- fplll *5*, *80*, *110*
- galet **18–20**, **24**, **27**, **29**
- Givaro . . . **4**, *5*, *13*, *72*, **78**, **80–82**, **95**, **106**, **108**, **110**, *110*, **115**, **116**, *118*, **119–121**
- gmp *5*, *78*, **95**, *110*, gmp120
- gnuplot **103**, **104**, **107**
- graphviz *20*
- IML *5*, **80**, **95**, **99**, *108*
- LinBox . *4*, **5**, *14*, **77–90**, **97–99**, **102–109**, **113–123**
- m4ri *13*, **36**, **85**, **86**, **90**
- Maple *5*, *24*
- NTL *4*, *5*, *98*, *101*, *108*
- Oski **61**, *67*
- Sage *5*, *81*, *98*, *102*, *113*
- STL *77*, **84**, **86**, *86*, **97**
- valgrind **106**, **117**, **119**

C

C/C++ . . . . . *voir* langages

clang . . . . . *voir* compilateurs

compilateurs
- clang **120**
- ekopath **120**
- gcc *51*, *82*, *101*, *110*, *119*, **120**, *120*
- icc *51*, *119*, **120**
- nvcc **62**, **120**

conception logicielle
- blocs de base *97*, **107–112**
- contrôleur–modules **91**
- patron modèle-vue-controlleur **43**
- patron stratégie **91**

Cuda . . . . . *voir* langages

cuSPARSE . . . . . *voir* bibliothèques & logiciels

D

documentation **114–117**

doxygen . . . . . *voir* bibliothèques & logiciels,documentation

E

ekopath . . . . . *voir* compilateurs

F

FFLAS-FFPACK . . *voir* bibliothèques & logiciels

ffspmvgpu . . . . . *voir* bibliothèques & logiciels

FLINT . . . . . *voir* bibliothèques & logiciels

fplll . . . . . *voir* bibliothèques & logiciels

G

galet (logiciel) . *voir* bibliothèques & logiciels
galet (jeu) ..... *voir* jeu de galet
gcc/g++ ..... *voir* compilateurs
Givaro ..... *voir* bibliothèques & logiciels
gmp ..... *voir* bibliothèques & logiciels
gnuplot ..... *voir* bibliothèques & logiciels
graphviz ..... *voir* bibliothèques & logiciels

I

icc/icpc ..... *voir* compilateurs
IML ..... *voir* bibliothèques & logiciels
instructions SIMD **4**, **95**, **108**

J

jeu de galet **16–21**
    règles **17**

K

kaapi ..... *voir* langages

L

langages
    ATI Close to Metal/Stream *51*
    C/C++ **5**, **19**, **78**, **79**, **103**, **119**
    Cuda *4*, **51**, **62**, **65**, **67**, **73**
    kaapi *71*, *130*
    MPI *51*
    openCL **4**, **51**, *67*, **73**
    openGL/GLSL *4*, *51*
    openMP **51**, **59**, **73**, **95**
    XML **19**, **20**, **43**, **117**
LinBox ..... *voir* bibliothèques & logiciels

M

Maple ..... *voir* bibliothèques & logiciels
m4ri ..... *voir* bibliothèques & logiciels
MPI ..... *voir* langages
multiplication matricelle
    algorithme de Bodrato **12**, **13**, *36*
    algorithme de Strassen **12**, **40**, *43*
    algorithme de Winograd .. **12**, *91*, **96**, *103*
    multiplication creuse **53–67**
    multiplication sur des entiers **95–97**
    solution `mul` **90–97**

N

NTL ..... *voir* bibliothèques & logiciels
nvcc ..... *voir* compilateurs

O

openCL ..... *voir* langages
openGL/GLSL ..... *voir* langages
openMP ..... *voir* langages
Oski ..... *voir* bibliothèques & logiciels

P

patron de conception ..... *voir* conception logicielle

S

Sage ..... *voir* bibliothèques & logiciels
SSE ..... *voir* instructions SIMD
STL ..... *voir* bibliothèques & logiciels

T

tests **117–119**

V

valgrind ..... *voir* bibliothèques & logiciels

X

XML ..... *voir* langages

Bibliographie

[Ale01] A. Alexandrescu — *Modern C++ design: generic programming and design patterns applied*. C++ in-depth series. Addison-Wesley, 2001.
Cité à la p. 78.

