Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis

Hamza Jeljeli

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis

THÈSE

présentée et soutenue publiquement le 16 juillet 2015

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Hamza Jeljeli

Composition du jury

Rapporteurs :	Francisco Rodríguez-Henríquez	Professeur CINVESTAV, Mexico
	Gilles Villard	Directeur de recherche CNRS, LIP, ENS Lyon
Examinateurs :	Sylvain Collange	Chargé de recherche INRIA, Rennes
	Laura Grigori	Directrice de recherche INRIA, Paris-Rocquencourt
	Bruno Lévy	Directeur de recherche INRIA, Nancy
Directeurs :	Emmanuel Thomé	Chargé de recherche INRIA, Nancy
	Jérémie Detrey	Chargé de recherche INRIA, Nancy

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

À mes chers Chedlia et Abdelmajid.
Pour tout ce que vous n'avez cessé de me donner depuis tant d'années...

Remerciements

Tout d'abord, je tiens à remercier Gilles Villard et Francisco Rodríguez-Henríquez pour avoir accepté la relecture de ce manuscrit et pour m'avoir guidé lors de la finalisation de ce travail. Mes remerciements vont également à Laura Grigori et à Sylvain Collange pour avoir bien voulu participer à ce jury. Je remercie également Bruno Lévy pour avoir été mon référent interne et membre de mon jury de thèse.

Je tiens à adresser ma plus grande reconnaissance à Emmanuel Thomé et à Jérémie Detrey, sans qui ce travail n'aurait pas pu être réalisé. Manu et Jérémie, merci pour la confiance que vous m'avez accordée, pour toutes les heures que vous avez consacrées à l'encadrement et à l'orientation de ce travail. Merci pour les relectures scrupuleuses de mes écrits et pour les conseils et critiques qui ont fait avancer mon travail. J'aimerai saluer votre disponibilité et vos qualités d'écoute et de compréhension !

Je souhaite exprimer ma gratitude à l'ensemble de mes instituteurs et mes professeurs pour toutes les valeurs et tous les enseignements qu'ils ont su me transmettre sur le plan scientifique et humain. J'ai une pensée particulière pour M. Aouadi, qui m'a encouragé à aller de l'avant et à qui je dois ma passion pour les mathématiques.

Je souhaite exprimer ma reconnaissance à mes professeurs de l'Ensimag, en particulier Jean-Louis Roch, qui à travers ses cours, m'a donné goût à la cryptographie et m'a fourni des conseils précieux et des informations utiles. Je remercie aussi Clément Pernet, qui m'a permis de faire mon premier projet de recherche.

Par ailleurs, je remercie Arjen Lenstra et Marcelo Kaihara, qui m'ont accordé l'opportunité d'effectuer mon stage de Master au sein du laboratoire LACAL, m'initiant ainsi à la programmation sur les cartes graphiques, ce qui m'a été d'une grande utilité pour mener ces travaux.

Je souhaite remercier l'Université de Lorraine et l'école doctorale IAEM qui m'ont accordé le financement nécessaire à la réalisation cette thèse. Je souhaite également remercier la direction et l'ensemble des membres du laboratoire Loria pour les conditions de travail privilégiées qui m'ont été offertes. Je tiens à remercier Karine Jacquot, Emmanuelle Deschamps, Sophie Drouot, Vanessa Binnet et Françoise Laurent qui ont pris le soin de m'expliquer les procédures d'inscription et d'installation et qui m'ont simplifié les tâches administratives complexes et la paperasse inévitable.

Je remercie chaleureusement Pierre-Etienne Moreau de m'avoir donné la possibilité d'enseigner aux mineurs. Je remercie également l'ensemble des enseignants du département Informatique et Sciences du Numérique de l'École des mines de Nancy, particulièrement, Guillaume Bonfante, Alain Tisserant, Pascal Vaxifiere, Dominique Benmouffek, Jean-Yves Marion, Olivier Ageron et Karën Fort.

Mes remerciements vont à l'ensemble de l'équipe CARAMEL pour leur accueil qui m'a fait sentir dès mon arrivée inclus dans l'équipe.

Un grand merci à Pierrick Gaudry, Paul Zimmermann, Marion Videau et Pierre-Jean Spaenlehauer. Pendant les quatre années que j'ai passées avec eux, les « vieux » ont su mettre à disposition des « jeunes » les conditions favorables à leur réussite et à leur évolution. Par leurs conseils, leurs critiques, les « trolls » de la pause café et l'excellente atmosphère qu'ils ont su créer, ils ont tous contribué à l'aboutissement de ce travail.

Mes remerciements vont également aux autres « Caramels », anciens et actuels. Je pense à Nicolas Estibals, Alexander Kruppa, Luc Sanselme, Hugo Labrande, Maike Massierer, Sorina Ionica, Masahiro Ishii, Nicholas Coxon et à mon cher partenaire de natation Stéphane Glondu.

Je remercie spécialement mes chers coburreaux : Razvan Barbulescu, Lionel Muller, Cyril Bouvier et les plus « jeunes » Laurent Grémy et Svyatoslav Covanov. Ils sont devenus plus que des collègues. Merci Razvan Julien pour toutes les explications sur la machinerie de FFS et de NFS, pour les soirées pizza et pour tous les moments que nous avons partagés ! Laurent, un grand merci pour avoir relu ce manuscrit ! Merci à toi et à Svyat d'avoir toujours ramené les trolls et la bonne humeur dans notre bureau, via XMPP ou dans la salle de ping-pong !

Je remercie Laurent Imbert, Pascal Giorgi et Bastien Vialla qui m'ont accueilli au LIRMM pendant un séjour enrichissant et agréable.

Je remercie tous les copains du pique-nique, en particulier, Meriem, Aurélien, Hugo, Hubert, Walid, Éric, Ludovic, Jordi et tous les autres doctorants que j'ai oubliés. Je garderai un bon souvenir de ces formidables moments de bavardages, de taquineries et de rires.

J'adresse mes derniers remerciements à ma famille et à mes amis en Tunisie et en France. Je remercie mes chers parents, Selma, Bilel et Taoufik pour leur présence, leur soutien affectif et leurs encouragements. Papa, Maman, pendant vingt-trois ans, vous n'avez épargné ni effort ni santé pour que je puisse avoir toutes les chances pour réussir mon cursus ! J'espère qu'aujourd'hui, l'aboutissement de ce travail couronnera vos sacrifices !

Sommaire

Introduction générale

Objectifs de la thèse

La cryptologie est la science qui étudie la sécurité de l'information et de la communication. Cette discipline date de presque quatre millénaires [1] et a été historiquement liée à des contextes militaires et diplomatiques. Aujourd'hui, avec les progrès numériques rapides, l'utilisation des outils cryptographiques s'est démocratisée et est devenue omniprésente dans les échanges numériques qu'on effectue de manière naturelle, par exemple lorsqu'on fait un paiement en ligne, ou quand on s'authentifie pour accéder à un bâtiment ou pour élire un représentant dans une élection électronique.

Le terme cryptologie désigne à la fois la cryptographie et la cryptanalyse. La cryptographie est la discipline qui s'intéresse à comment concevoir des algorithmes et des protocoles qui protègent l'information et la communication. La cryptanalyse est la discipline qui cherche à vérifier la robustesse de ces algorithmes et de ces protocoles. Le travail exposé dans ce mémoire s'inscrit dans un cadre cryptanalytique où l'on cherche à évaluer la sécurité de certaines primitives cryptographiques. Ces primitives sont paramétrées par une clé. Le niveau de sécurité fourni par une primitive, c'est-à-dire la difficulté pour un attaquant de compromettre la fonctionnalité cryptographique fournie par la primitive, dépend de la taille de la clé. Plus la taille de la clé augmente, plus la complexité d'attaquer la primitive augmente. Le travail d'un cryptanalyste consiste à vérifier que la sécurité pratique (calculatoire) correspond à celle annoncée par le cryptographe. Cette approche peut être réalisée en effectuant des calculs (ou attaques) qui vont cibler des tailles de clés de plus en plus grandes. Même si, dans certains cas, ces tailles sont plus petites que celles utilisées en pratique dans les implémentations, ces calculs permettent de mesurer la sécurité calculatoire et de la comparer avec celle souhaitée.

En cryptographie, il existe deux branches principales, celle dite symétrique (ou à clé secrète) et celle dite asymétrique (ou à clé publique). La première est celle qu'on vulgarise par l'analogie du « coffre-fort » où les deux parties qui communiquent partagent un secret qu'ils utilisent pour « protéger » leurs échanges. La seconde est celle qu'on vulgarise par la boîte aux lettres, où chaque partie possède un couple de clés : une clé publique qui correspond à la boîte aux lettres et une clé privée qui correspond à la clé de la boîte aux lettres. Pour « protéger » son message, un expéditeur utilise sa clé privée et/ou la clé publique du destinataire, ceci dépend de la fonctionnalité cryptographique qu'il souhaite mettre en place. Les systèmes reposant sur les deux types de cryptographie ont des avantages et des inconvénients. Ils sont relativement complémentaires, c'est pourquoi dans certains protocoles, les deux types sont combinés.

Le cadre de cette thèse est celui de la cryptographie à clé publique. La sécurité des primitives de la cryptographie à clé publique repose sur la difficulté supposée de résoudre certains problèmes

1. Le plus vieux document chiffré découvert date du 17^{e} siècle avant notre ère, en Mésopotamie.

mathématiques. Parmi les problèmes les plus connus, on trouve la factorisation des entiers sur laquelle repose le cryptosystème Rivest-Shamir-Adleman (RSA) et le logarithme discret sur lequel repose le protocole Diffie-Hellman (DH) et que nous considérons ici.

Plus spécifiquement, on s'intéresse à la cryptanalyse du problème du logarithme discret dans les sous-groupes multiplicatifs des corps finis. Les algorithmes utilisés dans ce contexte, en particulier les algorithmes de calcul d'index, nécessitent de résoudre d'importants systèmes linéaires creux définis sur des corps finis de grande caractéristique. Cette étape dite d'« algèbre linéaire » représente dans beaucoup de cas le goulot d'étranglement qui empêche de cibler des tailles de corps finis plus grandes. En effet, résoudre de tels systèmes nécessite des calculs longs, qui consomment beaucoup de mémoire et qui requièrent des ressources importantes.

L'objectif de cette thèse est d'explorer les éléments qui permettent d'accélérer l'algèbre linéaire issue des calculs de logarithmes discrets. Pour réaliser cet objectif, nous sommes amenés à considérer plusieurs aspects et à intervenir aux niveaux algorithmique, arithmétique et architectural :

— trouver les architectures pensées pour le calcul parallèle et qui sont appropriées aux spécificités du problème considéré ;
— réfléchir sur la distribution du calcul d'algèbre linéaire sur plusieurs unités de calcul, aussi bien dans le cas d'un cluster de machines, que dans le cas d'une machine parallèle ;
— adapter les algorithmes classiques d'algèbre linéaire de sorte à tirer avantage à la fois de l'accélération fournie par des architectures parallèles et des spécificités du problème ;
— réfléchir sur un format de représentation de la matrice creuse, qui minimise l'utilisation de la mémoire et qui soit adapté aux architectures considérées ;
— développer une arithmétique efficace pour les calculs sur des corps finis de grande caractéristique.

Les travaux décrits ici sont spécifiques au contexte du logarithme discret, dans le sens où les approches mentionnées et les choix considérés dépendent des caractéristiques des systèmes linéaires issus de ce type de calculs. Toutefois, un certain nombre de ces choix restent valables dans un contexte plus général d'algèbre linéaire creuse sur des corps finis.

Organisation du mémoire

Ce mémoire est composé de deux parties. Dans la première partie, nous effectuons un rappel sur le contexte du logarithme discret (chapitre 1) et un rappel sur les architectures matérielles et logicielles considérées et les modèles de programmation qui permettent de les utiliser (chapitre 2). La deuxième partie du mémoire est consacrée à la résolution des systèmes linéaires creux sur des corps finis de grande caractéristique. Il y a un parallèle entre les chapitres et les niveaux d'accélération et de parallélisation de l'algèbre linéaire :

— le chapitre 3 présente les algorithmes de résolution et montre comment l'utilisation des blocs dans ces algorithmes permet de distribuer le calcul en plusieurs calculs indépendants qui peuvent être effectués en parallèle ;
— le chapitre 4 discute de la parallélisation, sur plusieurs nœuds de calcul, de l'opération principale de l'algèbre linéaire : le produit matrice-vecteur ;
— le chapitre 5 traite du produit partiel sur un nœud en considérant différentes architectures (GPU, CPU), plus précisément on s'intéresse à comment représenter la matrice et comment implémenter le produit partiel sur ces architectures ;

— le chapitre 6 aborde les aspects arithmétiques, on cherche une représentation des corps finis qui soit adaptée à des architectures pourvues d'unités vectorielles et on étudie comment implémenter sur ces architectures l'arithmétique correspondante.

Le dernier chapitre illustre les choix et les techniques que nous avons considérés dans chaque niveau, sur des exemples qui correspondent à des records de calcul de logarithme discret dans lesquels nos implémentations ont été utilisées.

Première partie

Contexte et rappels

Chapitre 1

Logarithme discret en cryptographie

Ce chapitre introductif présente le problème du calcul du logarithme discret, qui est un problème central en cryptographie à clé publique. Dans ce chapitre, nous donnons un aperçu plutôt bref aussi bien des aspects cryptographiques que des aspects cryptanalytiques liés au calcul du logarithme discret. Nous partons d'un exemple d'application qui motive l'intérêt cryptographique pour étudier ce problème. Ensuite, nous définissons d'une manière formelle le problème du logarithme discret et les notions qui lui sont liées. Nous détaillons les propriétés requises pour les groupes utilisés ainsi que les applications dans lesquelles intervient ce problème. Dans un second temps, nous passerons à la cryptanalyse du problème du logarithme discret, en ciblant principalement les algorithmes de *calcul d'index* qui résolvent le logarithme discret sur les corps finis.

Sommaire

1.1 Problème de l'échange de clés

La clé est un outil central en cryptographie pour pouvoir mettre en place un protocole qui protège la communication entre deux parties. Dans le cadre de la cryptographie symétrique (également appelée cryptographie à clé secrète), les deux parties, que l'on appelle habituellement Alice et Bob, sont amenées à établir une clé secrète commune. On appelle ce problème le problème de l'échange de clé. Par le passé, les valises diplomatiques ou d'autres formes de canaux « sécurisés » ont été utilisées pour pouvoir transmettre la clé. De nos jours, il est pertinent de supposer qu'Alice et Bob n'ont accès qu'à un canal de communication non protégé. Nous supposons par ailleurs que les deux parties sont authentifiées de sorte à prévenir des attaques de type *Homme de milieu* (*Man in the middle* en anglais).

Diffie et Hellman ont proposé en 1976 une solution qui répond au problème de la création d'une clé commune [DH76]. De nos jours, le protocole d'échange de clés Diffie-Hellman est intégré dans différents navigateurs web et fait partie de plusieurs protocoles de sécurisation des échanges sur Internet tels que TLS (Transport Layer Security).

Le schéma de la figure 1.1 détaille le protocole qui permet à Alice et Bob de créer communément une clé. Une des deux parties, ici Alice, commence par choisir un groupe cyclique G de cardinal n, noté multiplicativement et engendré par g. Nous utilisons la notation $G = \langle g \rangle$ pour préciser que G est engendré par g. Alice choisit un entier $k_A \in [0, n-1]$, calcule g^{k_A} et l'envoie à Bob. Idem, Bob choisit un entier $k_B \in [0, n-1]$, calcule g^{k_B} et l'envoie à Alice. À la fin de l'échange, les deux parties peuvent alors construire la même clé $K = (g^{k_A})^{k_B} = (g^{k_B})^{k_A}$.

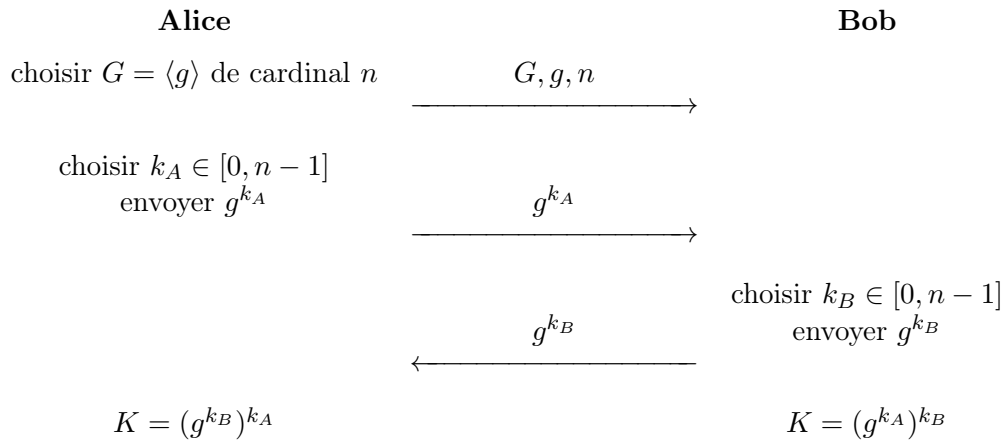


FIGURE 1.1 – Échange de clés Diffie-Hellman.

Un attaquant, qui écoute sur le canal, voit passer G, g, n, g^{k_A} et g^{k_B}. Il espère pouvoir deviner K à partir de ces observations. On appelle ce problème le problème Diffie-Hellman. Pour être en mesure de calculer K, l'attaquant pourrait espérer pouvoir calculer k_A ou k_B à partir de g^{k_A} et g^{k_B}, respectivement. Ce calcul correspond à une résolution du problème du logarithme discret que nous allons à présent définir plus en détail.

1.2 Problème du logarithme discret

1.2.1 Définitions

Reprenons le groupe cyclique G de cardinal n et engendré par g. L'opération d'exponentiation discrète est définie dans le groupe comme suit.

Définition 1.1 (Exponentiation discrète)**.** *Étant donnés le groupe G, un entier k dans $[0, n-1]$ et un élément $g \in G$, élever g à la puissance k consiste à calculer $g^k = \underbrace{g \times g \times \cdots \times g}_{k \text{ fois}}$.*

On note que l'exponentiation discrète, en utilisant la méthode d'exponentiation binaire, se calcule en $O(\log n)$ opérations dans G.

Le logarithme discret correspond à la réciproque de l'exponentiation. Le logarithme discret d'un élément du groupe est défini comme suit.

Définition 1.2 (Logarithme discret et problème du logarithme discret)**.** *Étant donnés un groupe G cyclique engendré par g et un élément h dans le groupe, le logarithme discret de h, en base g, qu'on note $\log_g h$, est l'unique entier k dans $[0, n-1]$ tel que*

$$h = g^k. \tag{1.1}$$

Le problème du logarithme discret consiste à calculer $k = \log_g h$ à partir de h et g.

Dès lors que la loi de groupe est calculable en temps polynomial en la taille des entrées, le problème de l'exponentiation discrète est dit « facile », c'est-à-dire, qu'il existe des algorithmes qui permettent de le résoudre en temps « court » avec des ressources de calcul limitées. Le problème inverse, celui du logarithme discret, est « difficile » dans certains groupes, c'est-à-dire nécessite un temps de calcul « long » même en utilisant un grand nombre de ressources. Les notions de « court » et « long » qu'on utilise ici sont évidemment relatives et seront clarifiées dans la suite quand nous allons préciser les complexités des opérations.

L'asymétrie entre l'exponentiation et le logarithme discret est utilisée en cryptographie à clé publique pour construire des *fonctions à trappe*, c'est-à-dire, des fonctions qui peuvent être aisément calculées lorsque la trappe est connue et dures à inverser sans connaissance de la trappe.

Dans l'exemple de l'échange de clés Diffie-Hellman, que nous avons précédemment évoqué, la sécurité du protocole repose sur la difficulté pour une troisième partie de pouvoir calculer des logarithmes discrets à partir des données échangées. Cette difficulté est formalisée dans la définition suivante.

Définition 1.3 (Problème Diffie-Hellman calculatoire (CDH))**.** *Étant donné le groupe G engendré par g, le problème CDH consiste à calculer $g^{k_A k_B}$ à partir de g, g^{k_A} et g^{k_B}.*

Il est clair que résoudre le problème du logarithme discret permettra à l'attaquant de résoudre le problème CDH. La réciproque est moins triviale [Mau94, MW96].

1.2.2 Groupes proposés

Les groupes proposés pour les applications cryptographiques doivent satisfaire les propriétés suivantes :

— les éléments du groupe peuvent être représentés à l'aide de $O(\log n)$ bits ;

— l'arithmétique dans le groupe G doit être efficace, en particulier l'exponentiation doit pouvoir être faite avec une complexité au plus polynomiale en la taille des entrées ;
— le problème du logarithme discret doit être aussi difficile que possible. Idéalement, la complexité des meilleurs algorithmes qui permettent de le résoudre devrait être exponentielle en la taille des entrées.

Il existe des groupes qui ne répondent pas à ces critères, tels que les groupes additifs $(\mathbb{Z}/n\mathbb{Z}, +)$, vu que le problème du logarithme discret dans ces groupes se résout avec l'algorithme d'Euclide, en temps polynomial en la taille des entrées.

Parmi les bons groupes au sens de la difficulté du problème du logarithme discret, nous trouvons les groupes multiplicatifs des corps finis $\mathbb{F}_{p^k}$ et les groupes des points de courbes elliptiques définies sur des corps finis. Ces deux familles de groupes satisfont les deux premières propriétés. La difficulté de résoudre le problème du logarithme discret varie en fonction du type de groupe. En effet, pour les corps finis, il existe des algorithmes pour résoudre le problème du logarithme discret avec une complexité sous-exponentielle ou quasi-polynomiale en la taille du corps, que nous allons détailler dans la section 1.4. Pour les courbes elliptiques, il existe aujourd'hui des algorithmes sous-exponentiels de cryptanalyse pour des familles particulières de courbes. Toutefois, il n'existe aucun algorithme de complexité sous-exponentielle pour calculer le logarithme discret sur une courbe elliptique générique.

1.2.3 Applications

Des primitives cryptographiques qui reposent sur la difficulté du logarithme discret ont été développées pour l'échange de clés, pour le chiffrement ou pour la signature.

Cryptosystème d'ElGamal. ElGamal a proposé en 1985 un système de chiffrement et de signature, reposant sur le problème du logarithme discret dans un corps fini [ElG85]. L'algorithme de signature est intégré dans la suite GnuPG (GNU Privacy Guard).

La signature DSA. L'algorithme DSA (Digital Signature Algorithm) est l'algorithme de signature standardisé par le NIST par le biais du standard DSS (Digital Signature Standard). Dans cet algorithme, un corps fini premier $\mathbb{F}_p$ est utilisé. Néanmoins, les opérations de calcul ne sont pas effectuées dans ${\mathbb{F}_p}^\times$, mais dans un sous-groupe d'ordre premier q qui divise $p-1$. Nous reviendrons dans la sous-section 1.4.1 sur cette idée de calculer dans un sous-groupe plutôt que dans tout le corps.

Les couplages.