[ALN+94] J. D. Andrew, A. Lumsdaine, X. Niu, R. Pozo & K. Remington — A sparse matrix library in C++ for high performance architectures, 1994.
Cité à la p. 50.

[Bai88] D. H. Bailey — Extra high speed matrix multiplication on the Cray-2. *SIAM Journal on Scientific and Statistical Computing* **9** (1988), n° 3, p. 603–607.
Cité à la p. 12.

[BDEZ12] R. Barbulescu, J. Detrey, N. Estibals & P. Zimmermann — Finding optimal formulae for bilinear maps. *IACR Cryptology ePrint Archive* **2012** (2012), p. 110.
Cité à la p. 129.

[BDG10] B. Boyer, J.-G. Dumas & P. Giorgi — Exact sparse matrix-vector multiplication on GPU's and multicore architectures. *In Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, p. 80–88, New York, NY, USA, 2010. ACM.
Cité à la p. 70.

[BDPZ09] B. Boyer, J.-G. Dumas, C. Pernet & W. Zhou — Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. *In Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, p. 55–62, New York, NY, USA, 2009. ACM.
Cité à la p. 36.

[BFF+09] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert & C. E. Leiserson — Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. *In SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, p. 233–244, New York, NY, USA, 2009. ACM.
Cité à la p. 52.

[BG09] N. Bell & M. Garland — Implementing sparse matrix-vector multiplication on throughput-oriented processors. *In SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 1–11, New York, NY, USA, 2009. ACM.
Cité aux p. 52, 59 et 61.

[Bin80] D. Bini — Relations between exact and approximate bilinear algorithms. applications. *Calcolo* **17** (1980), p. 87–97. 10.1007/BF02575865.
Cité à la p. 45.

[BJ11] L. Bourdev & J. Järvi — Efficient run-time dispatching in generic programming with minimal code bloat. *Sci. Comput. Program.* **76** (2011), nº 4, p. 243–257.
Cité à la p. 130.

[BM10] J.-F. Biasse & J. Michael — Practical improvements to class group and regulator computation of real quadratic fields. *Lecture notes in computer science* (2010).
Cité à la p. 98.

[Bod10] M. Bodrato — A Strassen-like matrix multiplication suited for squaring and higher power computation. *In* S. M. Watt, éditeur —*ISSAC 2010: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, p. 273–280. ACM, juillet 2010.
Cité aux p. 12 et 46.

[BP94] D. Bini & V. Pan — *Polynomial and Matrix Computations, Volume 1: Fundamental Algorithms.* Birkhauser, Boston, 1994.
Cité à la p. 12.

[CBH11] N. Courtois, G. V. Bard & D. Hulme — A new general-purpose method to multiply 3x3 matrices using only 23 multiplications. *CoRR* (2011).
Cité à la p. 129.

[CBS97] M. Clausen, P. Bürgisser & M. A. Shokrollahi — *Algebraic Complexity Theory*. Springer, 1997.
Cité à la p. 12.

[CC82] T.-W. J. Chou & G. E. Collins — Algorithms for the solution of systems of linear diophantine equations. *SIAM J. Comput.* **11** (1982), nº 4, p. 687–708.
Cité à la p. 98.

[CK91] D. G. Cantor & E. Kaltofen — On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica* **28** (1991), nº 7, p. 693–701.
Cité à la p. 70.

[Cop94] D. Coppersmith — Solving homogeneous linear equations over GF[2] via block Wiedemann algorithm. *Mathematics of Computation* **62** (1994), nº 205, p. 333–350.
Cité à la p. 69.

[CS05] Z. Chen & A. Storjohann — A BLAS based C library for exact linear algebra on integer matrices. *In ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, p. 92–99, New York, NY, USA, 2005. ACM.
Cité aux p. 96 et 99.

[CW90] D. Coppersmith & S. Winograd — Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation* **9** (1990), nº 3, p. 251–280.
Cité à la p. 12.

[DEG+05] J.-G. Dumas, W. Eberly, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, A. Lobo, C. Pernet, B. D. Saunders, W. J. Turner, G. Villard & Z. Wan — LinBox-1.0: Exact computational linear algebra, juillet 2005.
Cité à la p. 4.