Définition 1.4 (Couplage)**.** *Soient G_1, G_2 et G_T trois groupes cycliques de même cardinal. On note G_1 et G_2 additivement et G_T multiplicativement. Un couplage est une application bilinéaire*

$$e : G_1 \times G_2 \to G_T \tag{1.2}$$

qui vérifie les propriétés suivantes

1. *Non-dégénérescence :*

$$\begin{cases} \forall P \in G_1, \exists Q \in G_2 \text{ tel que } e(P,Q) \neq 1_{G_T} \\ \forall Q \in G_2, \exists P \in G_1 \text{ tel que } e(P,Q) \neq 1_{G_T} \end{cases}$$

2. *Bilinéarité :* $\forall P, P' \in G_1$ *et* $Q, Q' \in G_2$,

$$\begin{cases} e(P+P',Q) &= e(P,Q) \times e(P',Q) \\ e(P,Q+Q') &= e(P,Q) \times e(P,Q') \end{cases}$$

3. *Calculabilité : il existe un algorithme efficace pour calculer* e.

Les couplages ont été introduits en cryptographie par Menezes, Okamoto et Vanstone [MOV93] et par Frey et Rück [FR94] dans le cadre des attaques sur les courbes elliptiques supersingulières. Par la suite, différents systèmes cryptographiques qui reposent sur les couplages ont été proposés, parmi lesquels on peut citer l'échange de clés tripartite en un tour introduit par Joux [Jou00, Jou04], le chiffrement fondé sur l'identité de Boneh et Franklin [BF01] et de Sakai, Ohgishi et Kasahara [SOK00], le schéma de traçage de traîtres de Mitsunari, Sakai et Kasahara [MSK02] ou encore la signature courte de Boneh, Lynn et Shacham [BLS01]. Une étude d'un grand nombre de systèmes cryptographiques reposant sur les couplages a été réalisée par Dutta, Barua et Sarkar dans [DBS04].

En cryptographie à base de couplage, les groupes G_1 et G_2 correspondent à des groupes de points de courbes elliptiques ; le groupe G_T est un groupe multiplicatif d'un corps fini. La sécurité des systèmes utilisant les couplages nécessite que le problème du logarithme discret soit difficile dans les trois groupes.

1.2.4 Intérêt pour les logarithmes discrets

Le choix des cryptographes d'utiliser des systèmes reposant sur la difficulté du logarithme discret est motivé par plusieurs raisons. D'abord, ces systèmes répondent aux exigences de « bons » systèmes cryptographiques en terme de taille des clés et de niveau de sécurité atteint. De plus, ces systèmes représentent des alternatives aux systèmes dérivés de RSA qui reposent sur le problème de la factorisation d'entiers. Un troisième argument en faveur de l'étude des logarithmes discrets est que les couplages ne peuvent pas être réalisés avec RSA et ses variantes.

1.3 Algorithmes génériques

Dans un premier temps, nous allons voir certains algorithmes génériques de calcul de logarithme discret, c'est-à-dire des algorithmes qui s'appliquent sur n'importe quel groupe et qui ne tiennent pas compte des spécificités du groupe. Ces algorithmes sont plus efficaces que la recherche exhaustive qui a une complexité en $O(n)$, avec n le cardinal du groupe. Leurs complexités restent néanmoins exponentielles en la taille des entrées. Nechaev et Shoup ont même prouvé qu'il n'existe pas d'algorithme générique qui ait une complexité meilleure que $\Omega(\sqrt{n})$ lorsque n est premier [Nec94, Sho97]. Plus loin, dans la sous-section 1.4.1, nous allons voir une classe d'algorithmes non génériques qui sont spécifiques aux corps finis et qui permettent d'atteindre des complexités plus petites.

1.3.1 Baby-step–Giant-step

Cet algorithme a été inventé par Shanks en 1971 [Sha71]. En reprenant les notations précédentes, l'algorithme calcule le logarithme discret k d'un élément h de $G = \langle g \rangle$. L'idée est d'écrire k sous la forme de $c \times u + d$, pour $u = \lceil \sqrt{n} \rceil$. Le couple (c, d) est obtenu en trouvant une collision entre deux listes, une dite des « pas de bébé » et l'autre dite des « pas de géant ».

L'algorithme procède comme suit :

— on calcule $\mathcal{G} = \{g^d \text{ pour } d \in [0, u]\}$
— on calcule $\mathcal{B} = \{h(g^{-u})^c \text{ pour } c \in [0, n/u]\}$ et on s'arrête si $h(g^{-u})^c \in \mathcal{G}$.
— le logarithme de h est alors $c \times u + d$.

Le temps de calcul de l'algorithme *Baby-step–Giant-step* est dominé par le calcul des deux listes. C'est un algorithme déterministe qui a une complexité en temps en $O(\sqrt{n})$ en utilisant un espace mémoire en $O(\sqrt{n})$.

1.3.2 Pollard rho

L'algorithme *rho* a été introduit par Pollard en 1978 pour résoudre le problème du logarithme discret [Pol78] en adaptant l'algorithme du même nom pour la factorisation des entiers [Pol75]. C'est pourquoi il est assez commun de trouver dans la littérature la nomenclature *rho pour les logarithmes* pour distinguer cet algorithme de celui pour la factorisation.

Pour calculer le logarithme d'un élément h, l'algorithme *rho* repose sur la recherche de collisions dans les termes d'une séquence obtenue en appliquant une fonction pseudo-aléatoire f de G dans G sur des éléments de la forme $x_i = g^{a_i} h^{b_i}$.

Comme G est fini et que f est déterministe, la séquence obtenue contient un cycle. On s'attend à ce que la taille du cycle soit en $O(\sqrt{n})$. La recherche de collision est effectuée selon l'algorithme de Floyd pour la détection de cycle [Flo67]. Quand une collision est trouvée, nous avons une égalité de la forme $g^{a_1} h^{b_1} = g^{a_2} h^{b_2}$, pour des entiers connus a_1, b_1, a_2 et b_2. Cette égalité nous donne l'équation suivante :

$$(b_2 - b_1)k \equiv (a_1 - a_2) \pmod{n}$$

Si cette équation admet une solution (c'est-à-dire si $\text{pgcd}(b_2 - b_1, n) = 1$), alors k correspond au logarithme discret de h. Sinon, nous recommençons le calcul avec de nouvelles valeurs initiales pour les variables a_0 et b_0.

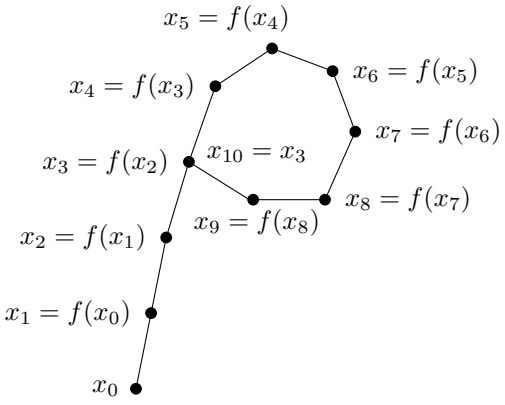


FIGURE 1.2 – Orbite typique de l'algorithme *rho* de Pollard.

L'algorithme *rho* de Pollard est probabiliste avec une complexité temps de $O(\sqrt{n})$ et une complexité mémoire de $O(\log n)$. C'est à ce jour l'algorithme générique le plus efficace en terme de temps et d'espace. Il existe un algorithme déterministe cousin de l'algorithme *rho*, souvent appelé l'algorithme *lambda* de Pollard, qui n'a pas une meilleure complexité, mais qui se parallélise mieux et qui est utile lorsque le logarithme recherché se trouve dans un intervalle limité [Pol78].

1.3.3 Pohlig-Hellman

L'algorithme de *Pohlig-Hellman*, parfois désigné aussi par méthode de *réduction de Pohlig-Hellman* permet de réduire le calcul du logarithme discret dans le groupe G en calculs de logarithmes discrets dans des groupes plus petits lorsque le cardinal de G n'est pas premier [PH78]. Supposons que nous connaissons la factorisation du cardinal du groupe G : $n = \prod_{i=1}^{r} p_i^{e_i}$, où $p_1, \ldots, p_r$ sont des nombres premiers et $e_1, \ldots, e_r$ sont les exposants correspondants. L'algorithme procède comme suit. Pour chaque i dans $\{1, \ldots, r\}$, on effectue e_i calculs de logarithme discret dans un sous-groupe isomorphe à $(\mathbb{Z}/p_i\mathbb{Z})$ (d'ordre p_i). Par la suite, le logarithme discret dans G peut être obtenu à partir des logarithmes calculés avec la reconstruction du théorème des restes chinois.

Le coût de la reconstruction, polynomial, est petit devant le coût du calcul du logarithme discret dans un sous-groupe. Par conséquent, la complexité de la résolution devient celle de la résolution du logarithme discret dans le plus grand sous-groupe d'ordre premier. Ainsi, combiner la méthode de *Pohlig-Hellman* avec la méthode *rho* de Pollard, va permettre de calculer des logarithmes discrets avec une complexité en $O\left(\sqrt{p}\right)$, où p est le plus grand diviseur premier de l'ordre n.

Le groupe G peut aussi avoir une structure particulière. Dans ces cas, exploiter la structure du groupe pour calculer le logarithme discret peut s'avérer une meilleure stratégie que d'appliquer la réduction de *Pohlig-Hellman*.

1.4 Logarithme discret dans les corps finis

1.4.1 Cadre du problème

Dans la suite de ce document, nous allons nous concentrer sur les groupes multiplicatifs des corps finis $\mathbb{F}_q$, où q est une puissance d'un nombre premier qu'on écrit sous la forme $q = p^k$. Suivant comment se comparent p et k, nous distinguons les corps de petite, moyenne ou grande caractéristique (voir figure 1.5) :

— Petite caractéristique : ce cas correspond à $\log p$ petit devant k.
— Moyenne caractéristique : ce cas correspond à $\log p$ de même ordre que k.
— Grande caractéristique : ce cas correspond à k petit devant $\log p$.

Nous allons aussi être amenés à considérer des cas plus particuliers tels que les corps premiers $\mathbb{F}_p$ ou les corps binaires $\mathbb{F}_{2^k}$.

À ce jour, le problème du logarithme discret dans les groupes multiplicatifs des corps finis est modérément difficile. Depuis quelques années, ce problème est la cible de plusieurs améliorations théoriques et pratiques.

Dans notre étude de la difficulté du logarithme discret dans les corps finis, l'objectif est de calculer le logarithme discret non pas vraiment dans le groupe $\mathbb{F}_{p^k}^{\times}$, mais dans un sous-groupe premier d'ordre suffisamment grand, qui est souvent le plus grand sous-groupe d'ordre premier de $\mathbb{F}_{p^k}^{\times}$. En effet, nous avons vu avec la technique de réduction de *Pohlig-Hellman* que résoudre le logarithme discret dans $\mathbb{F}_{p^k}^{\times}$ correspond à le résoudre dans son plus grand sous-groupe. Dans le contexte des applications, il est cryptographiquement pertinent de reposer sur la sécurité dans un sous-groupe suffisamment grand plutôt que de faire les calculs dans le groupe multiplicatif. Ainsi, si on cherche un « bon » corps fini $\mathbb{F}_{p^k}$, on va vérifier si $p^k - 1$ a un « grand » facteur premier, ce qui ne doit pas forcément être interprété comme le souhait que $(p^k - 1)/2$ soit premier. Par

« grand », nous voulons préciser qu'il doit résister à une attaque d'un algorithme générique, en l'occurrence Pollard rho.

1.4.2 Algorithmes de calcul d'index

Les groupes multiplicatifs des corps finis font partie des groupes pour lesquels les algorithmes de *calcul d'index* peuvent être utilisés pour résoudre le problème du logarithme discret. Ces algorithmes sont des adaptations des algorithmes de combinaison de congruences pour la factorisation des entiers. Le terme *calcul d'index* désigne la famille de ces algorithmes. En effet, ces algorithmes partagent un même schéma général, même si les détails relatifs aux étapes successives de ces algorithmes diffèrent et qu'ils n'utilisent pas toujours les mêmes outils et objets mathématiques.

Les algorithmes de calcul d'index résolvent le logarithme discret en un temps sous-exponentiel. Il est commun d'exprimer les complexités sous-exponentielles de ces algorithmes par le biais de la fonction sous-exponentielle $L_n(\alpha, c)$, introduite par Pomerance [Pom82].

Définition 1.5 (Fonction sous-exponentielle)**.** *Étant donnés les paramètres* $0 \leq \alpha \leq 1$ *et* $c > 0$*, la fonction sous-exponentielle est définie par*

$$L_n(\alpha, c) = \exp\left(c(1 + o(1))(\log n)^{\alpha}(\log\log n)^{1-\alpha}\right). \tag{1.3}$$

Le paramètre α est le paramètre principal de cette notation. Lorsque $\alpha = 0$, nous retrouvons une complexité polynomiale en la taille des entrées, c'est-à-dire en $O\left((\log n)^c\right)$. Si $\alpha = 1$, la complexité est exponentielle en la taille des entrées. Pour des valeurs de α intermédiaires, il s'agit de complexité sous-exponentielle.

L'idée principale des algorithmes de calcul d'index est de « construire » le logarithme discret d'un élément arbitraire à partir d'un ensemble de logarithmes discrets connus. La structure générale de ces algorithmes s'appuie sur trois étapes principales :

— l'étape de *crible*,
— l'étape d'*algèbre linéaire*,
— et l'étape de *logarithme individuel*.

La première étape cherche des relations linéaires entre les logarithmes discrets d'éléments appartenant à un sous-ensemble du groupe ; ce sous-ensemble est appelé *base de facteurs*. Quand suffisamment de relations ont été trouvées, les logarithmes des éléments de la base de facteurs peuvent être calculés pendant la deuxième phase en résolvant un système linéaire. Enfin, la troisième étape déduit le logarithme discret de n'importe quel élément du groupe à partir des logarithmes déjà calculés.

Afin de mieux clarifier les étapes des algorithmes de calcul d'index, nous allons commencer par présenter l'algorithme d'Adleman, le premier algorithme publié de calcul d'index dans le contexte du logarithme discret. Le choix de cet algorithme se justifie de par sa simplicité, mais aussi du fait que les algorithmes qui lui ont succédé peuvent être considérés comme des évolutions ou des optimisations de cet algorithme.

1.4.3 Algorithme d'Adleman

Cet algorithme a été inventé par Adleman en 1979 [Adl79]. Ici, nous présentons l'algorithme pour un groupe multiplicatif d'un corps fini premier $(\mathbb{F}_p)^{\times}$ engendré par g avec p un nombre premier. Néanmoins, la méthode a été généralisée à n'importe quel type de corps fini $\mathbb{F}_{p^k}$ [AD93]. L'algorithme est composé de 4 étapes :

Choix de la base de facteurs

La base de facteurs doit être un petit ensemble d'éléments du groupe tel qu'une grande proportion des éléments du groupe ont des facteurs premiers qui appartiennent tous à cette base. Ceci nous amène à définir la propriété de « friabilité ». Cette notion concerne aussi bien les entiers que les polynômes.

Définition 1.6 (Entier friable)**.** *Un entier est dit B-friable si tous ses facteurs premiers sont inférieurs à B.*

Définition 1.7 (Polynôme friable)**.** *Un polynôme est dit* B-friable *si sa factorisation en polynômes irréductibles ne comprend que des polynômes dont les degrés sont inférieurs ou égal à B.*

La borne B est appelée *borne de friabilité*. Dans l'algorithme d'Adleman, nous avons uniquement la friabilité des entiers. Une fois une borne de friabilité B choisie, la *base de facteurs* correspond à l'ensemble $\{-1\} \cup \{2, 3, \ldots, p_r\}$ contenant $r+1$ nombres premiers, tous inférieurs ou égaux à B.

Crible

Cette étape est aussi connue sous l'appellation de *collecte de relations*. Nous cherchons au moins $r+1$ relations linéairement indépendantes entre des éléments de la base de facteurs et des puissances de g, de la forme

$$g^x \equiv (-1)^{e_0} 2^{e_1} \ldots {p_r}^{e_r} \pmod{p}. \tag{1.4}$$

Nous avons besoin de tester la friabilité d'un élément et, si l'élément est friable, de pouvoir trouver sa factorisation. La factorisation d'un élément friable nous permet d'établir une relation. N'importe quel algorithme qui fournit un test de friabilité et la factorisation peut être utilisé. À titre d'exemple, nous pourrons mentionner l'algorithme *Trial division* ou des algorithmes qui sont plus rapides dans le cas d'une factorisation générale, en l'occurrence l'algorithme ECM (Elliptic Curve Method [Len87, BBB+12]) ou l'algorithme du crible quadratique [Pom84]. Toutefois, l'algorithme ECM est le plus efficace quand l'élément a de petits facteurs, ce qui est justement le cas lorsque celui-ci est friable.

Algèbre linéaire

En appliquant l'opération $\log_g$ sur l'équation 1.4, nous obtenons une relation qui implique les logarithmes des éléments de la base de facteurs et qui s'écrit sous la forme suivante

$$x \equiv e_0 \log_g(-1) + e_1 \log_g 2 + \cdots + e_r \log_g p_r \pmod{p-1}. \tag{1.5}$$

Nous formons avec les relations précédentes un système linéaire modulo $p-1$ et où les inconnues sont les logarithmes en base g des éléments de la base de facteurs. Le fait que nous disposions d'un nombre suffisant de relations va nous garantir de pouvoir calculer une solution du système. Résoudre le système correspond à calculer un élément non trivial du noyau de la matrice qui correspond au système. Une fois le système résolu, nous connaissons le logarithme discret de chaque élément de la base de facteurs.

Logarithme individuel

Cette dernière étape est aussi connue sous le nom de *descente*. Elle consiste à obtenir le logarithme discret de n'importe quel élément h du groupe à partir des logarithmes déjà calculés.

Nous cherchons un élément friable sur la base de facteurs de la forme $g^x h$, pour un x arbitraire. Quand cet élément est trouvé, nous pouvons alors écrire :

$$g^x h \equiv (-1)^{f_0} 2^{f_1} \dots {p_r}^{f_r} \pmod{p}. \tag{1.6}$$

Le logarithme discret k de h est alors calculé comme suit :

$$k = (f_0 \log_g(-1) + f_1 \log_g 2 + \dots + f_r \log_g p_r - x) \bmod (p-1). \tag{1.7}$$

Remarques

Bien choisir la taille de la base de facteurs est important. En effet, avec une base petite, les éléments friables sont plus rares et la tâche pour les trouver va être coûteuse. Si la base est grande, il faudra chercher beaucoup de relations et le système linéaire à résoudre sera plus grand.

Les étapes de *choix de la base de facteurs*, de *crible* et d'*algèbre linéaire* ne dépendent pas de l'élément h. Par conséquent, si nous sommes amenés à calculer plusieurs logarithmes discrets, seule l'étape de *logarithme individuel* est calculée plusieurs fois ; les autres étapes peuvent être donc considérées comme du pré-calcul.

En équilibrant les temps de calcul des phases de *crible* et d'*algèbre linéaire*, qui sont les étapes les plus coûteuses, la complexité totale de l'algorithme d'Adleman est égale en $L_p(\frac{1}{2}, \sqrt{2})$.

L'algorithme d'Adleman a été le précurseur en permettant de résoudre le problème du logarithme discret dans les corps premiers $\mathbb{F}_p$, avec une complexité en $L_p(\frac{1}{2}, \sqrt{2})$. Plus tard, différentes variantes de cet algorithme sont apparues et ont permis de le généraliser vers les autres corps et d'améliorer la complexité théorique. De nos jours, dans le cas général des corps finis $\mathbb{F}_{p^k}$, en fonction de comment se compare k par rapport à $\log p$, nous aurons des « régions » ; dans chacune d'entre elles, un certain algorithme de calcul d'index avec une certaine complexité est le plus efficace. Ainsi, la difficulté du problème du logarithme discret varie en fonction de la « région » (voir figure 1.5). Tous ces algorithmes sont des algorithmes cousins de l'algorithme NFS pour la factorisation.

Dans le prochaine section, nous allons présenter l'algorithme du crible du corps de fonctions (FFS pour Function Field Sieve) et l'algorithme du crible algébrique (NFS pour Number Field Sieve). Le choix de ces deux algorithmes parmi le spectre des algorithmes de calcul d'index est principalement motivé par le fait que les systèmes linéaires considérés dans les travaux exposés dans ce mémoire sont issus de calculs effectués avec ces deux algorithmes.

Nous n'allons pas fournir une description complète des algorithmes FFS et NFS, mais plutôt discuter de leurs spécificités par rapport au schéma de l'algorithme d'Adleman et les objets mathématiques considérés dans ces algorithmes.

1.4.4 Crible du corps de fonctions et crible algébrique

Crible du corps de fonctions pour les corps $\mathbb{F}_{p^k}$ de petite caractéristique

Les corps de petite caractéristique, en particulier de caractéristique 2, sont d'une grande importance du fait que l'arithmétique avec des petits corps de base, en particulier $\mathbb{F}_2$, peut

être implémentée efficacement. Un corps $\mathbb{F}_{p^k}$ peut être représenté de plusieurs façons. Ici, nous représentons le corps comme $\mathbb{F}_p[t]/\varphi(t)$, avec $\varphi(t)$ un polynôme irréductible de $\mathbb{F}_p[t]$ de degré k. Le polynôme $\varphi(t)$ est appelé polynôme de définition.

L'algorithme FFS [Adl94, AH99, JL02] est apparu en 1994, après l'algorithme de Coppersmith [Cop84] qui a permis de descendre en dessous de $L(\frac{1}{2})$ pour le cas particulier des corps de caractéristique 2, avec la complexité $L_{2^n}(\frac{1}{3}, 1.41)$. FFS s'applique aussi bien au cas de la caractéristique 2, qu'au cas de la caractéristique différente de 2, avec la complexité $L_{p^k}(\frac{1}{3}, 1.53)$. FFS a été longtemps considéré comme l'algorithme le plus efficace dans les corps de petite caractéristique. À partir de la fin de 2012, une série d'améliorations ont réduit dans un premier temps la complexité à $L_{p^k}(\frac{1}{4})$ [Jou13a, Jou13b, GGM+13]. Par la suite, un nouvel algorithme, inventé par Barbulescu, Gaudry, Joux et Thomé, a montré qu'il était possible de calculer des logarithmes discrets dans des corps de petite caractéristique en un temps quasi-polynomial qui s'exprime en $(\log p^{dk})^{O(\log\log p^{dk})}$, pour un petit entier d choisi d'une manière appropriée [BGJ+14].

L'algorithme FFS conserve les même étapes que l'algorithme d'Adleman, et y ajoute une étape dite de *Sélection polynomiale*. La tâche de la sélection polynomiale consiste à déterminer un couple de polynômes à deux variables $f, g \in \mathbb{F}_p[t][x]$, tels que leur résultant $\mathrm{Res}_x(f, g)$ contienne dans sa factorisation un polynôme irréductible de degré k. Ce polynôme irréductible correspond au polynôme $\varphi(t)$. Le choix des polynômes f et g est d'une part utile pour la représentation du corps, mais aussi intervient dans la recherche de relations et dans le logarithme individuel.

Le crible pour FFS adopte une autre stratégie que l'algorithme d'Adleman pour la collecte de relations. L'idée est de représenter le corps fini de deux façons différentes, comme représenté dans le diagramme de la figure 1.3. On prend un élément ϕ dans $\mathbb{F}_p[t][x]$ qui s'écrit sous la forme $\phi(t, x) = a(t) - b(t)x$. Ce élément est envoyé dans les deux corps de fonctions $\mathbb{F}_p[t][x]/f(x)$ et $\mathbb{F}_p[t][x]/g(x)$. Le diagramme étant commutatif, ceci permet d'obtenir une égalité dans $\mathbb{F}_{p^k}$. Comme pour n'importe quel algorithme de calcul d'index, il nous faut définir un critère de friabilité pour obtenir des relations qui ne font intervenir que des éléments appartenant à une certaine base de facteurs. FFS définit un critère de friabilité dans les deux ensembles ; plus précisément, on cherche des éléments ϕ tels que les résultants $\mathrm{Res}_x(\phi, f)$ et $\mathrm{Res}_x(\phi, g)$, où $\phi(t, x) = a(t) - b(t)x$, soient friables par rapport à une borne. Le crible est alors sensible au choix des polynômes f et g, en particulier à la taille de leurs coefficients. Le diagramme de la figure 1.3 résume les structures mathématiques impliquées.

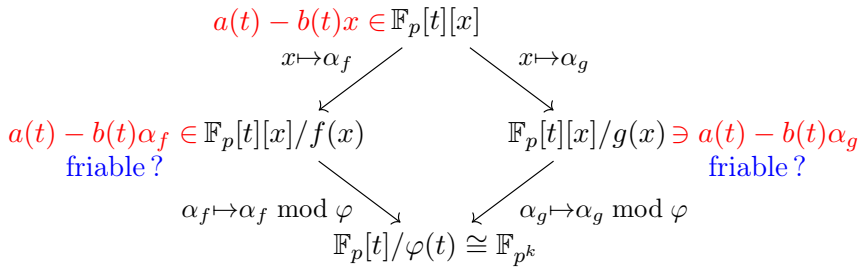


FIGURE 1.3 – Diagramme de crible pour FFS.

À l'issue de la phase de crible, nous retrouvons une phase d'algèbre linéaire, puis une phase dite de « descente » destinée au calcul d'un logarithme arbitraire. Ces phases sont très proches des procédures que nous avons décrites pour l'algorithme d'Adleman.

Crible algébrique pour les corps premiers $\mathbb{F}_p$

Les corps $\mathbb{F}_{p^k}^\times$, où p est grand, forment le domaine où le crible algébrique s'applique. Dans le cas particulier des corps premiers, ceci reste valide. L'algorithme du crible algébrique est une adaptation de l'algorithme du même nom pour la factorisation et hérite aussi de la structure générale d'un algorithme de calcul d'index pour le calcul des logarithmes discrets.

La première version de l'algorithme du crible algébrique pour le logarithme discret a été publiée par Gordon en 1993 [Gor93], puis plusieurs évolutions ont été proposées [SWD96, JL03]. Aujourd'hui, il a une complexité en $L_{p^k}(\frac{1}{3}, 1.9)$ [BP14, Sch00]. NFS est resté l'algorithme le plus efficace en grande caractéristique. Il existe une variante de NFS, appelée SNFS (Special Number Field Sieve), qui s'applique lorsque p est d'une forme spéciale. L'algorithme SNFS réduit la complexité de la résolution à $L_{p^k}(\frac{1}{3}, 1.53)$ [JP13, Sem02].

Le crible pour NFS suit la même approche que l'algorithme FFS, en utilisant un diagramme commutatif pour avoir deux représentations du corps $\mathbb{F}_p$. NFS choisit deux polynômes irréductibles f et g, à coefficients entiers, qui partagent une racine commune m dans $\mathbb{F}_p$. On suppose que g est degré 1. On associe aux polynômes f et g les deux anneaux $\mathbb{Z}[m]$ et $\mathbb{Z}[\alpha]$, où α est la racine de f dans le corps de nombres défini par f. Le crible cherche des polynômes de $\mathbb{Z}[x]$ de la forme $\phi = a - bx$ et tels que leurs résultants $\mathrm{Res}(\phi, f)$ et $\mathrm{Res}(\phi, g)$ soient friables. Dans le diagramme de la figure 1.4, nous schématisons ce procédé de collecte de relations. Dans le coté gauche, dit coté « rationnel », intervient une factorisation d'un nombre rationnel $a - bm$, alors que dans le coté gauche, dit coté « algébrique », il s'agit de décomposer l'idéal $(a - b\alpha)\mathcal{O}_{\mathbb{Q}(\alpha)}$, où $\mathcal{O}_{\mathbb{Q}(\alpha)}$ est l'anneau des entiers du corps de nombre $\mathbb{Q}(\alpha)$ défini par le polynôme f.

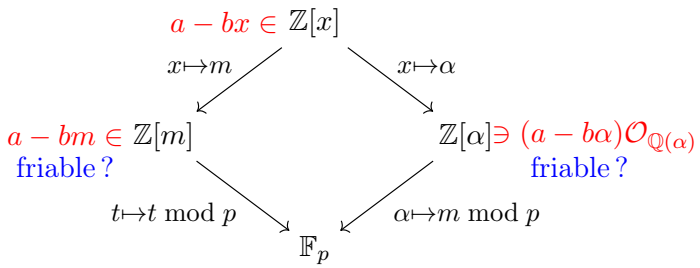


FIGURE 1.4 – Diagramme de crible pour NFS.

Une fois les relations collectées, on résout un système linéaire. À la fin, une descente suivant le schéma habituel est effectuée.

Comparaison des deux algorithmes de crible

Les algorithmes FFS et NFS partagent beaucoup de similitudes. L'algorithme FFS est plus rapide que l'algorithme NFS (voir les complexités dans la sous-section 1.4.6). En effet, lors de la sélection du polynôme f, il y a moins de contraintes avec l'algorithme FFS qu'avec l'algorithme NFS. Par conséquent, on arrive avec FFS à prendre un polynôme avec des coefficients petits, alors que ce n'est pas le cas avec l'algorithme NFS.

Les algorithmes FFS et NFS ne considèrent pas les mêmes objets mathématiques. Pendant le crible, FFS considère la factorisation des polynômes, alors que NFS considère la factorisation des entiers. Par conséquent, le test de friabilité de NFS est sous-exponentiel, alors que celui de

FFS est polynomial. Une autre différence majeure entre ces deux algorithmes concerne l'algèbre linéaire et plus spécifiquement, la nature des coefficients présents dans les systèmes linéaires. En effet, les systèmes linéaires issus de FFS contiennent des coefficients qui correspondent à des exposants et sont par conséquent de taille petite (inférieur à un mot machine), alors que les systèmes linéaires issus de NFS contiennent en outre des coefficients dont la taille est proche de celle de p. Ces coefficients plus grands correspondent aux caractères ℓ-adiques définis par Schirokauer [Sch93]. La présence de ces coefficients fait qu'à taille de système linéaire égale, l'algèbre linéaire correspondant à NFS est plus difficile qu'une algèbre linéaire correspondant à FFS. Nous reviendrons sur ce point plus loin dans la sous-section 3.2.1.

1.4.5 Problèmes difficiles

Dans cette sous-section, nous présentons les étapes calculatoirement difficiles dans les calculs des algorithmes de calcul d'index. Dans les algorithmes de calcul d'index apparus avant 2012, et dont la complexité est en $L(\frac{1}{3})$, les étapes de *crible* et d'*algèbre linéaire* sont les plus coûteuses en temps et en ressources. Dans les nouveaux algorithmes apparus après 2012 et dont la complexité est meilleure que $L(\frac{1}{3})$, l'étape de *logarithme individuel* devient aussi une étape coûteuse.

Crible

Le crible cherche des éléments qui vérifient une ou plusieurs conditions de friabilité par rapport à une ou plusieurs bornes données. Nous avons déjà vu avec les deux algorithmes FFS et NFS que cette étape est spécifique à chaque algorithme.

Pour pouvoir explorer l'espace des éléments à tester, il est commun d'utiliser les *réseaux euclidiens* (*lattices* en anglais) pour réduire l'espace de candidats potentiels [Pol93]. D'autres techniques plus spécifiques telles que les *special-*$\mathfrak{q}$ sont utilisées dans le contexte de l'algorithme FFS. Des informations sur les techniques d'accélération du crible pour l'algorithme FFS, ainsi que sur leurs apports en pratique peuvent être consultés dans [DGV13].

L'étape du crible est coûteuse. Dans les calculs récents de logarithmes discrets, le crible collecte des milliards d'éléments. Toutefois, cette étape possède l'avantage de bien se paralléliser. La recherche des relations peut être distribuée sur un grand nombre de machines. En effet, le grand espace dans lequel nous criblons peut être divisé en plusieurs sous-espaces, qui sont explorés en parallèle.

Algèbre linéaire

L'algèbre linéaire issue des algorithmes de calcul d'index consiste à résoudre un système d'équations linéaires homogènes. Le système est « grand » et « creux ». Par « creux », nous voulons dire que le nombre de coefficients non nuls dans une ligne est très faible. Pour donner un ordre de grandeur, il s'agit de systèmes qui contiennent des centaines de milliers voire des millions d'équations et où une équation contient uniquement quelques centaines de coefficients. Le système est défini dans $\mathbb{Z}/\ell\mathbb{Z}$, où ℓ est un grand nombre premier, qui correspond à l'ordre du groupe multiplicatif ou du sous-groupe multiplicatif dans lequel on regarde le logarithme discret.

L'étape d'algèbre linéaire a toujours été considérée comme le goulot d'étranglement des calculs de logarithme discret. On pourrait citer l'exemple du calcul mené par Gordon et McCurley dans $\mathbb{F}_{2^{503}}$ qui n'a pas abouti parce qu'ils n'arrivaient pas à résoudre le système linéaire [GM93]. L'algèbre linéaire est plus difficile dans le contexte du logarithme discret que dans le contexte

de la factorisation d'entiers. La différence est que, pour la factorisation, les systèmes sont définis sur $\mathbb{F}_2$, alors que pour le logarithme discret, les systèmes sont définis sur des grands corps finis. Les différents défis liés à cette algèbre linéaire sont la taille de la matrice, son caractère creux qui doit être bien exploité et l'arithmétique sur le grand corps fini. Pour faire face à ces défis, il est pertinent de pouvoir créer du parallélisme dans les calculs d'algèbre linéaire, car contrairement à l'étape précédente, il n'y a pas un parallélisme direct qui se dégage de la résolution des systèmes linéaires creux. Dans une grande partie de ce mémoire, nous allons détailler les défis et exposer différents approches et solutions utilisées pour accélérer cette étape.

Logarithme individuel

L'objectif de cette étape est d'exprimer l'élément dont on calcule le logarithme discret en fonction des éléments de la base de facteurs. L'approche consiste à exprimer un élément « grand » en fonction de plus petits éléments.

Comme les objets mathématiques varient entre les algorithmes de calcul d'index, il est difficile de décrire un schéma général de l'étape de *logarithme individuel*. Si on se place dans le cadre de l'algorithme FFS, la descente correspond à exprimer un polynôme de degré D avec des polynômes de degré $\sqrt{BD}$, où B est la borne de friabilité et à trouver une relation entre leurs logarithmes. Différentes stratégies de « descente » sont utilisées et combinées, telles que les fractions continues ou les bases de Gröbner.

Sélection polynomiale et filtrage

Dans les chantiers d'optimisation des algorithmes et des implémentations de calcul de logarithme discret, il est aussi important de mentionner les étapes de *sélection polynomiale* et de *filtrage*. Ces étapes ne sont pas calculatoirement difficiles, mais leurs résultats ont des impacts importants sur l'efficacité des étapes suivantes dans l'algorithme. La *sélection polynomiale* cherche des polynômes avec certaines propriétés qui influencent la difficulté de la recherche des relations [JL02, Bar13]. L'étape de *filtrage* agit sur les relations trouvées dans le *crible* de sorte à obtenir le plus petit système linéaire à résoudre [Bou13].

1.4.6 Panel des algorithmes de calcul d'index et leurs domaines de validité

Nous avons précédemment mentionné que, pour un corps fini donné $\mathbb{F}_{p^k}$, la taille de la caractéristique p va conditionner le choix d'un certain algorithme de calcul d'index. Plus précisément, les limites des régions de validité des algorithmes sont définies en fonction de comment se comparent k et p (voir figure 1.5).

p « petit »

On rappelle que ce cas correspond à $\log p$ petit devant k, exemple typique, $\mathbb{F}_{2^k}$. Avant 2012, FFS était l'algorithme le plus efficace dans cette région avec la complexité $L_{p^k}(\frac{1}{3}, 1.53)$. Actuellement, l'algorithme QPA (Quasi Polynomial Algorithm) de Barbulescu, Gaudry, Joux et Thomé, dont la complexité permet de calculer le logarithme discret avec une complexité quasi polynomiale. Cet algorithme et ses variantes sont les plus efficaces lorsque l'extension k a une forme particulière [BGJ+14]. Le cas où l'extension k est première représente le cas le plus difficile pour les nouveaux algorithmes et où FFS reste compétitif avec eux.

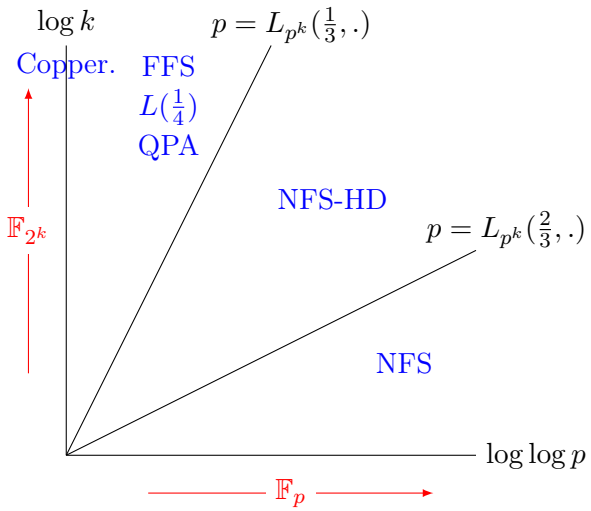


FIGURE 1.5 – Les domaines des algorithmes de calcul d'index pour $\mathbb{F}_{p^k}^\times$.

p « grand »

On rappelle que ce cas correspond à k petit devant $\log p$, exemple typique, $\mathbb{F}_p$. À ce jour, l'algorithme le plus efficace dans cette région est l'algorithme du crible algébrique (NFS), avec une complexité en $L_{p^k}(\frac{1}{3}, 1.9)$ [BP14]. La variante SNFS réduit la complexité à $L_{p^k}(\frac{1}{3}, 1.53)$ [JP13, Sem02].

p « moyen »

Entre les deux régions précédentes, nous trouvons le cas où $\log p$ et k sont de même ordre de grandeur. Dans cette région, une variante de l'algorithme du crible algébrique, dite NFS-HD, s'applique. Ce cas est connu pour être le plus difficile. Actuellement, la complexité de l'algorithme est descendue à $L_{p^k}(\frac{1}{3}, 2.24)$ [BP14, BGG+15].

1.4.7 Records de calcul de logarithme discret

Dans la table 1.1, nous recensons les derniers records de calcul de logarithme discret dans chaque domaine. Nous précisons aussi les données relatives aux records précédents : ainsi le lecteur verra quel est le « surcoût » de passer d'un calcul à un autre. Ce surcoût peut être en terme d'heures CPU si le même algorithme a été utilisé, mais peut aussi correspondre à l'utilisation d'un algorithme plus récent et plus efficace.

De l'observation de la table, il apparaît que le problème du logarithme discret dans les corps finis de petite caractéristique est devenu le moins difficile, grâce aux récentes améliorations théoriques. Dans le cas de la caractéristique 2, où l'extension est composée, le record est de 9234 bits, calculé avec l'algorithme $L(\frac{1}{4})$. Dans le cas où l'extension est un nombre premier, ce qui présente une difficulté plus importante, le record actuel est de 1279 bits, calculé avec une approche qui combine l'algorithme QPA et l'algorithme $L(\frac{1}{4})$, alors que le record précédent était de 809 bits calculé avec l'algorithme FFS. À ce jour, on situerait autour de 900 bits le seuil à partir duquel les nouveaux algorithmes battent FFS pour le cas où l'extension est un nombre premier. Pour la caractéristique 3, le record est de 3796 bits, calculé avec une variante de l'algorithme QPA.

Groupe	Date	Taille (bits)	Algorithme	Coût (h CPU)	Auteurs
$\mathbb{F}_{2^n}^\times$	01/2014	9234	$L(\frac{1}{4})$	398 k	R. Granger, T. Kleinjung et J. Zumbrägel [GKZ14]
	05/2013	6168	$L(\frac{1}{4})$	550	A. Joux [Jou13c]
$\mathbb{F}_{2^p}^\times$	10/2014	1279	QPA & $L(\frac{1}{4})$	35 k	T. Kleinjung [Kle14]
	04/2013	809	FFS	19.3 k	Équipe Caramel [BBD+13]
$\mathbb{F}_{3^n}^\times$	09/2014	3796	QPA	8.6 k	A. Joux et C. Pierrot [JP14]
	01/2014	1303	$L(\frac{1}{4})$	920	G. Adj, A. Menezes, T. Oliveira et F. Rodríguez-Henríquez [AMO+14]
p moyen	01/2013	1425	FFS	32 k	A. Joux [Jou13d]
	12/2012	1175	FFS	32 k	A. Joux [Jou12]
$\mathbb{F}_{p^2}^\times$	06/2014	529	NFS	1.92 k	R. Barbulescu, P. Gaudry, A. Guillevic et F. Morain [BGG+14]
$\mathbb{F}_p^\times$	06/2014	596	NFS	1.13 M	C. Bouvier, P. Gaudry, L. Imbert, H. Jeljeli et E. Thomé [BGI+14]
	02/2007	530	NFS	29 k	T. Kleinjung [Kle07]

TABLE 1.1 – Les records de calcul de logarithme discret sur les corps finis.

Les records pour les corps finis de moyenne caractéristique et à grande caractéristique sont respectivement de 1425 et de 596 bits. Ceci est dû au fait que dans ces régions, on ne connaît pas d'algorithmes qui ont une meilleure complexité que $L(\frac{1}{3})$.

Dans le chapitre 7, nous détaillerons deux calculs ; un premier relatif à un record en petite caractéristique sur une extension première et le second correspond à un record en grande caractéristique.

1.5 Conclusion

À travers ce chapitre, nous avons donné un aperçu du problème du logarithme discret dans les groupes pour lesquels il y a un intérêt cryptographique. Il apparaît que, depuis l'apparition du protocole d'échange de clé Diffie-Hellman il y a plus que trois décennies, le logarithme discret était l'objet d'une part d'avancées algorithmiques majeures mais aussi d'optimisations pratiques au niveau des implémentations. Aujourd'hui, ce problème est encore considéré comme difficile pour les groupes multiplicatifs des corps finis de moyenne et de grande caractéristique et pour les groupes additifs des courbes elliptiques et des courbes hyperelliptiques [Die11].

Chapitre 2

Calcul à hautes performances (HPC)

Ce chapitre est le chapitre « technologique » du mémoire, dans lequel nous allons introduire et décrire les concepts et les mécanismes liés aux architectures que nous allons utiliser pour accélérer le calcul d'algèbre linéaire. Dans un premier temps, nous allons répondre à la question : quelles sont les architectures pensées pour le calcul parallèle et qui répondent aux exigences des calculs que nous considérons ? Par la suite, nous allons décrire le modèle de programmation CUDA, qui permet d'utiliser les processeurs graphiques (GPU) NVIDIA pour accélérer une application. La dernière partie du chapitre sera consacrée à l'exécution du calcul sur un cluster. Plus spécifiquement, nous allons discuter les aspects liés aux communications, lorsque les nœuds de calcul sont des CPUs ou des GPU.

Sommaire

2.1 Calcul à hautes performances et algèbre linéaire

2.1.1 Architectures pensées pour le calcul parallèle

Avant l'apparition des premières plate-formes de calcul parallèle, un programme résolvant un problème correspondait à un calcul *séquentiel*. Dans ce cadre, le programme est composé d'une série d'instructions, exécutées sur une seule ressource, typiquement un processeur d'une machine mono-cœur. Une seule instruction est exécutée à la fois sur la ressource. Plus tard, les architectures ont permis la résolution *parallèle* du calcul, c'est-à-dire l'utilisation simultanée de plusieurs ressources. Le problème est alors divisé en différentes parties qui peuvent être résolues en parallèle. Chaque partie est transformée en une suite d'instructions. Chaque instruction est exécutée sur une ressource.

Aujourd'hui, nous trouvons un large ensemble d'architectures matérielles adaptées à la résolution parallèle, parmi lesquelles :

- Les processeurs multi-cœurs : ce sont les processeurs dont la puce contient au moins deux unités d'exécution (cœurs). Les configurations typiques sont 4, 6 ou 8 cœurs par puce.
- Les processeurs *many-cœurs* : ce sont des architectures multi-cœurs où le nombre de cœurs est de l'ordre de dizaines à quelques centaines, par exemple la famille des coprocesseurs Intel Xeon Phi.
- Les processeurs vectoriels : ce sont des processeurs pourvus de fonctionnalités de parallélisme de données ; c'est-à-dire qu'ils peuvent exécuter en parallèle une même instruction sur des données distinctes.
- Les processeurs graphiques (GPU) : les processeurs graphiques sont intégrés dans les cartes graphiques pour accélérer les traitements graphiques qui sont généralement hautement parallèles (*embarrassingly parallel*).
- Les circuits logiques configurables : ce sont des architectures reconfigurables, à l'image des FPGA, où le programmeur configure le circuit à l'aide de blocs logiques et de ressources de routage. Ces architectures permettent d'exploiter le parallélisme matériel.
- Les clusters de calcul : un cluster de calcul est un ensemble de machines interconnectées par un réseau de communication, généralement à haut débit.

Les architectures que nous avons décrites dans la liste précédente, qui n'est pas exhaustive, se distinguent de par le niveau du parallélisme qu'elles exploitent :

- instruction ;
- donnée ;
- mémoire ;
- tâche.

Les architectures parallèles sont généralement classées selon l'organisation des données, des processus et des instructions. Flynn a introduit dans sa taxonomie [Fly72] 4 modes d'exécution possibles :

- SISD (Single Instruction Single Data) : Ce mode correspond au fonctionnement traditionnel d'une machine mono-processeur qui n'exploite aucun parallélisme.
- SIMD (Single Instruction Multiple Data) : Une même instruction est effectuée en parallèle sur des données différentes. On trouve ce mode dans les processeurs vectoriels.
- MISD (Multiple Instructions Single Data) : Plusieurs instructions agissent en parallèle sur une même donnée. Il n'existe pas beaucoup d'architectures qui implémentent cette catégorie.