[DGEVU07] J.-G. Dumas, P. Giorgi, P. Elbaz-Vincent & A. Urbańska — Parallel computation of the rank of large sparse matrices from algebraic K-theory. *In* S. Watt, éditeur —*PASCO 2007*, p. 43–52. Waterloo University, Ontario, Canada, juillet 2007.
Cité aux p. 69 et 70.

[DGG+02] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner & G. Villard — LinBox: A generic library for exact linear algebra. *In Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*. World Scientific Pub, août 2002.
Cité à la p. 84.

[DGP02] J. G. Dumas, T. Gautier & C. Pernet — Finite field linear algebra subroutines. *In* T. Mora, éditeur —*ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, p. 63–74, New York, NY, USA, juillet 2002. ACM.
Cité aux p. 13 et 49.

[DGP04] J.-G. Dumas, P. Giorgi & C. Pernet — FFPACK: Finite field linear algebra package. *In* J. Gutierrez, éditeur —*ISSAC'2004*, p. 119–126. ACM Press, New York, juillet 2004.
Cité à la p. 13.

[DGP08] J.-G. Dumas, P. Giorgi & C. Pernet — Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.* **35** (2008), n° 3, p. 1–42.
Cité aux p. 4, 13, 49 et 95.

[DGPS10] J.-G. Dumas, T. Gautier, C. Pernet & B. D. Saunders — LinBox founding scope allocation, parallel building blocks, and separate compilation. *In* K. Fukuda, J. Van Der Hoeven & M. Joswig, éditeurs —*3rd International Congress on Mathematical Software, ICMS 2010, September, 2010*, vol. 6327 de *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, p. 77–83, Kobe, Japon, septembre 2010. Springer.
Cité aux p. 79 et 108.

[DGR10] J.-G. Dumas, T. Gautier & J.-L. Roch — Generic design of Chinese remaindering schemes. *In* M. M. M. . J.-L. Roch, éditeur —*Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, p. 26–34, Grenoble, France, juillet 2010. Association for Computing Machinery.
Cité aux p. 72 et 96.

[DHSS94] C. C. Douglas, M. Heroux, G. Slishman & R. M. Smith — GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics* **110** (1994), p. 1–10.
Cité aux p. 12, 22, 36 et 37.

[Dix82] J. D. Dixon — Exact solution of linear equations using p-adic expansions. *Numerische Mathematik* **40** (1982), p. 137–141.
Cité à la p. 99.

[DMV+08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf & K. Yelick — Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *In High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, p. 1–12, novembre 2008.
Cité à la p. 130.

[DS02] F. Desprez & F. Suter — Impact of Mixed–Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. Research Report RR-4482, INRIA, 2002.
Cité à la p. 38.

[FWH04] E. Fogel, R. Wein & D. Halperin — Code flexibility and program efficiency by genericity: Improving cgal's arrangements. *In In Proc. 12th Annu. Euro. Sympos. Alg*, p. 664–676. Springer-Verlag, 2004.
Cité à la p. 77.

[Gam95] E. Gamma — *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
Cité à la p. 78.

[GG99] J. v. Gathen & J. Gerhard — *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.
Cité à la p. 70.

[Gi004] P. Giorgi — *Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque* LinBox. Thèse de doctorat, École normale supérieure de Lyon, décembre 2004.
Cité aux p. 78 et 99.

[GJK+05] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser & S. Schupp — Generic programming and high-performance libraries. *International Journal of Parallel Programming* **33** (2005), p. 145–164. 10.1007/s10766-005-3580-8.
Cité à la p. 78.

[GJS01] M. Giesbrecht, M. J. J. Jr. & A. Storjohann — Algorithms for large integer matrix problems. *In* S. Boztas & I. Shparlinski, éditeurs —*AAECC*, vol. 2227 de *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, p. 297–307. Springer, 2001.
Cité à la p. 98.

[GJV03] P. Giorgi, C.-P. Jeannerod & G. Villard — On the complexity of polynomial matrix computations. *In* R. Sendra, éditeur —*Proceedings of the 2003 ACM International Symposium on Symbolic and Algebraic Computation, Philadelphia, Pennsylvania, USA*, p. 135–142. ACM Press, New York, août 2003.
Cité aux p. 70 et 72.

[GVR+05] T. Gautier, G. Villard, J.-L. Roch, J.-G. Dumas & P. Giorgi — Givaro, a C++ library for computer algebra: exact arithmetic and data structures. Software, ciel-00000022, octobre 2005. www-lmc.imag.fr/Logiciels/givaro.
Cité à la p. 4.