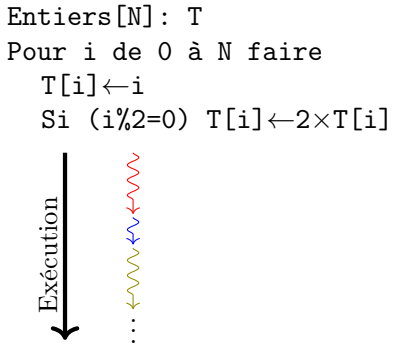
— MIMD (Multiple Instructions Multiple Data) : Ce mode est utilisé dans les machines multi-processeurs, où plusieurs processeurs exécutent différentes instructions sur différentes données.

Plus tard, cette classification a été complétée par les deux sous-catégories suivantes du mode MIMD :

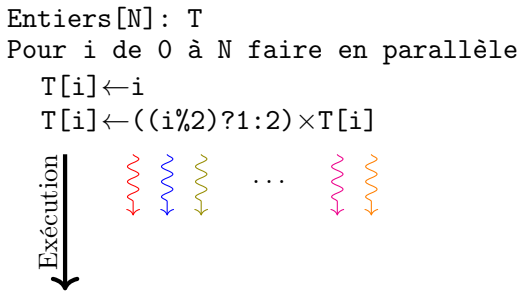
— SPMD (Single Program Multiple Data) : Un même programme (code) est exécuté par les processeurs, mais les instructions effectuées par chaque processeur peuvent varier. On trouve ce mode dans les processeurs graphiques.

— MPMD (Multiple Programs Multiple Data) : On rencontre ce mode par exemple sur un processeur bi-cœur, où on exécute un programme différent sur chaque cœur.

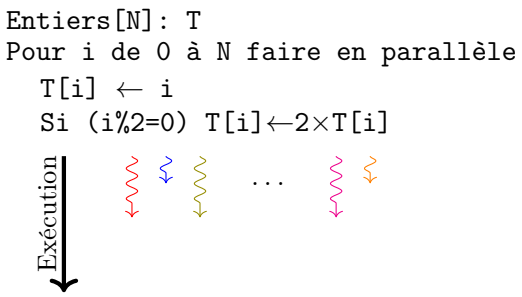
Dans la figure 2.1, nous illustrons le fonctionnement des modes SISD, SIMD et SPMD sur des programmes (écrits en pseudo-code), où on traite un tableau d'entiers. Dans la sous-figure 2.1a, il y a un traitement purement séquentiel (SISD). Dans la sous-figure 2.1b, les mêmes instructions sont effectuées en parallèle sur les éléments du tableau (SIMD). Dans la sous-figure 2.1c, le même programme n'exécute pas les mêmes instructions, parce que nous avons ajouté un test (SPMD).



(a) SISD



(b) SIMD



(c) SPMD

FIGURE 2.1 – Modes d'exécution : SISD, SIMD et SPMD.

2.1.2 Architectures appropriés à notre type d'algèbre linéaire

Dans cette section, nous allons réfléchir sur les caractéristiques que doit satisfaire une architecture appropriée à notre contexte applicatif d'algèbre linéaire. Nous avons mentionné à la sous-section 1.4.5 que notre problème est la résolution d'un système linéaire grand et creux sur un grand corps fini (voir section 3.2 pour plus de détails sur les caractéristiques des entrées). La

brique de base du calcul d'algèbre linéaire, si l'on utilise des algorithmes *boîte noire* comme ce sera le cas de ceux présentés au chapitre 3, est l'opération de multiplication de la matrice creuse par un vecteur ou par un bloc de vecteurs. Nous avons besoin d'une architecture qui exploite différents niveaux du parallélisme (voir section 3.3). L'architecture doit satisfaire les critères suivants :

1. La taille importante de la matrice et des vecteurs fait que nous avons besoin de pouvoir stocker et traiter d'une manière efficace des données dont les tailles peuvent atteindre quelques dizaines de gigaoctets.
2. La matrice est creuse, avec des zones très creuses. Ceci fait que les accès sur le ou les vecteurs d'entrée risquent d'être irréguliers. C'est pourquoi nous avons besoin de mémoires et de mécanismes de cache qui permettent de limiter les pénalités dues à l'irrégularité des accès.
3. Les opérations arithmétiques sont effectuées sur un grand corps fini. Il est important que l'architecture logicielle permette d'écrire des briques de calcul arithmétique spécifiques et efficaces. De plus, cette arithmétique peut être accélérée en utilisant une représentation qui exploite la vectorisation (voir chapitre 6).
4. Si nous envisageons de distribuer l'algèbre linéaire sur un cluster de machines, les calculs ne peuvent pas être complètement indépendants et sont donc amenés à partager des données de taille importante. C'est pourquoi nous avons besoin d'un réseau et de mécanismes de communication efficaces en terme de latence et de débit pour minimiser les coûts de communication. Nous pourrons aussi mettre en place un parallélisme de tâches, et dans ce cas, nous aurons besoin d'un modèle approprié.

À partir de cette description, nous concluons que les circuits configurables ne sont pas appropriés à cause de leur mémoire limitée (le critère 1 est non satisfait). Par contre, les CPU munis de plusieurs cœurs et d'unités vectorielles, ainsi que des accélérateurs de type GPU ou processeur *many-cœurs* se positionnent comme des architectures appropriées (les critères 1, 2 et 3 sont satisfaits ; le critère 1 peut être contraignant pour les GPU à partir de très grandes tailles).

Dans le cadre de ce travail, nous allons étudier l'utilisation des CPU multi-cœurs et vectoriels et des GPU pour le problème d'algèbre linéaire.

2.2 CPU multi-cœurs et vectoriels

Un CPU multi-cœurs est un processeur qui contient deux cœurs ou plus gravés sur la même puce. Le premier processeur multi-cœur est apparu au milieu des années 80 avec le CPU bi-cœur de Rockwell International. Dès le début des années 2000, différents constructeurs tels que IBM, AMD, Intel ont commencé à commercialiser des processeurs multi-cœurs. L'idée d'introduire plusieurs unités de calcul dans un même processeur permet d'implémenter physiquement le *multi-processing* et répond à la problématique du besoin continu d'augmenter la fréquence du CPU.

Un processeur vectoriel est un processeur pourvu d'instructions vectorielles, c'est-à-dire d'instructions appropriées au traitement parallèle de données d'un bloc de taillé fixée, qu'on appelle vecteur. Par le passé, les architectures vectorielles ont été développées pour les supercalculateurs, par exemple pour les machines Cray et ILLIAC. De nos jours, la vectorisation est exploitée dans les processeurs incorporant des instructions SIMD, par exemple dans tous les processeurs Intel à partir du Pentium III, dans les processeurs POWER de IBM et également dans le processeur CELL de la PlayStation 3.

Dans le cadre de ce travail, nous allons considérer les microprocesseurs de type x86 et les jeux d'instructions SIMD suivants :

— SSE (Streaming SIMD Extensions) qui fournit des registres 128 bits ;
— AVX (Advanced Vector Extensions) qui fournit des registres 256 bits.

Un registre SSE peut contenir 2 composantes 64 bits et exécuter deux instructions en parallèle, il peut aussi être configuré de sorte à contenir 4 composantes, sur 32 bits chacune. Un registre AVX peut contenir 4 composantes 64 bits ou 8 composantes 32 bits.

2.3 Programmation sur les GPU : Modèle CUDA

Dans ce travail, nous allons nous restreindre aux plate-formes NVIDIA et nous allons utiliser le modèle de programmation CUDA.

Le nom CUDA (Compute Unified Device Architecture) désigne l'architecture matérielle et logicielle spécifique aux GPU NVIDIA et qui permet d'exploiter leur capacité de traitement massivement parallèle pour des tâches plus génériques que les traitements graphiques et qui vont s'étendre à des applications de calcul scientifique, de simulation, etc. Cette orientation vers l'utilisation du processeur graphique pour des applications pas nécessairement graphiques est communément désignée par l'acronyme GPGPU (General-purpose Processing on Graphics Processing Units), qu'on traduirait par le calcul généraliste sur processeurs graphiques.

CUDA met à disposition de programmeurs grand public une plate-forme de calcul parallèle qui supporte conjointement les langages C, C++ et Fortran. Il existe aussi des interfaces pour d'autres langages telles que `PyCUDA` pour Python, `jCUDA` pour Java, etc. La description qui suit, ainsi que le travail présenté dans ce mémoire, sont basés sur le modèle de programmation de CUDA, parce que nous allons nous limiter aux plate-formes NVIDIA. Néanmoins, il est possible d'utiliser d'autres modèles de programmation, en l'occurrence celui de OpenCL [OpenCL]. OpenCL propose un modèle de programmation multi-plateforme qui cible plusieurs systèmes parallèles hétérogènes tels que les GPU (NVIDIA, AMD, Intel, etc), les CPU multi-cœurs et les processeurs CELL de la PlayStation 3.

Le modèle CUDA est assez largement décrit dans la littérature [SK10, CUDAa] et des plate-formes de documentation en ligne lui sont consacrées [2, 3].

2.3.1 Architectures considérées

Les cartes graphiques NVIDIA sont classées selon leur architecture. L'architecture correspond à des évolutions chronologiques de la structure et de l'organisation matérielles du GPU et du modèle de programmation qui permet de l'utiliser. Dans le cadre de ce travail, nous avons eu accès à différents modèles de cartes graphiques qui appartiennent à deux architectures :

— Fermi [Fermi] ;
— Kepler [Kepler].

Pour distinguer les révisions d'une architecture, NVIDIA utilise un identifiant numérique appelé compute capability, de la forme $x.y$, où x désigne l'architecture (par exemple 2 pour Fermi et 3 pour Kepler) et y désigne la révision [CUDAa]. À titre d'exemples, nous trouvons pour l'architecture Fermi les compute capabilities 2.0 et 2.1 et pour l'architecture Kepler les compute capabilities 3.0, 3.5 et 3.7. Les cartes que nous utilisons sont soit de compute capability

2. NVIDIA CUDA ZONE : https://developer.nvidia.com/cuda-zone
3. GTC On-Demand : http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php

2.0 soit de compute capability 3.0. Les mécanismes et les spécifications que nous allons décrire dans la suite référent à ces deux compute capability.

Nous utilisons les cartes suivantes :

— GeForce GTX 580 (Fermi, compute capability 2.0) ;
— Tesla M2050 (Fermi, compute capability 2.0) ;
— GeForce GTX 680 (Kepler, compute capability 3.0).

Il existe une autre classification des cartes graphiques NVIDIA selon la gamme : GeForce, Quadro et Tesla. La gamme GeForce est une gamme « grand public » destinée principalement au marché des jeux vidéo. La gamme Quadro est destinée aux « professionnels » et cible des applications de type CAO (Conception Assistée par Ordinateur). La gamme Tesla a été conçue pour des applications de calcul scientifique, en fournissant des fonctionnalités spécifiques telles que les codes ECC (Error Correction Codes) qui certifient l'exactitude des calculs et la fonctionnalité Direct GPU qui améliore les communications. Contrairement à la classification selon l'architecture, la classification selon la gamme est peu pertinente dans notre contexte.

2.3.2 Modèle d'exécution

Le CPU qui est connecté au GPU et qui le contrôle est appelé *hôte* (*host*). Le GPU est désigné par le terme *périphérique* (*device*). Un programme CUDA est composé d'un code exécuté sur l'*hôte* et d'un code exécuté sur le *périphérique*. Le code exécuté sur le périphérique est composé de fonctions, appelés *kernels*, qui lorsqu'elles sont appelées sont exécutées plusieurs fois en parallèle par plusieurs *threads* CUDA en suivant le mode SPMD (Single Program, Multiple Data).

L'exécution typique d'un programme CUDA qui lance un seul *kernel* est composée des actions suivantes, qui sont spécifiées explicitement par le programmeur dans le code, qui exploite à ces fins des appels de l'interface CUDA (voir figure 2.2) :

1. créer des données sur la mémoire de l'*hôte* ;
2. transférer les données vers la mémoire du *périphérique* ;
3. lancer un *kernel* depuis le CPU ;
4. exécuter le *kernel* en parallèle sur le GPU ;
5. synchroniser le GPU et le CPU ;
6. copier les résultats vers la mémoire du CPU.

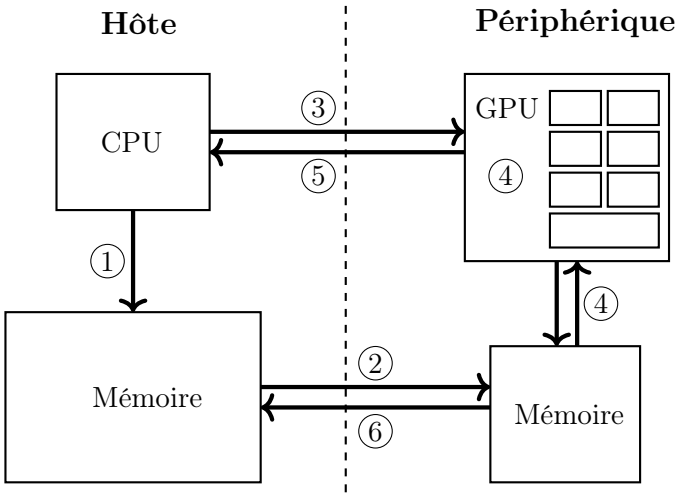


FIGURE 2.2 – Représentation chronologique de l'exécution d'un programme CUDA.

2.3.3 Hiérarchie GBT

Les *threads* sont organisés en *blocks* (blocs en français). Un ensemble de *blocks* forme un *grid* (grille en français). Cette hiérarchie est appelée la hiérarchie GBT (*Grid* de *Blocks* de *Threads*) (voir figure 2.3) :

— le *thread* est une instance du *kernel*. Il possède un identifiant unique dans son *block* ;
— le *block* est constitué de *threads* qui peuvent communiquer et se synchroniser. Un *block* a un identifiant unique dans son *grid* ;
— le *grid* est l'ensemble des *blocks* qui exécutent le même *kernel*.

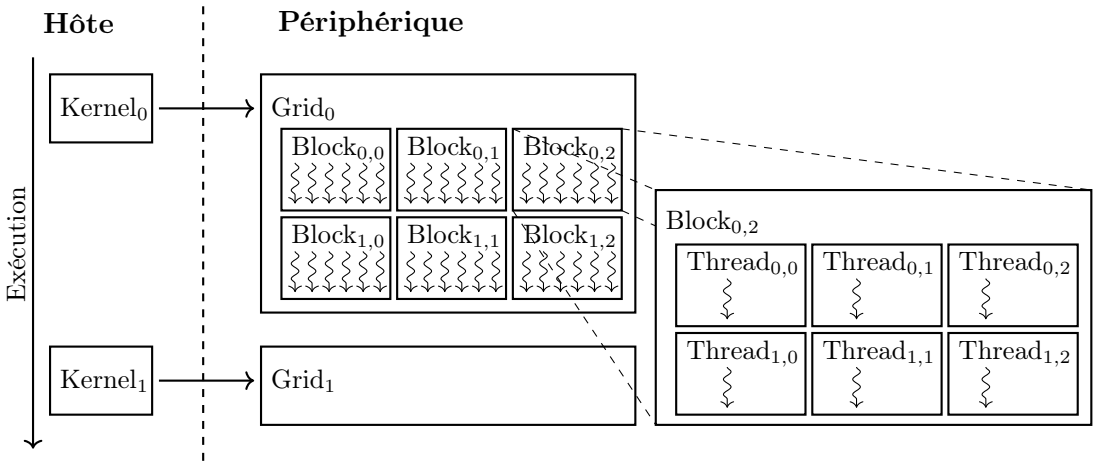


FIGURE 2.3 – Organisation du calcul selon la hiérarchie GBT.

Le programmeur est amené à définir les paramètres de dimension et de taille de la hiérarchie GBT en accord avec les spécifications de son application :

— la dimension du *grid* de *blocks* ? 1D, 2D ou 3D ;
— les longueur, largeur et hauteur du *grid* de *blocks* : x ?, y ? et z ? ($\max(x) = 65535$ pour Fermi et $\max(x) = 2^{31} - 1$ pour Kepler, $\max(y) = \max(z) = 65535$) ;
— la dimension du *block* de *threads* ? 1D, 2D ou 3D ;

— les longueur, largeur et hauteur du *block* de *threads* : x ?, y ? et z ? ($\max(x) = \max(y) = 1024, \max(z) = 64$ et $\max(threads) = 1024$).

L'ensemble de ces caractéristiques sont fondamentalement attachées au *kernel*.

2.3.4 Hiérarchie des cœurs

Le GPU comporte une grille composée d'un grand nombre de cœurs CUDA. Cette grille est appelée *Streaming Processor Array* (SPA) en vocabulaire CUDA. Les cœurs CUDA, aussi désignés par l'appellation *Streaming Processor* (SP), sont organisés en multiprocesseurs, qu'on appelle aussi *Streaming Multiprocessors* (SM pour Fermi et SMX pour Kepler). Le nombre de cœurs est de l'ordre du millier par carte, par exemple, la GeForce GTX 680 possède 1536 cœurs. Un SM contient 32 cœurs CUDA.

Lorsque le calcul d'un *kernel* est effectué sur la carte, les *blocks* sont distribués sur les SM disponibles. Un SM peut contenir simultanément jusqu'à 16 *blocks*. Dans une même génération de cartes graphiques, la puissance de calcul de la carte va être proportionnelle au nombre de SM. Un GPU avec deux fois plus de SM va mettre deux fois moins de temps pour exécuter un *kernel*.

Chaque SM groupe les *threads* appartenant à un même *block* en des ensembles de 32 *threads*, appelés *warps*. Le SM crée, ordonnance et exécute les *warps* sur ses différents cœurs. Chaque instruction du *kernel* est exécutée par les 32 *threads* du *warp* simultanément. Le mode d'exécution des *threads* du même *warp* est de type SIMD. Toutefois, il est possible que deux *threads* d'un même *warp* prennent deux branches différentes dans le code, on parle alors d'une divergence de code ou de divergence de branche. Dans ce cas, le paradigme SIMD n'est plus maintenu ; les deux branches ne sont pas exécutées en parallèle, mais l'une après l'autre, ce qui affecte négativement les performances.

Si nous cherchons à établir un parallèle entre la hiérarchie GBT et la hiérarchie des cœurs, le *grid* est exécuté sur le SPA ; il y a une analogie entre le *block* et le SM, même si un SM exécute plusieurs *blocks* ; et les *thread* correspondraient aux cœurs CUDA.

2.3.5 Hiérarchie de la mémoire

CUDA fournit plusieurs espaces mémoire qui se différencient de par leur taille, la vitesse de leurs accès, leur usage et les mécanismes de cache qui leur sont dédiés. Il existe des mémoires sur puce (*on-chip*), dont la taille est relativement petite et dont les accès sont rapides, et des mémoires hors-puce (*off-chip*), qui composent la mémoire DRAM (Dynamic Random-Access Memory) et dont les tailles sont plus importantes et les accès plus lents.

La hiérarchie de la mémoire est composée des mémoires suivantes :
— les registres dans chaque SM ;
— la mémoire locale dans la DRAM ;
— la mémoire partagée dans chaque SM ;
— la mémoire globale dans la DRAM ;
— la mémoire texture dans la DRAM ;
— la mémoire constante dans la DRAM.

Il existe plusieurs niveaux de cache d'accès :
— un cache de premier niveau (L1) dans chaque SM ;
— un cache de second niveau (L2) partagé par tous les SM ;
— un cache texture dans chaque SM.

Les caches L1 et L2 accélèrent les accès aux mémoires locale et globale, alors que le cache texture accélère les lectures de la mémoire texture. Le cache L1 partage un espace commun avec la mémoire partagée. Sa taille est configurable ; elle peut être 16 ou 48 ko pour les GPU de compute capability 2.0 et 16 , 32 ou 48 ko pour les GPU de compute capability 3.0 dans chaque SM. La taille du cache L2 est 768 ko pour les GPU de compute capability 2.0 et 1536 ko pour les GPU de compute capability 3.0. La taille du cache texture est 12 ko par SM.

Dans la figure 2.4, nous schématisons une vue d'ensemble des différents mémoires et caches, ainsi que les entités de la hiérarchie GBT qui leur correspondent. Nous donnons ensuite des détails sur chaque mémoire.

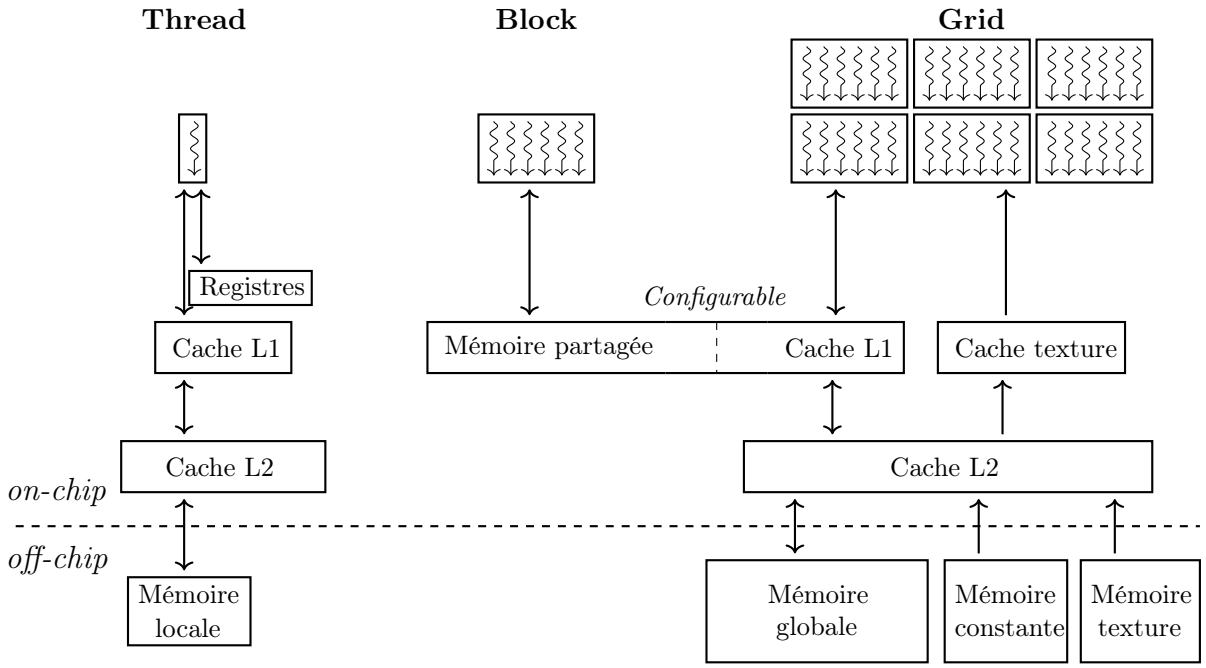


FIGURE 2.4 – Vue d'ensemble des espaces mémoire et des caches et les correspondances avec les entités de la hiérarchie GBT.

Registres

Le premier espace de mémoire est celui des registres qui sont utilisés par les *threads* pour effectuer les instructions et stocker les variables locales. Les registres sont des registres 32 bits. Leur nombre est de 32k par SM pour les GPU de compute capability 2.0 et de 64k par SM pour les GPU de compute capability 3.0. Ces registres sont partagés par tous les *threads* exécutés sur le SM. Comme le nombre de *threads* qui sont exécutés simultanément sur un SM est de l'ordre de quelques centaines voire du millier, le nombre des registres alloués à un *thread* est de l'ordre de quelques dizaines (au plus 63).

Mémoire locale

La mémoire locale complète l'espace mémoire réservé au *thread*. Elle est utilisée pour stocker des structures qui consommeraient beaucoup de registres. Comme c'est une mémoire *off-chip* (une partie de la DRAM), ses accès sont lents, mais bénéficient des niveaux de cache L1 et L2.

Mémoire partagée

La mémoire partagée est une mémoire dédiée à l'échange entre les *threads* d'un même *block*. Typiquement, elle est utilisée quand on a besoin de combiner des résultats partiels de plusieurs *threads*. La mémoire partagée se trouve dans chaque SM et elle est commune à tous les *blocks* qui sont exécutés sur le SM. La mémoire partagée est une mémoire on-chip, c'est l'espace mémoire qui a le plus grand débit et la plus petite latence, si nous ne considérons pas les registres. Par exemple, l'accès à la mémoire partagée est environ 100 fois plus rapide que l'accès à la mémoire globale [Har13a].

Dans chaque SM, l'espace mémoire dédié à la mémoire partagée est commun avec le cache L1. Le programmeur peut partitionner cet espace avec 2 configurations possibles : (48ko de cache L1 et 16ko de mémoire partagée) ou (16ko de cache L1 et 48ko de mémoire partagée). La configuration d'équipartition (32ko de cache L1 et 32ko de mémoire partagée) est uniquement supportée par les GPU de compute capability 3.0.

La mémoire partagée est organisée en 32 bancs (ou *banks*) de sorte à ce que des mots de 32 bits successifs se trouvent dans des bancs successifs [4]. Lorsque les *threads* d'un *warp* accèdent aux bancs, les requêtes ne doivent pas générer de conflit de banc. Un conflit de banc se produit lorsque les *threads* accèdent à des mots distincts appartenant à un même *banc*. Lorsque les *threads* accèdent au même mot (multicast), il n'y a pas de conflit de banc. Dans la figure 2.5, nous donnons quelques exemples d'accès à la mémoire partagée et nous précisons si le schéma d'accès génère ou non un conflit de banc.

Un schéma (*pattern*) d'accès mémoire qui ne génère pas de conflit permet de paralléliser les requêtes, autrement dit d'avoir une seule requête pour tous les *threads* du *warp*. Dans le cas contraire, un conflit de banc se traduit par une séquentialisation des requêtes. Évidemment, minimiser voire éviter les conflits de bancs permet de maximiser les performances d'accès à la mémoire partagée.

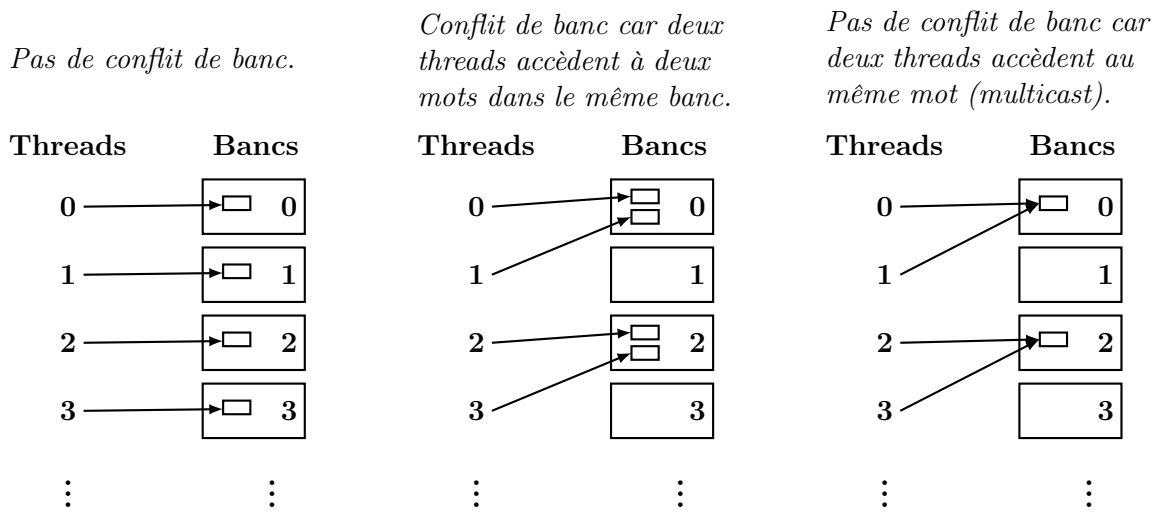


FIGURE 2.5 – Exemples de schémas d'accès à la mémoire partagée.

4. Pour les GPU de compute capability 3.0, il existe un mode 64 bits, où des mots de 64 bits se trouvent dans des bancs successifs.

Mémoire globale

La mémoire globale sert d'une part pour les transferts entre l'hôte et le périphérique, mais permet aussi de récupérer les entrées du *kernel* et d'écrire ses résultats finaux. Tous les *threads* du *grid* peuvent lire et écrire des données dans la mémoire globale.

La taille de la mémoire globale n'est pas une spécification de NVIDIA, pour un modèle de GPU donné. Sur le marché, nous pouvons trouvé deux à trois tailles possibles. Par exemple, pour le modèle GTX 680, nous trouvons sur le marché des cartes avec 2 Go ou 4 Go de DRAM.

Les accès à la mémoire globale sont « cachés » dans les caches L1 et L2 pour les GPU de compute capability 2.0 et seulement dans le cache L2 pour les GPU de compute capability 3.0. Une ligne du cache est de taille 128 octets. Lorsqu'un *warp* accède à la mémoire globale, sa requête est décomposée en plusieurs requêtes de 128 octets. En l'occurrence, si la taille d'un mot accédé par un *thread* est 4 octets, une seule requête est effectuée ; si la taille des mots est 8 octets pour un *thread*, deux requêtes sont effectuées.

Il existe un mécanisme de fusion d'accès (ou *coalescing*) du *warp* qui tend à fusionner les requêtes des 32 *threads*, c'est-à-dire permet de les effectuer simultanément. L'efficacité de ce mécanisme dépend de la localité des données. Lorsque nous observons les modèles des accès mémoire, il existe un schéma idéal, lorsque les accès sont contigus et alignés. Dans ce cas, la fusion est totale et on atteint les performances d'accès optimales. Deux autres schémas sont possibles. Le premier schéma correspond au cas où les accès des 32 *threads* restent contigus mais ne sont plus alignés, on parle alors d'un *offset* non nul. Le second schéma correspond au cas où les 32 *threads* accèdent à des zones espacées, on parle alors d'un *stride* différent de 1. Nous avons schématisé dans la figure 2.6 ces différents cas en considérant un accès d'un mot de 4 octets pour chaque *thread*.

Harris détaille dans [Har13b] le mécanisme de fusion d'accès et présente les résultats de mesure du débit d'accès à la mémoire globale lorsqu'on varie l'*offset* et le *stride*. Dans ses benchmarks, il a utilisé des cartes de compute capability 1.0, 1.3 et 2.0. Nous avons refait l'expérience de Harris sur une carte GeForce GTX 680 (compute capability 3.0) [5]. Les résultats sont visualisés dans la figure 2.7 et sont assez proches des résultats obtenus par Harris sur les GPU de compute capability 2.0. Pour le cas de l'offset non nul, le mécanisme de fusion d'accès n'est pas affecté et nous continuons à avoir de bonnes performances avec les GPU dont la compute capability est supérieure à 2.0 [6]. Pour le cas du *stride* différent de 1, nous avons une baisse des performances proportionnelle à la largeur du *stride*. Il est aussi envisageable de combiner les deux cas précédents, les performances sont alors proches de ce que nous observons pour le cas du *stride* différent de 1.

5. Code disponible sur GitHub : https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/coalescing-global

6. Pour les GPU dont la compute capability est inférieure à 2.0, Harris a observé une baisse des performances.

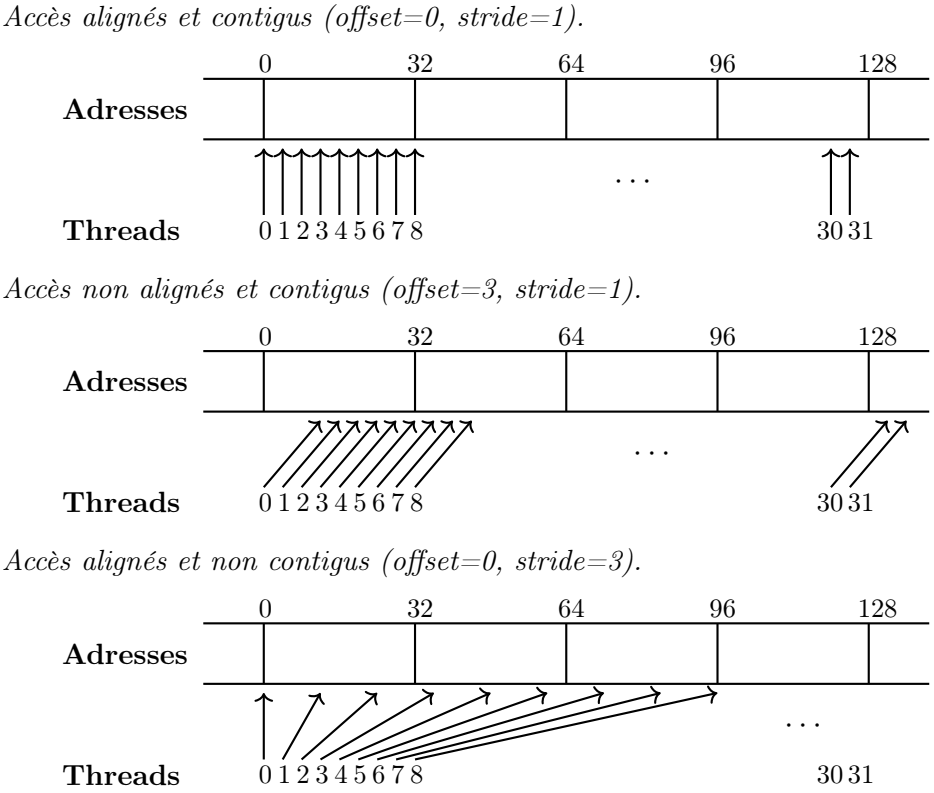


FIGURE 2.6 – Différents schémas d'accès à la mémoire globale par un *warp*. Chaque *thread* accède à un mot de 4 octets.

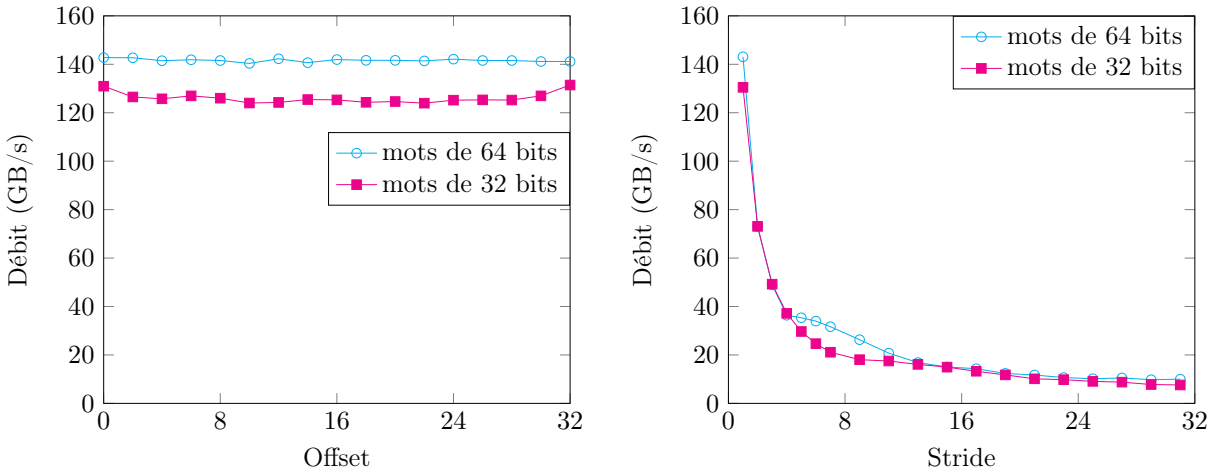


FIGURE 2.7 – Variation du débit effectif d'accès à la mémoire globale en fonction de l'*offset* (à gauche) et du *stride* (à droite) en précision simple et double sur une GTX 680.

Dans certains cas, nous pouvons être amenés à avoir nécessairement des schémas d'accès mémoire avec un *stride* plus grand que 1. La mémoire partagée est alors utilisée comme intermédiaire pour fusionner les accès. Par exemple, pour une lecture, on procède en deux temps ; on effectue une première copie des données de la mémoire globale à la mémoire partagée avec

des accès contigus, ensuite les *threads* lisent les données de la mémoire partagée avec des accès espacés. Ceci fonctionne parce que les accès à la mémoire partagée ne sont pas pénalisés par un espacement.

Mémoire texture

Les textures sont des concepts hérités de l'univers du traitement de l'image et de l'infographie qui représentent le cadre d'application traditionnel des GPU. Dans ces contextes, l'utilisation des textures permet d'exploiter la localité spatiale et de traiter d'une manière efficace des pixels adjacents. Dans un contexte de calcul générique, la localité spatiale correspond à un *thread* qui accède à une adresse proche des adresses accédées par des *threads* voisins.

La mémoire texture est une partie allouée de la mémoire globale ; c'est une zone mémoire off-chip en lecture seule. Elle est cachée dans le cache texture qui est localisé sur la puce. Lorsque l'application possède des schémas d'accès mémoire qui présentent une localité de données, utiliser la mémoire texture plutôt que la mémoire globale accélère les opérations de lecture.

Mémoire constante

La mémoire constante est une autre partie allouée de la mémoire globale. La mémoire constante est en lecture seule ; son mécanisme d'accès diffère de celui de la mémoire globale. La latence des accès en lecture à la mémoire constante est significativement plus petite que celle des accès à la mémoire globale. Toutefois, sa taille est limitée à 64 ko.

Type	Entité	Pénalité sur les temps d'accès
Registres	*thread*	1×
Locale	*thread*	100×
Partagée	*Block*	1×
Globale	*Grid*	100×
Texture	*Grid*	Dépend de l'application
Constante	*Grid*	1×

TABLE 2.1 – Récapitulatif des caractéristiques des mémoires.

Nous avons identifié dans cette hiérarchie de mémoire 9 niveaux de mémoire et de cache. L'ensemble de ces niveaux vont intervenir lors de l'implémentation des briques de l'algèbre linéaire. Nous allons dans le chapitre 5 détailler la façon avec laquelle les différents niveaux sont pris en compte et l'impact de ces choix.

2.3.6 Synchronisation des threads

Nous avons déjà vu que le *block* correspond à l'ensemble des *threads* qui sont amenés à collaborer entre eux pour effectuer une tâche. Ces *threads* peuvent échanger des données via la mémoire partagée et se synchroniser. Seuls les *threads* d'un même *warp* sont exécutés physiquement en même temps. Dans le cas où on a un intérêt à synchroniser l'ensemble des *threads* du *block*, il faut appeler une primitive de barrière de synchronisation, qui va garantir que tous les *threads* atteignent la même instruction avant de continuer. Une synchronisation de *threads* appartenant à des *warps* distincts est par conséquent plus coûteuse qu'une synchronisation de *threads* du même *warp*.

Dans certaines applications, on peut avoir un intérêt à synchroniser et à faire communiquer des *threads* appartenant à des *blocks* distincts. Dans ce cas, l'échange de données est réalisé à travers la mémoire globale. La synchronisation s'effectue en invoquant séparément deux *kernels* ; un premier *kernel* à la fin duquel tous les *threads* du *grid* écrivent à la mémoire globale et un second *kernel* au début duquel tous les *threads* du *grid* lisent les données de la mémoire globale. Ce schéma de synchronisation est évidemment alourdi par les invocations des *kernels* et les transferts de données.

Le jeu d'instructions CUDA fournit des instructions atomiques en mémoire partagée et en mémoire globale qui garantissent qu'une opération est effectuée par un *thread* sans l'interférence d'autres *threads*. Le jeu d'instructions fournit également pour les *threads* d'un *warp* des opérations d'échange de variable (*shuffle*), qui sont utiles dans des calculs de collecte de résultats [CUDAa].

2.3.7 Programmation plus bas niveau

L'environnement de programmation de NVIDIA propose en complément au language haut-niveau CUDA, un langage pseudo-assembleur, PTX (Parallel Thread Execution) [PTXa]. PTX fournit un jeu d'instructions, qu'un programmeur utilise pour écrire un code complet en PTX ou pour insérer des fonctions PTX dans un code CUDA [PTXb].

2.3.8 Bonnes pratiques pour l'optimisation du code CUDA

Le programmeur d'une application CUDA doit appliquer un ensemble de « bonnes pratiques » lors du design d'un *kernel* qui vont permettre de tirer avantage d'une manière efficace de la puissance de calcul du GPU. Nous allons décrire une liste non exhaustive de pratiques. Certaines pratiques sont spécifiques aux architectures Fermi et Kepler. Ces idées sont détaillées dans [CUDAa, chap. 5] et [CUDAb].

Optimisation algorithmique

Tout d'abord, il est important de n'employer le GPU que pour des algorithmes adaptés à la programmation parallèle. Il faut cibler des calculs (ou des parties de calcul) coûteux pour pouvoir amortir le temps des transferts *hôte-périphérique* qui sont importants. Par la suite, on est amené à adapter le *kernel* aux caractéristiques architecturales du GPU pour profiter de l'accélération fournie par le périphérique. En particulier, nous précisons trois actions principales :

— Il faut partager le travail sur les *threads* et les *blocks* de sorte à ce que les charges des multiprocesseurs soient équilibrés.
— Il faut privilégier le calcul par rapport au transfert. En effet, le GPU dispose d'une grande puissance arithmétique, alors que ses accès mémoire peuvent être très lents. Dans certains cas, recalculer une donnée est plus efficace que de la chercher en mémoire. La latence d'une opération arithmétique est de l'ordre de 20 cycles, celle d'un accès mémoire peut aller jusqu'à plus de 400 cycles [Vol10].
— Il faut exploiter la synchronisation physique du *warp*, réduire la synchronisation entre les *threads* du *block* qui engendre un surcoût et éviter le plus possible la synchronisation entre différents *blocks*.

Limitation de la divergence de code

Nous avons déjà mentionné la divergence de code, qui se produit lorsqu'au moins deux *threads* d'un même *warp* exécutent des instructions différentes. Typiquement lorsque nous avons unique-

ment deux instructions différentes, le SM séquentialise l'exécution des deux branches.

C'est pourquoi lorsqu'on réfléchit à comment distribuer le travail sur la hiérarchie GBT, il est plus intéressant de faire en sorte qu'un *warp* ne contienne que des *threads* qui exécutent un même code.

Dans certains cas, on peut être obligé d'avoir un comportement différent pour un ou plusieurs *threads* du *warp*. Une solution pour maintenir le paradigme SIMD est d'avoir la divergence dans les données plutôt que dans les instructions. Cette approche n'est réalisable que dans certains cas.

Amélioration de la bande passante de la mémoire

Les accès mémoire représentent le plus souvent le goulot d'étranglement d'une application CUDA. Il est important d'exploiter efficacement les différents niveaux de mémoires et de caches :

- Il faut privilégier l'utilisation des mémoires qui résident sur la puce (la mémoire partagée) et les mémoires qui ont une meilleure bande passante (la mémoire texture et la mémoire constante). En particulier, il faut utiliser la mémoire partagée plutôt que la mémoire globale tant que les tailles des données le permettent.
- Il faut mettre en place des schémas d'accès à la mémoire globale qui maximisent la fusion des accès, ainsi que des schémas d'accès à la mémoire partagée qui minimisent les conflits de bancs.
- Dans certains cas, il est intéressant d'utiliser la mémoire partagée comme intermédiaire pour fusionner des accès espacés à la mémoire globale (voir la sous-section 2.3.5).
- Il faut aussi mettre en place des schémas d'accès qui optimisent les caches (*cache-friendly*), en particulier le cache L1 et le cache texture.

Augmentation du remplissage

Le taux de remplissage (*occupancy* en terminologie NVIDIA) représente une mesure de l'utilisation des SM ; il est calculé comme le ratio moyen du nombre de *warps* exécutés sur un SM par le nombre maximal de *warps* [CUDAa]. Un taux de remplissage élevé indique que beaucoup de *threads* sont exécutés simultanément, ce qui permet de dissimuler la latence arithmétique, c'est-à-dire le temps nécessaire à un *thread* pour effectuer une opération. Pour un même ensemble de calculs menés, un taux de remplissage élevé améliore les performances ; alors que l'inverse, c'est-à-dire un taux de remplissage faible se traduit généralement par un impact négatif sur les performances.

L'utilisation des ressources est le facteur principal qui agit sur le remplissage. En effet, le nombre des registres et la quantité de mémoire partagée sont limités sur un SM. Si un *thread* consomme « beaucoup » de registres ou si un *block* alloue « beaucoup » de mémoire partagée, l'ordonnanceur va maintenir moins de *warps* par SM [LR11]. Le nombre de *threads* par *block* a aussi un impact sur le remplissage. En effet, un SM peut exécuter au plus 8 *blocks*. Par conséquent, si la taille des *blocks* est petite, nous aurons peu de *threads* qui sont exécutés en parallèle et nous ne pouvons plus masquer la latence des opérations. On masque la latence d'une opération en recouvrant avec d'autres opérations. Généralement, le nombre de *threads* par *block* devrait être entre 128 et 512 pour avoir des performances correctes.

Néanmoins, augmenter le remplissage n'est pas nécessairement l'unique stratégie pour masquer la latence. Volokov décrit dans [Vol10] une autre approche qui, au lieu d'augmenter le remplissage, va tendre à le réduire en rajoutant du parallélisme d'instructions. En d'autres termes, ceci consiste à diminuer le nombre de *threads* qui sont exécutés simultanément, en ayant plus

d'instructions par *thread*. Le parallélisme d'instructions va pouvoir masquer la latence. Dans certaines applications, cette stratégie améliore les performances plus que la stratégie d'augmenter le remplissage.

Les *kernels* que nous allons rencontrer ici, sont des *kernels* dont les performances sont limitées par les accès mémoire. Pour ces *kernels*, la stratégie d'augmenter le remplissage est plus rentable. C'est pourquoi dans ce travail, en particulier au chapitre 5, nous serons amenés à augmenter le remplissage, parce qu'il fournit une amélioration des performances des *kernels*.

2.3.9 Mesure de performance

NVIDIA offre plusieurs outils d'analyse et de profilage qui permettent de mesurer les performances d'une application et d'identifier les facteurs qui limitent les performances. Nous allons principalement utiliser deux outils :

— CUDA Occupancy Calculator : outil de calcul du remplissage, qui se présente sous la forme d'un tableur [7], où le programmeur indique les ressources consommées (registres et mémoire partagée) et le nombre de *threads* par *block* ;

— `nvvp` : outil de profilage graphique configurable (son équivalent non graphique est `nvprof`) qui effectue des mesures de métriques sur la consommation des ressources, l'efficacité des accès mémoire, la divergence de code, etc.