[HHM10] S. A. Haque, S. Hossain & M. M. Maza — Cache friendly sparse matrix-vector multiplication. *In Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, p. 175–176, New York, NY, USA, 2010. ACM.
Cité à la p. 50.

[HK71] J. E. Hopcroft & L. R. Kerr — On minimizing the number of multiplications necessary for matrix multiplication. **20** (1971), n° 1, p. 30–36.
Cité à la p. 11.

[HK81] J.-W. Hong & H. T. Kung — I/O complexity: The red-blue pebble game. *In STOC*, p. 326–333. ACM, 1981.
Cité à la p. 18.

[HLJJ+96a] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao & T. Turnbull — Implementation of Strassen's algorithm for matrix multiplication. *In* ACM, éditeur —*Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*. ACM Press and IEEE Computer Society Press, 1996.
Cité à la p. 12.

[HLJJ+96b] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao & T. Turnbull — Strassen's algorithm for matrix multiplication : Modeling analysis, and implementation. Rapport technique, Center for Computing Sciences, novembre 1996. CCS-TR-96-17.
Cité aux p. xiii, 12, 13, 14, 16, 17, 21, 22, 23, 32 et 37.

[HMM98] G. Havas, B. S. Majewski & K. R. Matthews — Extended gcd and hermite normal form algorithms via lattice basis reduction. *Experimental Mathematics* **7** (1998), p. 125–136.
Cité à la p. 98.

[JM86] R. W. Johnson & A. M. McLoughlin — Noncommutative bilinear algorithms for 3×3 matrix multiplication. *SIAM J. Comput.* **15** (1986), n° 2, p. 594–603.
Cité à la p. 39.

[KB79] R. Kannan & A. Bachem — Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM J. Comput.* **8** (1979), n° 4, p. 499–507.
Cité à la p. 98.

[KL94] E. Kaltofen & A. Lobo — Factoring high-degree polynomials by the black box Berlekamp algorithm. *In* ACM, éditeur —*ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation: July 20–22, 1994, Oxford, England, United Kingdom*, p. 90–98, pub-ACM:adr, 1994. ACM Press.
Cité à la p. 49.

[Kre76] A. Kreczmar — On memory requirements of Strassen's algorithms. *In* A. Mazurkiewicz, éditeur —*Proceedings of the 5th Symposium on Mathematical Foundations of Computer Science*, vol. 45 de *LNCS*, LNCS, p. 404–407, Gdańsk, Poland, septembre 1976. Springer.
Cité à la p. 26.

[KS91] E. Kaltofen & B. D. Saunders — On Wiedemann's method of solving sparse linear systems. *In Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)*, vol. 539 de *LNCS*, LNCS, p. 29–38, octobre 1991.
Cité à la p. 69.

[Lad76] J. D. Laderman — A non commutative algorithm for multiplying 3×3 matrices using 23 multiplications. *Bull. of the Am. Math. Soc.* **82** (1976), n° 1, p. 126–128.
Cité à la p. 39.

[LDT09] Y. Li, J. Dongarra & S. Tomov — A note on auto-tuning GEMM for GPUs. *In Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, p. 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.
Cité à la p. 130.

[MC79] R. T. Moenck & J. H. Carter — Approximate algorithms to derive exact solutions to systems of linear equations. *In Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, p. 65–73, London, UK, 1979. Springer-Verlag.
Cité à la p. 99.

[MFPT10] M. Martone, S. Filippone, M. Paprzycki & S. Tucci — On BLAS operations with recursively stored sparse matrices. *In Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, septembre 2010.
Cité à la p. 57.

[MJ07] B. Meyer & P. Jouvelot — *Conception et programmation orientées objet*. Eyrolles, 2007.
Cité à la p. 117.

[MM09] M. Moreno-Mazza — communications personelles, 2009.
Cité à la p. 18.

[MW01] D. Micciancio & B. Warinschi — A linear space algorithm for computing the Hermite normal form. *In ISSAC*, p. 231–236, 2001.
Cité à la p. 98.

[OM10] S. Oh & B.-R. Moon — Automatic reproduction of a genius algorithm: Strassen's algorithm revisited by genetic search. *Trans. Evol. Comp* **14** (2010), n° 2, p. 246–251.
Cité à la p. 129.

[Pan84] V. Pan — *How to multiply matrices faster*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.
Cité à la p. 12.

[Per01] C. Pernet — Implementation of Winograd's fast matrix multiplication over finite fields using ATLAS level 3 BLAS. Rapport technique, Laboratoire Informatique et Distribution, juillet 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.
Cité aux p. 4 et 13.

[PH08] L. Plagne & F. Hülsemann — BTL++: From performance assessment to optimal libraries. *In Proceedings of the 8th international conference on Computational Science, Part III*, ICCS '08, p. 203–212, Berlin, Heidelberg, 2008. Springer-Verlag.
Cité à la p. 107.

[PS10] C. Pernet & W. Stein — Fast computation of hermite normal forms of random integer matrices. *Journal of Number Theory* **130** (2010), n° 7, p. 1675–1683.
Cité à la p. 98.

[RT92] J.-L. Roch & D. Trystram — Parallel Winograd multiplication. *In* W. Joosen & E. Milgrom, éditeurs —*EWPC'92, Parallel Computing: from theory to sound practice*, p. 578–581, Barcelona, 1992. IOS Press.
Cité à la p. 38.

[SA05] H. Sutter & A. Alexandrescu — *C++ Coding Standards: 101 Rules, Guidelines, And Best Practices*. The C++ In-Depth Series. Addison-Wesley, 2005.
Cité à la p. 78.

[Set73] R. Sethi — Complete register allocation problems. *In Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, p. 182–195, New York, NY, USA, 1973. ACM.
Cité à la p. 16.

[SL98] J. G. Siek & A. Lumsdaine — The matrix template library: A generic programming approach to high performance numerical linear algebra. *In International Symposium on Computing in Object-Oriented Parallel Environments*, Lecture Notes in Computer Science **1505**, p. 59–70, 1998.
Cité à la p. 77.

[SL00] J. G. Siek & A. Lumsdaine — Concept checking: Binding parametric polymorphism in C++. *In Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.
Cité à la p. 118.

[Smi02] W. D. Smith — Fast matrix algorithms and multiplication formulae. Rapport technique, Temple University, 2002.
Cité à la p. 39.

[Sto94] A. Storjohann — Computation of hermite and smith normal forms of matrices. Mémoire de Master., University of Waterloo, 1994.
Cité à la p. 98.

[Str69] V. Strassen — Gaussian elimination is not optimal. *Numerische Mathematik* **13** (1969), p. 354–356.
Cité à la p. 11.

[Str73] V. Strassen — Vermeidung von Divisionen. *J. Reine Angew. Math.* **264** (1973), p. 184–202.
Cité à la p. 40.

[Str94] B. Stroustrup — *The design and evolution of C++*. Programming languages/C++. Addison-Wesley, 1994.
Cité aux p. 78 et 79.

[SVC03] P. Stathis, S. Vassiliadis & S. Cotofana — A hierarchical sparse matrix storage format for vector processors. *In in Proceedings of IPDPS 2003*, p. 61, 2003.
Cité à la p. 57.

[SWY11] B. D. Saunders, D. H. Wood & B. S. Youse — Numeric-symbolic exact rational linear system solver. *In Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ISSAC '11, p. 305–312, New York, NY, USA, 2011. ACM.
Cité à la p. 99.

[TOU02] S.-A.-A. TOUATI — *La consommation en registres en présence de parallélisme d'instructions*. Thèse de doctorat, Université de Versailles-Saint Quentin en Yvelines, Inria Rocquencourt, juin 2002.
Cité à la p. 21.

[TS06] P. Tvrdik & I. Simecek — A new approach for accelerating the sparse matrix-vector multiplication. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on* **0** (2006), p. 156–163.
Cité aux p. 50 et 61.

[Tur06] W. J. Turner — A block Wiedemann rank algorithm. *In* J.-G. Dumas, éditeur —*Proceedings of the 2006 ACM International Symposium on Symbolic and Algebraic Computation, Genova, Italy*, p. 332–339. ACM Press, New York, juillet 2006.
Cité à la p. 69.

[TWBS09] D. Tsafrir, R. W. Wisniewski, D. F. Bacon & B. Stroustrup — Minimizing dependencies within generic classes for faster and smaller programs. *In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, p. 425–444, New York, NY, USA, 2009. ACM.
Cité à la p. 108.