2.4 Passage à l'échelle d'un cluster de calcul

Maintenant, nous nous intéressons à l'exécution d'une application sur un cluster de calcul. Le cluster est composé d'un ensemble de nœuds. Les nœuds sont interconnectés par un réseau de communication à haut débit, typiquement de l'InfiniBand (IB). Nous considérons les applications où les différents nœuds de calcul partagent des données de taille importante, ce qui correspond à notre contexte d'algèbre linéaire. Ce sont donc les aspects liés aux communications que nous allons détailler. Nous allons nous limiter aux architectures qui nous intéressent : les CPU multi-cœurs et les GPU.

2.4.1 Le réseau InfiniBand (IB)

Le réseau InfiniBand est un réseau de communication à haute performance qui fournit une latence de l'ordre de la microseconde et un débit de plusieurs dizaines de Gbit/s (voir dans la table 2.2 la latence et le débit des différents réseaux InfiniBand). Il est possible d'utiliser la couche native IB verbs ou de passer par une bibliothèque de communication (tel que MPI) qui utilise directement la couche IB verbs.

	Latence	Débit théorique
InfiniBand DDR	2.5 µs	16 Gb/s
InfiniBand QDR	1.3 µs	32 Gb/s
InfiniBand FDR-10	0.7 µs	41.25 Gb/s
InfiniBand FDR	0.7 µs	54.54 Gb/s
InfiniBand EDR	0.5 µs	100 Gb/s

TABLE 2.2 – Latence et débit des différents réseaux InfiniBand.

7. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Dans le cadre de ce travail, nous utiliserons des clusters équipés du réseau InfiniBand QDR ou du réseau InfiniBand FDR.

2.4.2 Communications CPU

Nous supposons que le cluster est composé de machines multi-cœurs. Supposons que nous avons des processus ou des threads [8] qui sont exécutés sur les cœurs des machines. Nous avons alors deux types de communication :

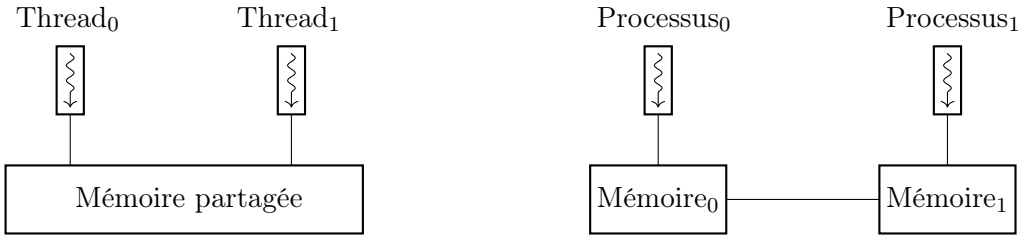
— les communications entre les processus exécutés sur le même nœud ;
— les communications entre les processus exécutés sur des nœuds distincts.

Ces deux types de communication sont désignés respectivement par les communications *intra-nœud* et les communications *inter-nœuds*.

Il existe deux modèles de communication. À chacun de ces modèles correspond un certain nombre d'outils et d'intergiciels disponibles. La question à ce niveau est de comparer ces deux modèles, en particulier en terme du coût lié à l'utilisation de leurs outils respectifs et à quel point ces outils correspondent aux contraintes de notre application.

— Le modèle mémoire partagée, utilisé dans OpenMP et POSIX Threads, où les *threads* communiquent en partageant leurs données sur une mémoire commune (voir figure 2.8a).
— Le modèle de passage de messages, où la mémoire est distribuée sur tous les processus qui, pour échanger des données, sont amenés à effectuer des transferts (voir figure 2.8b). Les modèles de passage de messagess, typiquement MPI, utilisent généralement des procédés de type RDMA (Remote Direct Memory Access), qui ne font pas intervenir les CPU, plutôt que des procédés de type *polling* qui font intervenir les CPU. Par exemple, les implémentations MPI utilisent le RDMA sur les réseaux InfiniBand et le RoCE (RDMA over Converged Ethernet) sur les réseaux Ethernet. Ces techniques sont plus efficaces en terme de latence et de débit.

Le modèle de passage de messages de MPI se positionne logiquement pour des communications *inter-nœuds*, alors que celui de OpenMP est plus approprié aux communications *intra-nœud*.



(a) Modèle à mémoire partagée (b) Modèle de passage de messages

FIGURE 2.8 – Modèles de communication.

Pour des applications qui regroupent les deux types de communication, comme celles que nous considérons ici, trois stratégies peuvent s'appliquer [RHJ13] :

— utiliser un même modèle, typiquement MPI, dans tous les communications, en assignant un ou deux processus par cœur ;

8. Différence entre procesus et *thread* : un processus est une instance du programme, un *thread* est un bout de code qui est exécuté en parallèle. Dans le modèle MPI, on parle de processus. Dans le modèle OpenMPI, on parle plutôt de threads.

— utiliser à la fois MPI et OpenMP (ou POSIX) où MPI gère les communications *inter-nœuds* et OpenMP gère les communications *intra-nœud*, en assignant à chaque nœud un processus MPI, qui exécute sur les cœurs du nœud plusieurs *threads* OpenMP ;
— utiliser MPI pour les communications *inter-nœuds* et les directives MPI-3 d'utilisation de la mémoire partagée pour les communications *intra-nœud*.

La première stratégie est la plus simple à mettre en œuvre vu qu'un seul modèle est utilisé. Toutefois, les communications *intra-nœud* ne sont potentiellement pas optimales, même si elles seront plus rapides que les communications *inter-nœuds* parce qu'elles ne sont pas effectuées à travers le réseau. La seconde stratégie permet de diminuer la mémoire consommée, vu que les *threads* OpenMP partagent de la mémoire, mais peut présenter certaines difficultés liées à l'inadéquation des deux modèles, typiquement les implémentations MPI actuelles ne supportent pas beaucoup le *multi-threading*. La troisième stratégie représente théoriquement l'alternative la plus prometteuse à l'utilisation MPI+OpenMP.

2.4.3 Communications GPU

Nous supposons maintenant que les nœuds du cluster hébergent des GPU et nous discutons de comment transférer les données d'un GPU à un autre. De même que pour les communications CPU, nous devons traiter deux cas possibles (et non exclusifs) : le premier quand un seul nœud héberge deux GPU ou plus, et le second quand les GPU sont sur des nœuds distincts. Par souci de simplification, nous considérons uniquement deux GPU et le scénario d'envoyer des données de l'un à l'autre. On peut facilement étendre ceci vers le cas d'un plus grand nombre de GPU et pour construire des routines de communication plus compliquées.

Communications *intra-nœud*

CUDA offre trois schémas de communication possibles :
— Passer par le CPU : la communication doit impliquer l'*hôte*. Elle est composée de deux transferts, une copie du *périphérique* vers l'*hôte* (D2H : Device to Host), puis une copie de l'*hôte* vers le *périphérique* (H2D : Host to Device) (voir figure 2.9).

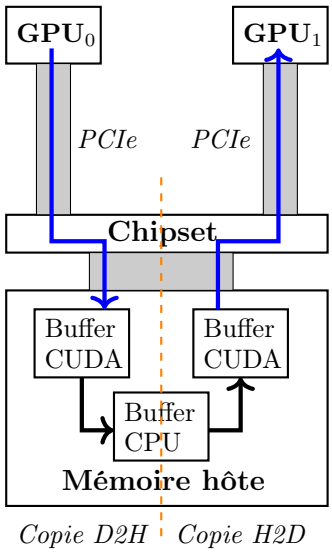


FIGURE 2.9 – Communication GPU *intra-nœud* en passant par le CPU.

— Copie du *périphérique* vers le *périphérique* (D2D pour Device to Device) : du point de vue du programmeur, il s'agit une copie directe des *buffers* GPU. Toutefois, le transfert continue de passer par la mémoire de l'*hôte*. Par rapport au cas précédent, la copie est cependant complètement *pipelinée*, alors que dans le premier cas, les deux transferts sont séparément *pipelinés* (voir figure 2.10).

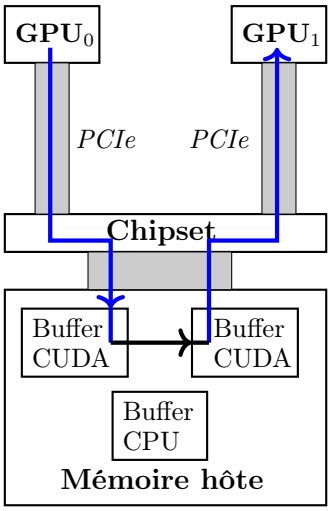


FIGURE 2.10 – Communication GPU *intra-nœud* avec copie du *périphérique* vers le *périphérique*.

— Accès Direct Pair à Pair (P2P DMA pour Peer-to-Peer Direct Memory Access) : la fonctionnalité permet aux GPU de partager des données indépendamment du CPU [Sch11] (voir figure 2.11). Cette fonctionnalité requiert d'activer l'accès pair à pair dans chaque GPU. La fonctionnalité P2P DMA réduit le surcoût du passage par la mémoire de l'*hôte* et ainsi accélère les transferts.

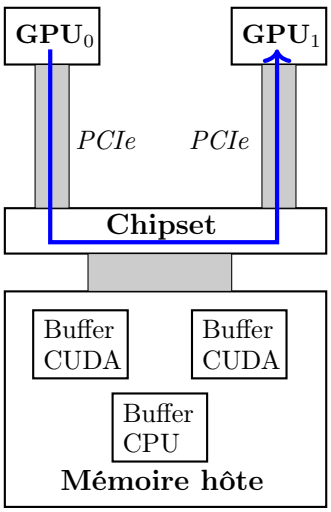


FIGURE 2.11 – Communication GPU *intra-nœud* avec un Accès Direct Pair à Pair.

Pour vérifier l'accélération fournie par le P2P DMA, nous avons mesuré les débits et les latences relatives à chaque approche. L'expérience a été réalisée avec deux cartes NVIDIA GeForce GTX 680. Nous avons mesuré le temps d'une copie d'un GPU vers un GPU pour des tailles crois-

santes de données. L'observation de la figure 2.12 montre que l'utilisation de la fonctionnalité P2P DMA accélère les communications en terme de latence et de débit.

	Latence en µs
Passer par CPU	24
Copie D2D	22
Copie P2P DMA	18

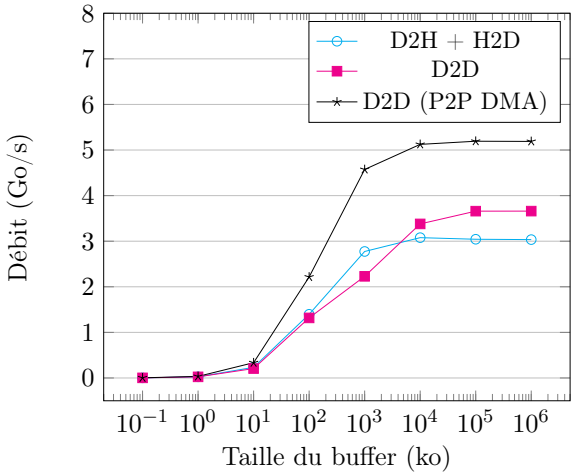


FIGURE 2.12 – Comparaison des communications GPU *intra-nœud* en terme de latence et de débit.

Communications *inter-nœuds*

L'option triviale est d'effectuer le transfert en trois étapes : effectuer une copie du GPU vers le CPU en utilisant les routines CUDA (copie D2H), enuite utiliser MPI pour faire une copie entre les CPU (copie H2H), et finalement une copie CUDA du CPU vers le GPU destinataire (copie H2D) (voir figure 2.13).

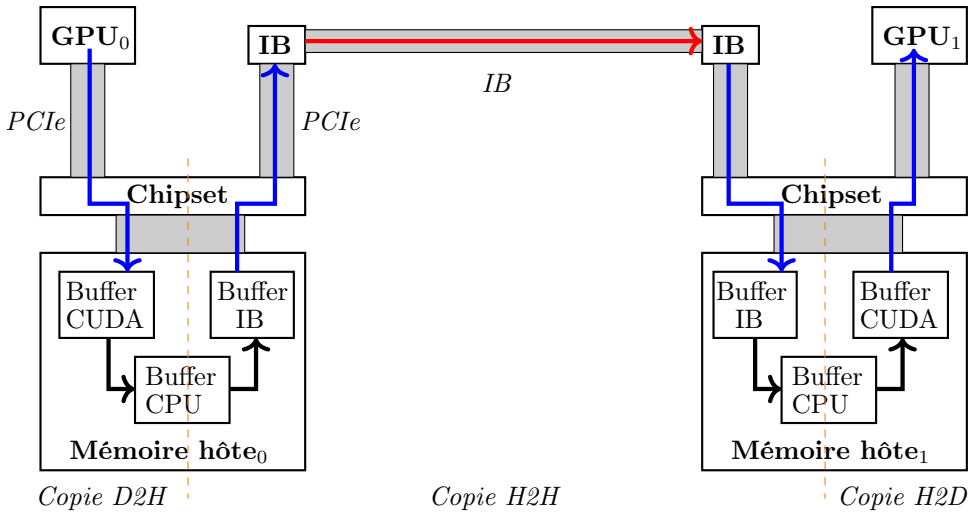


FIGURE 2.13 – Communication GPU *inter-nœuds* en 3 étapes.

Toutefois, il est possible d'éviter de passer par le CPU en utilisant la fonctionnalité *Cuda-aware MPI* qui combine MPI et CUDA [Kra13a, Kra13b]. Cette fonctionnalité permet d'utiliser les buffers GPU directement dans les routines MPI. Du point de vue du programmeur, le transfert de données se réduit à un seul appel à une routine MPI. Avec la fonctionnalité *Cuda-aware MPI*,

les transferts sont complètement *pipelinés*, alors qu'avec la première approche, chaque copie est séparément *pipelinée*. La fonctionnalité est intégrée dans plusieurs bibliothèques MPI, telles que MVAPICH2 (à partir de la version 1.8) et OpenMPI (à partir de la version 1.7).

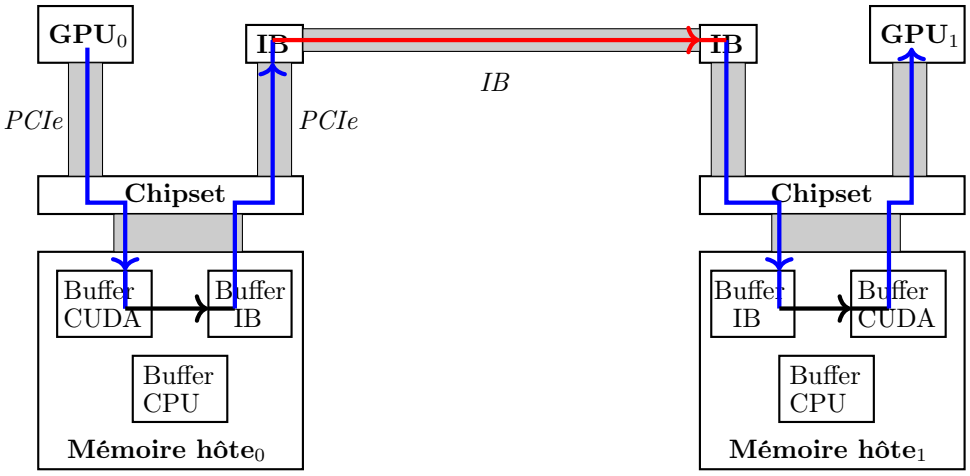


FIGURE 2.14 – Communication GPU *inter-nœuds* avec *Cuda-aware MPI*.

Pour estimer l'accélération qu'apporte la fonctionnalité *Cuda-aware MPI*, nous avons réalisé des mesures des communications GPU *inter-nœuds* entre deux cartes GeForce GTX 680 installées sur deux nœuds connectés par de l'InfiniBand QDR. Nous avons utilisé CUDA 5.0 et Open MPI 1.7.3. Nous avons ajouté aux mesures des communications GPU des mesures relatives à une copie d'un buffer CPU à un autre buffer CPU en utilisant les routines MPI, pour avoir une référence.

	Latence en µs
Copie D2D sans Cuda-aware MPI	15
Copie D2D avec Cuda-aware MPI	13
Copie H2H avec MPI	1

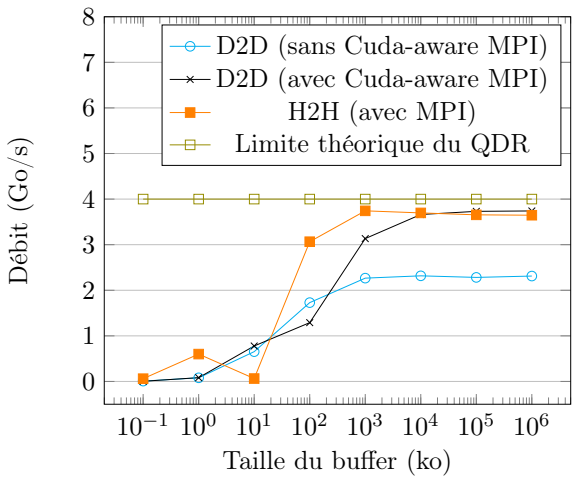


FIGURE 2.15 – Comparaison en terme de latence et de débit des communications GPU *inter-nœuds*, avec et sans *Cuda-aware MPI*, ainsi qu'une comparaison avec la communication entre *hôtes*.

La figure 2.15 montre que la fonctionnalité *Cuda-aware MPI* permet d'atteindre à partir d'une grande taille des données envoyées le débit d'un transfert MPI d'un buffer CPU à un autre buffer CPU. Ces deux débits sont proches de la limite théorique du QDR (4 Go/s), qui correspond au débit maximal qu'une communication *inter-nœuds* peut atteindre. Le gain fourni par l'accé-

lération *Cuda-aware MPI* est semblable à celui fourni par le P2P DMA pour les communications *intra-nœud* (voir figure 2.12).

Il existe une autre fonctionnalité, *GPUDirect*, qui optimise les communications [GPUDirect] en enlevant le passage par la mémoire de l'*hôte* (voir figure 2.16). En utilisant cette fonctionnalité, la latence d'un transfert de donnée est réduite par rapport à un transfert qui passe par le CPU. Toutefois, le *GPUDirect* ne devrait pas amener une amélioration au niveau du débit [PHV+13]. Nous n'avons pas pu déployer cette fonctionnalité dans notre application, car elle n'est supportée que par des cartes Tesla et Quadro récentes, auxquelles nous n'avions pas accès. Le lecteur intéressé par des mesures des performances de l'utilisation de *GPUDirect* peut se référer au travail de [PHV+13] qui rapporte des mesures en terme de latence et de débit.

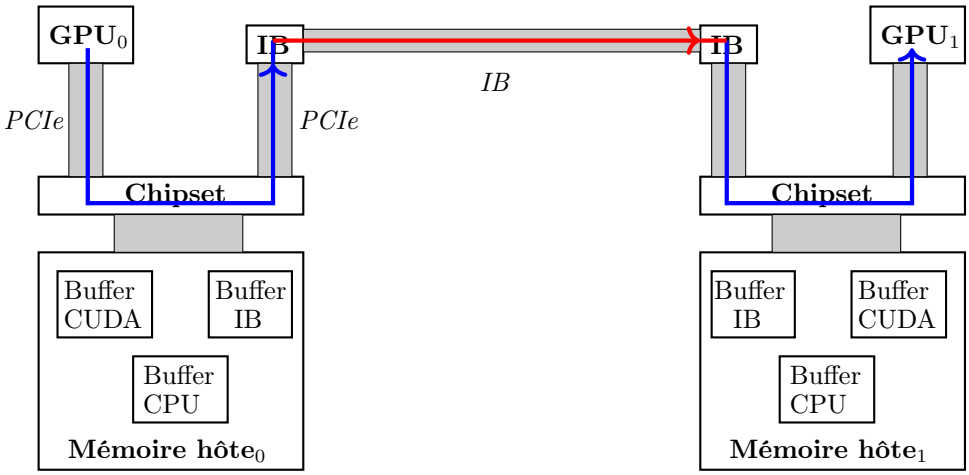


FIGURE 2.16 – Communication GPU *inter-nœuds* avec *Cuda-aware MPI* et *GPUDirect*.

Les mesures de performance effectuées pour les communications *intra-* et *inter-nœuds* ont été obtenues avec des communications bloquantes de type `MPI_Send` et `MPI_Recv`. Ces résultats sont très proches de ceux que nous avons obtenus avec des communications non-bloquantes de type `MPI_Isend` et `MPI_Irecv`.

2.5 Conclusion

Ce chapitre « technologique » a principalement servi pour la familiarisation du lecteur avec la programmation sur les GPU NVIDIA, aussi bien pour le cas d'un seul GPU, que pour le cas multi-GPU. Nous avons introduit la terminologie NVIDIA et le modèle de programmation CUDA. Nous avons décrit l'organisation des *threads*, des cœurs et des mémoires et discuté les mécanismes de communication pour les cas multi-CPU et multi-GPU. Nous aurons recours dans les chapitres suivants à la plupart de ces concepts et de ces mécanismes pour décrire les approches et les solutions proposées.

Deuxième partie

Résolution de systèmes linéaires creux

Chapitre 3

Algèbre linéaire creuse pour le logarithme discret

Ce chapitre présente le problème d'algèbre linéaire qui résulte des calculs de logarithme discret. L'algèbre linéaire considérée ici correspond à la résolution de systèmes linéaires creux sur des corps finis, c'est-à-dire des calculs exacts. De nombreux aspects des méthodes issues des calculs numériques ne sont pas applicables dans ce contexte. Le point principal de divergence des ces deux types d'algèbre linéaire est que dans le contexte numérique les méthodes utilisées tendent à converger vers la solution recherchée, alors que dans le cas exact, sur un corps fini, il n'est pas possible d'approcher ou de converger vers la solution.
Une partie importante de ce chapitre a fait l'objet de la publication [Jel14b].

Sommaire

3.1 Présentation du problème

Les systèmes d'équations linéaires que nous allons étudier ici sont issus des algorithmes de calcul d'index. Pour une description plus détaillée de ces algorithmes, le lecteur peut se référer au chapitre 1. Dans ces algorithmes, l'étape de *crible* fournit des relations entre les logarithmes de certains éléments. Par la suite, l'étape de *filtrage* en quelque sorte « nettoie » et « prépare » le système à résoudre. Nous avons besoin de résoudre ce système pour pouvoir calculer dans l'étape de *logarithme individuel* le logarithme discret de n'importe quel élément du corps, à partir des logarithmes déjà connus.

Nous calculons le logarithme discret dans un sous-groupe multiplicatif d'un corps fini $\mathbb{F}_q$. L'ordre de ce sous-groupe est premier et est noté ℓ. On rappelle que ℓ est un diviseur de $q-1$ (voir sous-section 1.4.5). Les entrées de la phase d'algèbre linéaire sont :

— le nombre premier ℓ ;
— une matrice carrée et singulière A, à N lignes et colonnes et définie dans le corps fini $(\mathbb{Z}/\ell\mathbb{Z})$.

Nous cherchons un élément non trivial du noyau de la matrice, autrement dit une solution de l'équation

$$Aw \bmod \ell = 0 \tag{3.1}$$

Nous allons considérer des matrices issues des calculs avec les algorithmes FFS et NFS. Les propriétés que nous allons décrire peuvent ne pas être valides pour des matrices issues d'autres algorithmes de calcul d'index.

Fabrication de la matrice par l'étape de filtrage

L'étape de filtrage est une étape de pré-calcul qui permet de réduire la taille de la matrice. La description que nous fournissons du filtrage est basée sur les travaux [Cav02, chap. 3] et [Bou13].

À l'entrée du filtrage, la matrice est très grande et très creuse (environ 20 coefficients non nuls par ligne). La matrice n'est généralement pas carrée ; elle contient plus de lignes que de colonnes (c'est-à-dire plus d'équations que d'inconnues). L'objectif du filtrage est de diminuer la taille de la matrice sans trop la densifier ; la densité finale correspond à quelques centaines de coefficients non nuls par ligne. On désire généralement pour la suite de l'algèbre linéaire une matrice carrée.

Cette étape de pré-calcul tend à augmenter le nombre moyen de coefficients non nuls par ligne, mais permet de diminuer la taille de la matrice. La diminution de la taille est importante pour la résolution effective du système linéaire. D'une part, elle réduit la quantité de mémoire nécessaire à la représentation et au traitement de la matrice. D'autre part, elle diminue le coût de la résolution effective, sachant que les algorithmes de résolution effective, comme nous allons le voir dans la prochaine section, ont une complexité au moins quadratique, si ce n'est cubique, en la taille de la matrice. Au fur et à mesure que le filtrage agit sur la matrice, le coût estimé de la résolution effective diminue. Le filtrage est arrêté lorsque le coût recommence à augmenter.

Il existe une autre méthode pour diminuer la taille de la matrice, qui est celle de l'*élimination gaussienne structurée* (SGE pour Structured Gaussian Elimination) [LO90, PS92]. Les approches SGE et celle du filtrage sont similaires ; elles se distinguent de par la stratégie et les critères qu'elles emploient pour le choix des lignes à combiner ou à supprimer.

Dans la table 3.1, nous illustrons l'évolution de la taille de la matrice pendant le filtrage avec deux exemples de matrices issues de calculs concrets de logarithme discret ; une matrice FFS

qui correspond à la résolution du logarithme discret dans $\mathbb{F}_{2^{809}}$ et une matrice NFS issue de la résolution du logarithme discret dans un corps premier $\mathbb{F}_{p_{155}}$, où p_{155} est un nombre premier de 155 chiffres décimaux.

Calcul	FFS pour $\mathbb{F}_{2^{809}}$	NFS pour $\mathbb{F}_{p_{155}}$
#relations uniques / #inconnues	80 963 931 / 39 357 159	26 835 094 / 14 891 504
Poids moyen des lignes avant le filtrage	21.8	24.2
Taille de la matrice après le filtrage (N)	3 602 667	2 561 574
Poids moyen des lignes après le filtrage	100	105

TABLE 3.1 – Évolution de la matrice pendant le filtrage avec les exemples dans $\mathbb{F}_{2^{809}}$ et $\mathbb{F}_{p_{155}}$.

3.2 Caractéristiques des entrées

Le nombre ℓ est un « grand » nombre premier, de l'ordre de quelques centaines de bits. D'un point de vue cryptographique, le nombre ℓ doit être suffisamment grand de sorte à ce que le sous-groupe correspondant résiste à des attaques de type Pollard rho (voir sous-section 1.4.1 pour de plus amples explications).

La matrice A est grande. Sa dimension N peut aller de centaines de milliers à des dizaines de millions de lignes, si on se réfère aux calculs récents de logarithme discret.

La matrice est creuse. Une analyse asymptotique donne une densité indicative de $O\left(\log^2 N\right)$ coefficients par ligne. En pratique, pour les problèmes étudiés, cette densité est de quelques centaines de coefficients par ligne.

La table 3.2 donne les caractéristiques des entrées de l'algèbre linéaire correspondant aux exemples précédents.

Calcul	FFS pour $\mathbb{F}_{2^{809}}$	NFS pour $\mathbb{F}_{p_{155}}$
Taille de ℓ (bits)	202	511
Taille de la matrice (N)	3 602 667	2 561 574
Nombre de coefficients non nuls	360 266 822	268 965 377
Poids moyen des lignes	100	105

TABLE 3.2 – Caractéristiques des matrices utilisées pour les exemples dans $\mathbb{F}_{2^{809}}$ et $\mathbb{F}_{p_{155}}$.

3.2.1 Nature des coefficients

La matrice A est définie sur le corps $(\mathbb{Z}/\ell\mathbb{Z})$. Toutefois, la majorité des coefficients sont « petits », dans le sens où ils peuvent être représentés par un entier de petite taille, tenant dans un mot machine (éventuellement signé). Ces coefficients correspondent à des exposants dans les relations trouvées dans la phase de *crible*.

Pour les matrices issues d'un calcul FFS, tous les coefficients sont « petits ». Les courbes de la figure 3.1 montrent la répartition des valeurs des coefficients dans la matrice-exemple de FFS pour $\mathbb{F}_{2^{809}}$. En abscisse, on représente les valeurs des coefficients et en ordonnée le nombre d'apparition dans la matrice.

Tous les coefficients sont compris entre -35 et 36. En échelle linéaire, il y a deux grands pics qui correspondent aux valeurs -1 et 1 et deux autres pics moins importants pour les valeurs -2 et 2. En effet, 92.7% des coefficients sont des ± 1 et 4.5% des coefficients sont des ± 2. Quand

nous passons en échelle logarithmique, nous observons une répartition quasi triangulaire que nous n'expliquons pas ici ; l'explication pourrait être trouvée avec une étude statistique de l'étape de filtrage.

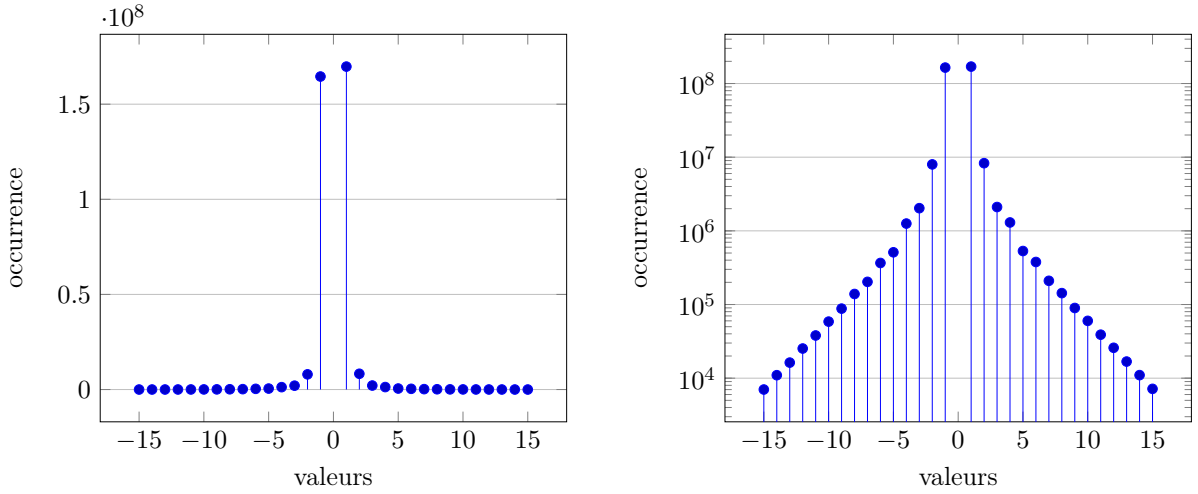


FIGURE 3.1 – Répartition des valeurs des coefficients dans la matrice-exemple de FFS pour $\mathbb{F}_{2^{809}}$ en échelle linéaire (à gauche) et en échelle logarithmique (à droite).

Pour les matrices issues d'un calcul NFS, la majorité des coefficients sont « petits ». Toutefois, en plus des coefficients petits, nous avons des colonnes particulières (dites *colonnes de caractères*). Ces colonnes sont denses et contiennent des éléments qui sont « grands », c'est-à-dire leur représentant dans $[0, \ell[$ occupe autant de mots machine que ℓ (voir sous-sous-section 1.4.4). Le nombre de ces colonnes ne dépasse généralement pas la dizaine ; en l'occurrence pour la matrice-exemple de NFS pour $\mathbb{F}_{p_{155}}$, il y a 5 *colonnes de caractères*. Dans cette matrice, 95% des coefficients sont « petits ».

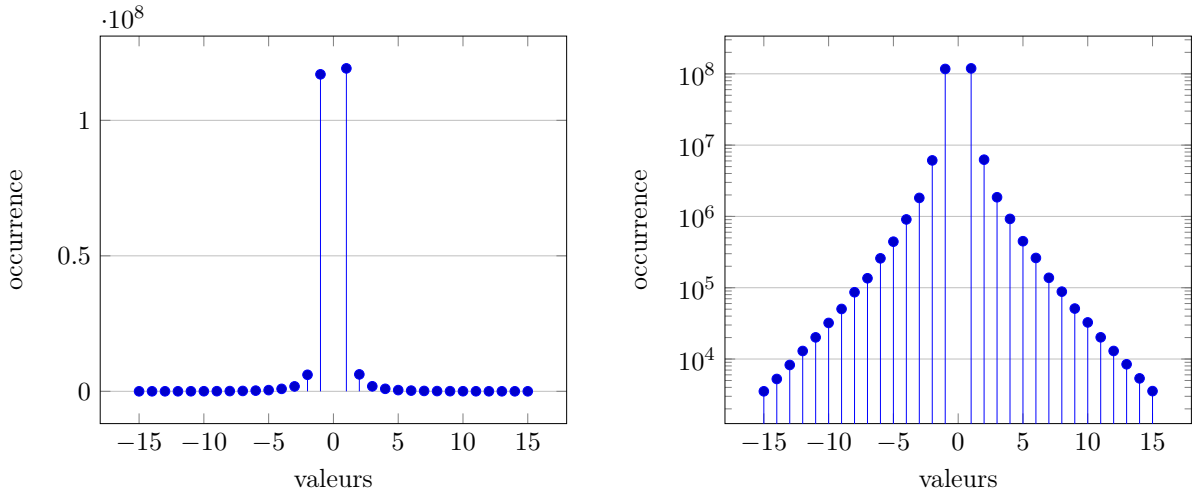


FIGURE 3.2 – Répartition des valeurs des coefficients « petits » dans la matrice-exemple de NFS pour $\mathbb{F}_{p_{155}}$ en échelle linéaire (à gauche) et en échelle logarithmique (à droite).

Les courbes de la figure 3.2 montrent la répartition des valeurs des coefficients « petits ».

Tous les coefficients sont compris entre -35 et 35. De même que pour la matrice-exemple pour FFS, nous observons les deux pics correspondant aux valeurs -1 et 1 ainsi que la répartition quasi triangulaire des valeurs.

3.2.2 Distribution des coefficients non nuls

La distribution des coefficients non nuls de la matrice creuse n'est pas régulière. Toutefois, les coefficients non nuls sont plus ou moins localisés dans certaines parties de la matrice. Les figures 3.3 et 3.4 montrent les distributions typiques des matrices issues des algorithmes FFS et NFS. Ces figures ont été obtenues en calculant les densités de blocs contenant quelques centaines de lignes et de colonnes (800 pour la figure 3.3 et 600 pour la figure 3.4). Un bloc est ensuite visualisé par un point.

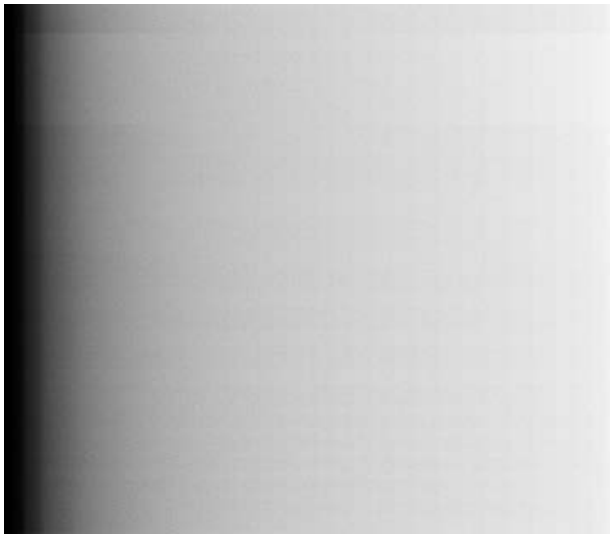


FIGURE 3.3 – Distribution des coefficients non nuls dans la matrice-exemple de FFS pour $\mathbb{F}_{2^{809}}$.

Dans la matrice-exemple issue de FFS, les premières colonnes de la matrice sont relativement denses, ensuite la densité des colonnes diminue progressivement. On note l'évolution de la densité des colonnes comme suit :

Indices des colonnes	Densité des colonnes	Contribution des colonnes au nombre total de coefficients non nuls de la matrice
0-76	>10%	22.5%
77-475	1-10%	10.6%
476-4948	0.1-1%	13.4%
4949-68580	0.01-0.1%	17.6%
68581-3602666	<0.01%	35.9%

La densité des lignes ne change pas considérablement.

FIGURE 3.4 – Distribution des coefficients non nuls dans la matrice-exemple NFS pour $\mathbb{F}_{p_{155}}$.

Comme avec la matrice FFS, la densité des colonnes de la matrice NFS diminue progressivement, sauf pour les *colonnes de caractères*, qui sont denses (ces colonnes sont représentées dans la figure 3.4 par un trait continu fin à l'extrême droite de la matrice). La densité des colonnes évolue comme suit :

Indices des colonnes	Densité des colonnes	Contribution des colonnes au nombre total de coefficients non nuls de la matrice
0-76	>10%	22.2%
77-602	1-10%	12.7%
603-6122	0.1-1%	13%
6123-75183	0.01-0.1%	16.2%
75183-2561568	<0.01%	31.1%
2561568-2561573	100%	4.8%

3.3 Du parallélisme pour résoudre le problème

Résoudre ce problème d'algèbre linéaire nécessite, comme nous allons le détailler dans ce chapitre, des calculs lourds. En effet, les algorithmes de résolution de tels problèmes ont une complexité au moins quadratique en la dimension de la matrice N. Rappelons que, pour résoudre des logarithmes discrets dans des groupes de plus en plus grands, les matrices sont de plus en plus grandes. Par ailleurs, ces grandes matrices posent des problèmes de stockage, avec des tailles de données qui peuvent atteindre quelques dizaines de gigaoctets. Par conséquent, il est important de pouvoir *paralléliser* les calculs d'algèbre linéaire, pour que ce calcul puisse être distribué en plusieurs sous-calculs, où chaque sous-calcul est exécuté sur un nœud de calcul. Une distribution du calcul initial sur plusieurs unités calculatoires diminue le temps d'exécution et les exigences en terme de puissance de calcul et de mémoire par rapport à un calcul qui aurait été fait uniquement sur une seule unité calculatoire.

Il existe quatre niveaux de parallélisme pour la résolution de systèmes linéaires creux sur les corps finis :

1. Le niveau algorithmique : les algorithmes de résolution dits « par blocs » permettent de distribuer le calcul de l'algèbre linéaire en plusieurs calculs indépendants. Ce parallélisme est détaillé dans la section 3.5, en particulier celui de l'algorithme Wiedemann par blocs (voir les sous-sections 3.5.3 et 3.5.4).
2. Le niveau du produit matrice-vecteur sur plusieurs nœuds : l'opération centrale dans les algorithmes de résolution est un produit de la matrice creuse par un vecteur. Le chapitre 4 explore comment distribuer cette opération sur plusieurs nœuds de calcul, où chaque nœud effectue un produit partiel.
3. Le niveau du produit partiel : si le produit partiel est effectué sur une architecture hautement parallèle, par exemple un processeur graphique, plusieurs *threads* vont collaborer pour calculer le produit partiel. Le chapitre 5 discute la stratégie de parallélisation de ce produit.
4. Le niveau de l'arithmétique sur les corps finis : une opération arithmétique sur un grand corps fini, dont un élément occupe plusieurs mots machine, peut être parallélisée. En l'occurrence, le système de représentation RNS (Residue Number System) permet de distribuer une opération arithmétique en plusieurs « petites » opérations indépendantes. Le chapitre 6 détaille cet aspect.

3.4 Solveurs directs et itératifs

Pour résoudre des systèmes d'équations linéaires, deux familles d'algorithmes peuvent être utilisées :

— les méthodes directes ;
— les méthodes itératives.

Nous allons étudier l'efficacité de ces méthodes dans notre contexte. Parmi ces méthodes, certaines ne s'adaptent pas forcément très bien aux systèmes définis sur des grands corps finis.

3.4.1 Les méthodes directes

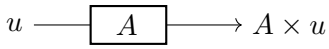
Ces méthodes sont des algorithmes classiques pour l'algèbre linéaire numérique, sur des nombres réels. Parmi ces méthodes, nous trouvons l'élimination gaussienne et les décompositions Cholesky, LU et QR [FM67, Wes68, PTV+92].

Les méthodes directes requièrent $O\left(N^{\omega}\right)$ opérations dans le corps, où l'exposant ω est égal à 3 dans le cas du produit matriciel naïf, et à 2.807 si l'algorithme de Strassen [Str69] est utilisé. Dans notre contexte, la matrice correspondant au système linéaire est creuse, mais appliquer les méthodes directes tendra à la densifier. Ceci augmente la quantité de mémoire nécessaire pour représenter la matrice. En effet, la complexité en espace mémoire des méthodes directes atteint $O\left(N^2\right)$, car la matrice devient dense. Ainsi, en employant une méthode directe, la mémoire nécessaire pour traiter la matrice de FFS pour $\mathbb{F}_{2^{809}}$ serait passée de 3 Go à 13 To.

3.4.2 Les méthodes itératives

Parmi les méthodes itératives, il y a la méthode du gradient conjugué [HS52] et les algorithmes de Lanczos [Lan52] et de Wiedemann [Wie86].

Les méthodes itératives sont des méthodes dîtes *black-box*, dans le sens où elles n'agissent pas sur la matrice ; elles utilisent la matrice uniquement pour effectuer une opération de produit matrice-vecteur.



Les méthodes nécessitent $O(N)$ produits matrice-creuse–vecteur. Leur complexité en espace mémoire est en $O(N\gamma)$, où γ désigne le nombre moyen de coefficients non nuls par ligne.

Si un produit matrice-creuse–vecteur est effectué en moins de $O\left(N^{\omega-1}\right)$ opérations dans le corps, les méthodes itératives ont une meilleure complexité asymptotique. Cette condition est réalisable, vu que les matrices issues du calcul de logarithme discret contiennent un nombre petit de coefficients non nuls. Le nombre moyen γ de coefficients non nuls par ligne est négligeable devant la dimension N. Ainsi, la complexité du produit matrice-creuse–vecteur, qui est égale à $O(N\gamma)$, est meilleure que $O\left(N^{\omega-1}\right)$, lorsque $\gamma < N^{\omega-2}$. Par conséquent, les méthodes itératives sont asymptotiquement plus rapides que les méthodes directes pour notre contexte.

Nous pouvons conclure que pour résoudre les systèmes linéaires issus de calcul de logarithme discret, les arguments de la complexité en mémoire et de la complexité en temps sont en faveur des méthodes itératives par rapport aux méthodes directes.

3.5 Algorithmes itératifs de résolution d'algèbre linéaire

3.5.1 Algorithme de Lanczos

L'algorithme de Lanczos [Lan52] a été inventé pour résoudre des systèmes linéaires avec des coefficients réels. Il se base sur le procédé d'orthogonalisation de Gram-Schmidt, utilisé sur $\mathbb{R}$ pour construire une base de vecteurs orthogonaux à partir d'un ensemble de vecteurs non liés. L'algorithme de Lanczos adapte ce procédé d'orthogonalisation pour des espaces de Krylov. On définit un espace de Krylov K à partir d'un vecteur arbitraire v et de la matrice symétrique $B = {}^tAA$ comme le sous-espace engendré par les vecteurs $v, Bv, B^2v, \ldots$.

On définit la forme bilinéaire symétrique suivante de $K \times K$ vers $\mathbb{Z}/\ell\mathbb{Z}$:

$$\langle x, y \rangle = {}^txBy \text{ pour } x, y \in K \tag{3.2}$$

L'algorithme de Lanczos construit une suite de vecteurs $w_0, w_1, \ldots, w_k$ dont les éléments sont orthogonaux :

$$w_0 = v \tag{3.3}$$

$$w_1 = Bw_0 - \frac{\langle Bw_0, Bw_0 \rangle}{\langle w_0, Bw_0 \rangle} w_0 \tag{3.4}$$

$$\vdots$$

$$w_k = Bw_{k-1} - \frac{\langle Bw_{k-1}, Bw_{k-1} \rangle}{\langle w_{k-1}, Bw_{k-1} \rangle} w_{k-1} - \frac{\langle Bw_{k-1}, Bw_{k-2} \rangle}{\langle w_{k-2}, Bw_{k-2} \rangle} w_{k-2} \tag{3.5}$$

L'algorithme se termine lorsqu'il trouve un vecteur isotrope pour l'application bilinéaire (le vecteur $x \in K$ est un vecteur isotrope si et seulement si $\langle x, x \rangle = 0$). Alors, nous pouvons déduire un vecteur du noyau de la matrice A avec une bonne probabilité. Le cas problématique arrive lorsque nous trouvons un vecteur qui satisfait ${}^t(Ax)Ax = 0$ et que $Ax \neq 0$, ce qui arrive avec une faible probabilité lorsque nous travaillons dans un corps fini.

La dimension du sous-espace de Krylov est proche de N. La complexité de l'algorithme est alors celle de N itérations. Si on ne tient pas compte des coûts des produits scalaires, chaque itération correspond à deux produits matrice-vecteur, une multiplication par la matrice A et une multiplication par sa transposée ${}^tA$. Pour une étude plus approfondie de l'algorithme de Lanczos, on oriente le lecteur vers l'article [EK97].

3.5.2 Algorithme de Wiedemann

L'algorithme de Wiedemann prend deux vecteurs aléatoires $x, y \in (\mathbb{Z}/\ell\mathbb{Z})^N$. Il est composé de 3 étapes essentielles :

1. *Produits scalaires* : on calcule les $2N$ premiers termes de la suite $(a_i)_{i\in\mathbb{N}} \in (\mathbb{Z}/\ell\mathbb{Z})^{\mathbb{N}}$, où $a_i = {}^t x A^i y$.
2. *Générateur linéaire* : on calcule le polynôme minimal de la séquence, qui est le polynôme $F(X) = \sum_{i=0}^{d} f_i X^i$ de plus petit degré d tel que $\sum_{i=0}^{d} f_i a_{k+i} = 0$ pour tout $k \geq 0$. Le degré d est proche de N.
3. *Évaluation* : on calcule $w = F(A)z$, où z est un vecteur arbitraire de $(\mathbb{Z}/\ell\mathbb{Z})^N$.