[UEKM96] Úlfar Erlingsson, E. Kaltofen & D. Musser — Generic gram-schmidt orthogonalization by exact division. *In Proceedings of the 1996 international symposium on Symbolic and algebraic computation*, ISSAC '96, p. 275–282, New York, NY, USA, 1996. ACM.
Cité à la p. 108.

[VCS00] S. Vassiliadis, S. Cotofana & P. Stathis — BBCS based sparse matrix-vector multiplication: Initial evaluation. *In 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, p. 1–06, 2000.
Cité à la p. 57.

[VDY05] R. Vuduc, J. W. Demmel & K. A. Yelick — OSKI: A library of automatically tuned sparse matrix kernels. *In Institute of Physics Publishing*, 2005.
Cité à la p. 50.

[VGMF09] F. Vazquez, E. M. Garzon, J. A. Martinez & J. J. Fernandez — The sparse matrix vector product on GPUs. *Technical Report* (2009).
Cité aux p. 52 et 55.

[VM05] R. W. Vuduc & H.-J. Moon — Fast sparse matrix-vector multiplication by exploiting variable block structure. *In* L. T. Yang, O. F. Rana, B. D. Martino & J. Dongarra, éditeurs —*HPCC*, vol. 3726 de *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, p. 807–816. Springer-Verlag Inc., 2005.
Cité aux p. 50, 56 et 61.

[Wan06] Z. Wan — An algorithm to solve integer linear systems exactly using numerical methods. *J. Symb. Comput.* **41** (2006), p. 621–632.
Cité à la p. 99.

[Wie86] D. H. Wiedemann — Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory* **32** (1986), n° 1, p. 54–62.
Cité à la p. 69.

[Win71] S. Winograd — On multiplication of 2x2 matrices. *Linear Algebra and Application* **4** (1971), p. 381–388.
Cité aux p. 11 et 12.

[WPD01] R. C. Whaley, A. Petitet & J. J. Dongarra — Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* **27** (2001), n° 1–2, p. 3–35.
Cité à la p. 101.

Titre : Multiplication matricielle efficace et conception logicielle pour la bibliothèque de calcul exact LinBox.

Résumé : Dans ce mémoire de thèse, nous développons d'abord des multiplications matricielles efficaces. Nous créons de nouveaux ordonnancements qui permettent de réduire la taille de la mémoire supplémentaire nécessaire lors d'une multiplication du type Winograd tout en gardant une bonne complexité, grâce au développement d'outils externes *ad hoc* (jeu de galets), à des calculs fins de complexité et à de nouveaux algorithmes hybrides. Nous utilisons ensuite des technologies parallèles (multicœurs et GPU) pour accélérer efficacement la multiplication entre matrice creuse et vecteur dense (SpMV), essentielles aux algorithmes dits *boîte noire*, et créons de nouveaux formats hybrides adéquats.
Enfin, nous établissons des méthodes de *design* générique orientées vers l'efficacité, notamment par conception par briques de base, et via des auto-optimisations. Nous proposons aussi des méthodes pour améliorer et standardiser la qualité du code de manière à pérenniser et rendre plus robuste le code produit. Ces méthodes sont appliquées en particulier à la bibliothèque de calcul exact LinBox.

Mots-clés : algèbre linéaire exacte, matrices creuses, SpMV, matrices denses, multiplication matricielle dense, jeu de galet, ordonnancements, patrons de conception, bibliothèque mathématique générique.

Title: Efficient matrix multiplication and design for the exact linear algebra library LinBox.

Abstract: We first expose in this memoir efficient matrix multiplication techniques. We set up new schedules that allow us to minimize the extra memory requirements during a Winograd-style matrix multiplication, while keeping the complexity competitive. In order to get them, we develop external tools (pebble game), tight complexity computations and new hybrid algorithms. Then we use parallel technologies (multicore CPU and GPU) in order to accelerate efficiently the sparse matrix–dense vector multiplication (SpMV), crucial to *blackbox* algorithms and we set up new hybrid formats to store them.
Finally, we establish generic design methods focusing on efficiency, especially via building block conceptions or self-optimization. We also propose tools for improving and standardizing code quality in order to make it more sustainable and more robust. This is in particular applied to the LinBox computer algebra library.

Keywords: exact linear algebra, sparse matrix, SpMV, dense matrix, fast matrix multiplication, pebble game, schedulings, design patterns, generic mathematics library.