La suite calculée dans l'étape *Produits scalaires* est une suite récurrente linéaire. Son polynôme minimal F est un diviseur du polynôme minimal μ_A de la matrice A. Supposons que les deux polynômes sont égaux. Puisque la matrice a un noyau non trivial, son polynôme minimal s'écrit sous la forme $\mu_A(X) = X^k P(X)$. L'exposant k est petit et est généralement égal à 1. Nous avons calculé le polynôme w comme $P(A)z$. Il existe alors un entier i qui est au plus égal à k tel que $A^i w = 0$. Par conséquent $A^{i-1}w$ est un vecteur non trivial de la matrice.

Si la matrice n'est pas singulière, le sortie w de l'algorithme est le vecteur nul. Sinon (c'est-à-dire si la dimension du noyau de la matrice est non nulle), alors le vecteur w est avec une grande probabilité un élément non nul du noyau de la matrice. La probabilité d'échec d'une exécution de l'algorithme de Wiedemann est en $O\left(\frac{1}{\ell}\right)$ [Wie86].

Les noms des étapes (en italique) correspondent aux appellations originales qui apparaissent dans l'article de Wiedemann [Wie86]. Dans le logiciel `CADO-NFS` [CADO] et dans notre implémentation d'algèbre linéaire, on utilise les noms `Krylov`, `Lingen` et `Mksol` pour désigner les programmes correspondant aux étapes *Produits scalaires*, *Générateur linéaire* et *Évaluation* respectivement.

En théorie, x est tiré aléatoirement dans $(\mathbb{Z}/\ell\mathbb{Z})^N$. Toutefois, en pratique, on le prend dans la base canonique, ou éventuellement de faible poids de Hamming, de sorte à pouvoir simplifier le produit scalaire entre ${}^t x$ and $A^i y$. Au lieu d'effectuer un produit scalaire complet, on prend juste l'élément (ou les éléments) de $A^i y$ qui correspond (ou correspondent) à la coordonnée (ou aux coordonnées) non nulle(s) de x. Cette approche ne pose empiriquement pas de problème avec les matrices qu'on étudie.

L'algorithme de Wiedemann requiert $3N$ produits matrice-vecteur pour les étapes *Produits scalaires* et *Évaluation*. Pour l'étape *Générateur linéaire*, on peut utiliser l'algorithme de Berlekamp-Massey [Ber68, Mas69] ou l'algorithme d'Euclide, dont la complexité est $O\left(N^2\right)$ opérations dans le corps, ou encore un algorithme asymptotique rapide tel que l'algorithme Half-gcd [Knu70, BGY80] dont la complexité est $O\left(N(\log N)^2\right)$ opérations dans le corps. Un produit matrice-creuse–vecteur est effectué en $O\left(N\gamma\right)$ opérations dans le corps, avec γ le nombre moyen de coefficients non nuls par ligne. Ainsi, les complexités relatives aux étapes *Produits scalaires* et *Évaluation* sont quadratiques en la dimension N. Dans ce mémoire, nous reviendrons plus en détails sur l'accélération des étapes *Produits scalaires* et *Évaluation* qui sont les étapes dominantes du calcul. Plus spécifiquement, nous allons étudier l'optimisation de l'opération produit matrice-creuse–vecteur, qui est centrale dans ces deux étapes. Plus de détails sur le calcul du générateur linéaire sont donnés dans l'article [Tho02].

À présent, nous allons nous intéresser aux versions « par blocs » des algorithmes de résolution. Coppersmith [Cop93] et Montgomery [Mon95] ont élaboré simultanément la version par blocs

de l'algorithme de Lanczos. La version par blocs de l'algorithme de Wiedemann a été introduite par Coppersmith [Cop94]. Dans ce recueil, nous nous contenterons de détailler l'algorithme de Wiedemann par blocs, parce qu'il présente certaines propriétés de parallélisme dont nous tirerons avantage, alors que paralléliser l'algorithme de Lanczos nécessite des contraintes plus strictes. En l'occurrence, l'algorithme de Wiedemann par blocs peut être exécuté sur différents clusters indépendants, alors que l'algorithme de Lanczos par blocs ne peut être exécuté que sur un seul cluster.

3.5.3 Algorithme de Wiedemann par blocs

L'algorithme de Wiedemann par blocs remplace le vecteur $y \in (\mathbb{Z}/\ell\mathbb{Z})^N$ par un bloc de n vecteurs $y^{(0)}, \ldots, y^{(n-1)}$, chacun dans $(\mathbb{Z}/\ell\mathbb{Z})^N$, et d'une manière similaire, remplace le vecteur x par un bloc de m vecteurs. La séquence de scalaires a_i est alors remplacée par une séquence de matrices $m \times n$. Les étapes de l'algorithme deviennent :

1. *Produits scalaires* : cette étape calcule les $\lceil \frac{N}{n} \rceil + \lceil \frac{N}{m} \rceil$ premiers termes de la séquence $(a_i)_{i \in \mathbb{N}}$.
2. *Générateur linéaire* : on cherche un générateur linéaire pour la séquence précédente. La sortie de *Générateur linéaire* est composé de n générateurs $F^{(0)}, \ldots, F^{(n-1)}$, chacun un polynôme dans $\mathbb{Z}/\ell\mathbb{Z}$ de degré plus petit que $\lceil \frac{N}{n} \rceil$.
3. *Évaluation* : on calcule l'élément suivant $w = \sum_{j=1}^{n} F^{(j)}(A) y^{(j)}$.

Les paramètres (n, m) sont désignés dans la littérature par les terminologies *blocking factors* ou *blocking parameters*.

Le nombre total des itérations des étapes *Produits scalaires* et *Évaluation* devient $\lceil \frac{2N}{n} \rceil + \lceil \frac{N}{m} \rceil$. Chaque itération est dominée par le temps d'un produit matrice–bloc-de-n-vecteurs. Si m et n sont du même ordre de grandeur et négligeables devant $\log N$, la complexité de l'étape *Générateur linéaire* devient $O\left(nN^2\right)$ en utilisant l'algorithme Berlekamp-Massey matriciel et $O\left(nN \log N (\log N + \log p)(n^{\omega-2} + \log N) \log\log N\right)$ avec l'algorithme rapide [DMT74, BL94, Tho02].

3.5.4 Intérêt des blocs

Dans cette section, supposons que les paramètres m et n sont égaux. Nous reviendrons dans la sous-section suivante sur le choix des valeurs pour m et n.

Les algorithmes par blocs présentent deux avantages principaux. Le premier avantage est l'amélioration la probabilité de succès. Ce bénéfice est d'autant plus important pour les systèmes définis sur $\mathbb{F}_2$, où la probabilité d'échec de l'algorithme de Wiedemann simple est trop élevée. Cette probabilité d'échec devient négligeable avec l'algorithme de Wiedemann par blocs. Une argumentation détaillée peut être trouvée dans [Tho03, Vil97].

Le second avantage consiste à remplacer le produit matrice-vecteur par le produit matrice–bloc-de-vecteurs. En effet, dans l'algorithme de Wiedemann simple, les étapes *Produits scalaires* et *Évaluation* sont composées de 3N itérations ; chaque itération correspond à un produit de la matrice creuse par un vecteur. L'algorithme de Wiedemann par blocs divise le nombre des itérations par n ; et chaque itération correspond à un produit de la matrice par un bloc de n vecteurs. Or, effectuer un produit d'une matrice creuse par un bloc de n vecteurs coûte souvent moins que n produits de la même matrice par un vecteur. En effet, lorsque qu'on traite simultanément

un bloc de vecteurs, la lecture des coefficients de la matrice est « factorisée ». Ceci est particulièrement intéressant pour les systèmes définis sur $\mathbb{F}_2$ où l'on prend les *blocking parameters* des multiples de 64 et où l'on encode un bloc de 64 vecteurs dans un mot machine. L'arithmétique est alors plus efficace que si nous multiplions par des vecteurs de bits.

On peut aussi tirer avantage autrement de l'opération du produit matrice–bloc-de-vecteurs en distribuant cette opération en plusieurs produits matrice-vecteur ou plusieurs produits matrice–bloc-de-vecteurs. En effet, remarquons que lorsqu'on effectue le produit d'une matrice par un bloc de n vecteurs de la forme $v = A \times u$, la j^{e} colonne du bloc de sortie, qu'on note $v^{(j)}$, dépend uniquement de la j^{e} colonne du bloc d'entrée, notée $u^{(j)}$. Ainsi, le calcul $v = A \times u$ peut être distribué en n tâches parallèles, chaque tâche calcule $v^{(j)} = Au^{(j)}$. La figure 3.5 schématise cette distribution sur un exemple avec $n = 3$. En distribuant l'opération centrale des étapes *Produits scalaires* et *Évaluation*, on arrive à distribuer le calcul de chaque étape en n calculs parallèles et indépendants qui ne nécessitent ni synchronisation, ni communication, excepté à la fin lorsqu'on combine tous les résultats.

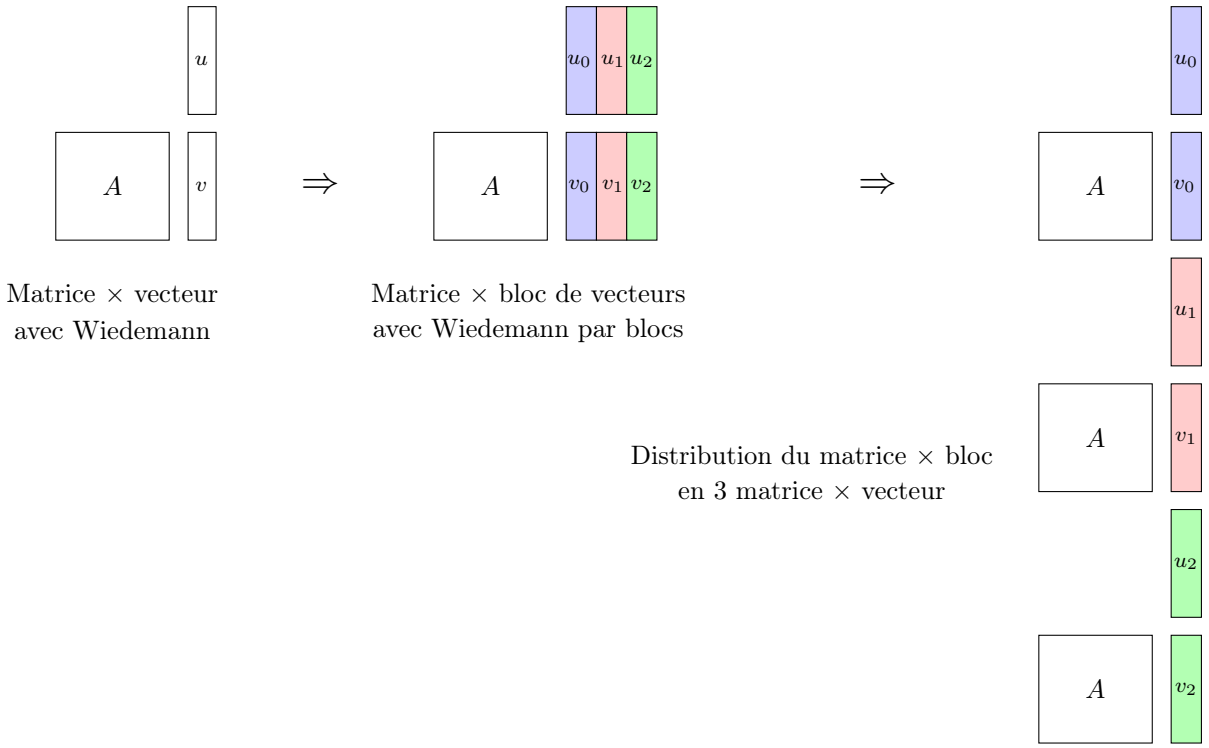


FIGURE 3.5 – Évolution de l'opération centrale entre Wiedemann simple et Wiedemannn par blocs et distribution du produit matrice × bloc en plusieurs produits matrice × vecteur.

Pour récapituler, nous venons de détailler deux façons possibles pour tirer avantage de l'utilisation des blocs. La première façon consiste à réduire la complexité totale des deux étapes les plus importantes, en continuant d'avoir une exécution séquentielle. La seconde est de créer du parallélisme en distribuant le calcul de chaque étapes en n calculs parallèles, chaque calcul étant exécuté sur un nœud de calcul. Ces deux façons ne sont pas mutuellement exclusives. Typiquement, lorsque on travaille dans $\mathbb{F}_2$, on prend généralement $n = 64n'$. Ainsi, on distribue le calcul sur n' calculs parallèles ; chaque calcul effectue simultanément le produit de la matrice par 64 vecteurs, représentés dans un seul mot machine.

3.5.5 Choix des paramètres (n, m)

En théorie, il y a une liberté totale dont le choix des valeurs pour les paramètres. Toutefois, prendre des (n, m) très grands va rendre l'étape *Générateur linéaire* difficile. En effet, à N constant, la complexité de cette étape est linéaire ou davantage en $(n + m)$.

Pour le choix de n, lorsque nous disposons de n' unités (nœuds) de calcul, on prend souvent n égal à un multiple de n'. Pour les systèmes définis sur $\mathbb{F}_2$, comme nous l'avons mentionné, on prend $n = 64n'$. Pour les systèmes définis sur des grands corps finis, on prend $n = n'$ ou bien $n = 2n'$ si l'architecture du nœud permet de traiter simultanément et d'une manière efficace deux vecteurs, par exemple avec des opérations SIMD.

Pour le choix de m, il n'y a pas de contrainte particulière. On tend généralement à prendre m plus grand que n, car si nous prenons les vecteurs du bloc x dans la base canonique (voir la remarque dans la sous-section 3.5.2), multiplier par x devient très peu cher. Augmenter m devient alors intéressant, vu que ceci permet de réduire le nombre d'itérations de l'étape *Produits scalaires* sans surcoût (rappel : ce nombre est égal à $\lceil \frac{N}{n} \rceil + \lceil \frac{N}{m} \rceil$).

Le choix des paramètres (m, n) est aussi défini par des contraintes d'implémentation. Par exemple, dans le logiciel `CADO-NFS` [CADO] m est choisi égal à $2n$ pour une question d'efficacité de l'exécution parallèle du programme `Lingen`, qui se parallélise d'autant mieux que $\gcd(n, m)$ est grand.

3.6 Conclusion

Nous avons présenté le problème d'algèbre linéaire et les spécificités liées aux matrices qui interviennent dans les calculs de logarithme discret. Nous avons détaillé les différents niveaux de parallélisme que nous pourrons utiliser pour résoudre efficacement ce problème sur des architectures pensées pour le calcul parallèle. Nous avons montré l'intérêt d'utiliser dans notre contexte les algorithmes spécifiques au caractère creux et nous avons détaillé comment l'emploi des blocs dans ces algorithmes permet de distribuer les calculs d'algèbre linéaire.

Chapitre 4

Paralléliser le produit matrice-vecteur sur plusieurs nœuds

Dans ce chapitre, nous allons discuter la parallélisation du produit matrice-vecteur sur plusieurs nœuds de calcul. Nous allons présenter le modèle de calcul et décrire les différentes étapes d'un produit matrice-vecteur parallèle. Une parallélisation efficace nécessite de distribuer des charges de travail équilibrées sur les nœuds de calcul et de minimiser les coûts de communications. Dans notre application, la question de l'équilibre des charges est d'autant plus importante, à cause du caractère creux des matrices considérées.
Les travaux détaillés dans ce chapitre ont été publiés dans [Jel14b].

Sommaire

4.1 Modèle

Nous supposons disposer d'un ensemble de *nœuds* de calcul, identiques, organisés dans une *grille* bidimensionnelle carrée (i.e., chaque case de la grille correspond à un nœud) de taille $t \times t$. Les nœuds sont interconnectés par un *réseau* de communication. Chaque nœud est identifié par ses coordonnées (i, j) dans la grille.

Dans ce modèle, la notion de *nœud* de calcul est une notion qui fait abstraction de la nature du *nœud*. Un *nœud* peut correspondre à un cœur dans une machine, à une machine indépendante ou à une carte graphique. On insiste sur le fait de ne pas confondre cette notion avec la notion du nœud de cluster, telle que nous l'avons utilisée dans le chapitre 2 et qui correspond à une machine connectée au réseau.

L'objectif est de pouvoir utiliser cette grille de calcul pour paralléliser le produit matrice-creuse–vecteur (SpMV pour Sparse-Matrix–Vector product). Plus précisément, les entrées sont la matrice creuse A et un vecteur d'entrée, dense, qu'on note u. Les *nœuds* de calcul collaborent ensemble pour pouvoir calculer le vecteur de sortie $v \leftarrow Au$. Le produit est itératif, dans le sens où le vecteur de sortie de l'itération k devient le vecteur d'entrée de l'itération $(k + 1)$.

4.2 Distribution de la charge de travail

La matrice A est divisée en *sous-matrices* carrées, de même taille $\frac{N}{t} \times \frac{N}{t}$, de sorte que chaque sous-matrice est assignée à un *nœud* de la grille.

La distribution particulière des coefficients non nuls dans la matrice creuse fait que les nœuds ont des charges de travail déséquilibrées. Les nœuds qui ont les sous-matrices les plus denses mettent plus de temps pour traiter leurs sous-matrices que les nœuds qui ont les sous-matrices les plus creuses.

Pour le type particulier des matrices dont nous disposons, le problème de déséquilibre peut être résolu d'une manière efficace. Pour corriger le problème, nous appliquons des permutations des lignes et des colonnes, de telle sorte à ce que la distribution des coefficients non nuls pour toutes les sous-matrices soient proches (ces distributions sont aussi proches de celle de la matrice A).

Une possibilité pour obtenir cette permutation est de trier les colonnes par leurs poids (on rappelle que le poids est le nombre de coefficients non nuls) et de les distribuer d'une manière uniforme sur les nœuds, puis de procéder de la même façon avec les lignes. Ceci est possible par le fait que l'écart-type du poids des lignes est beaucoup plus petit que celui du poids des colonnes.

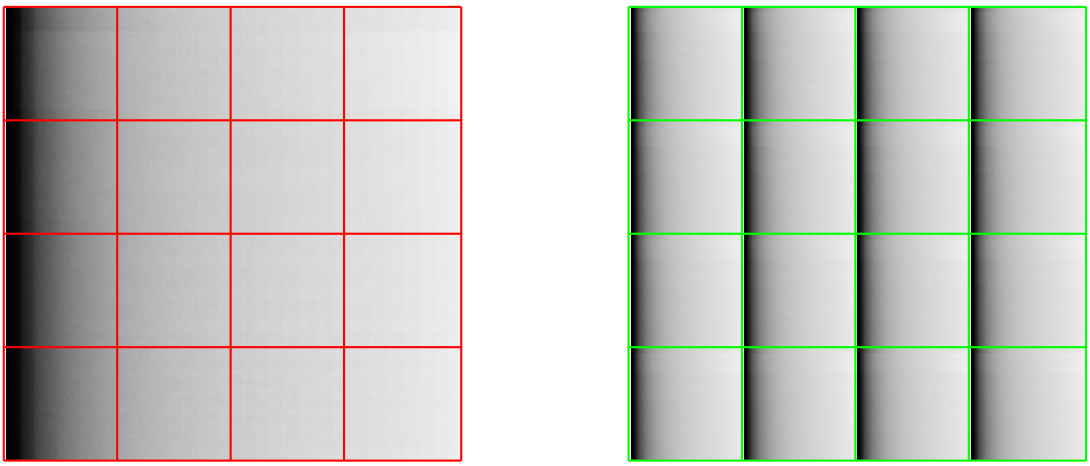
Nous illustrons cette technique d'équilibrage sur la matrice-exemple du calcul de FFS dans $\mathbb{F}_{2^{809}}$ (voir table 3.2). Nous supposons que t est égal à 4, c'est-dire que nous voulons diviser la matrice en 16 sous-matrices. La figure 4.1a montre la distribution des coefficients non nuls dans la matrice initiale et la variation de la densité des sous-matrices. La matrice totale contient 360.2 M coefficients non nuls ; une distribution équitable des charges de travail nécessiterait que chaque sous-matrice contienne 6.25% des coefficients non nuls, soit environ 22.5 M coefficients. Toutefois, les densités de la sous-matrice la plus dense et de la sous-matrice la moins dense sont comme suit :

— la sous-matrice la moins dense (en haut à droite) contient 2.1 M coefficients non nuls, soit 0.6% des coefficients non nuls de la totalité de la matrice ;
— la sous-matrice la plus dense (en bas à gauche) contient 81.8 M coefficients non nuls, soit 22.7% des coefficients ;

Nous appliquons le procédé de permutation de lignes et de colonnes. La figure 4.1b visualise la distribution des coefficients non nuls dans la matrice obtenue et dans ses sous-matrices. Maintenant, les densités de la sous-matrice la plus dense et de la sous-matrice la moins dense deviennent :

— la sous-matrice la plus dense contient 22.7 M coefficients non nuls, soit 6.3% des coefficients non nuls de la totalité de la matrice ;
— la sous-matrice la moins dense contient 22.2 M coefficients non nuls, soit 6.16% des coefficients ;

D'une part, les sous-matrices ont des densités relativement proches. D'autre part, les distributions spatiales des coefficients non nuls sont proches. Ainsi, des nœuds identiques mettent quasiment le même temps pour traiter les sous-matrices.



(a) Matrice initiale. (b) Matrice permutée.

FIGURE 4.1 – Distribution des coefficients non nuls pour une matrice initiale et une matrice permutée (matrice de FFS pour $\mathbb{F}_{2^{809}}$).

Les permutations des lignes et des colonnes ne changent pas le rang de la matrice et n'ont par conséquent pas d'impact sur la solution du système initial. Nous cherchons un élément du noyau de la matrice permutée, duquel un élément du noyau de la matrice A pourra par la suite être facilement déduit.

4.3 Schéma de calcul/communication

Nous raisonnons sur les communications entre nœuds selon le modèle de passage de messages (voir sous-section 2.4.2), même si dans la pratique, d'autres opérations (accès direct à la mémoire DMA) sont utilisées. Chaque processus qui est exécuté sur le nœud de calcul possède une mémoire privée et les processus effectuent des transferts de type envoi/réception pour s'échanger leurs données.

Au début d'une itération, un *nœud* (i, j) stocke dans sa mémoire la sous-matrice A_{ij} et le j^{e} fragment u_j du vecteur d'entrée u. À la fin de l'itération, le *nœud* (i, j) a besoin d'avoir dans sa mémoire le j^{e} fragment v_j du vecteur de sortie pour pouvoir calculer l'itération suivante. La longueur d'un fragment des vecteurs d'entrée ou de sortie est $\frac{N}{t}$ éléments.

Le produit matrice-vecteur parallèle est effectué en trois étapes, selon le schéma suivant :

1. *SpMV* : Chaque *nœud* (i, j) calcule le SpMV partiel $A_{ij}u_j$.
2. *Réduction* : Chaque *nœud* diagonal (i, i) collecte et somme les résultats partiels des nœuds de la ligne i. En effet, chaque nœud (i, j) calcule une contribution à v_i (i^{e} fragment de v) ; et la somme des contributions des nœuds de la ligne correspond à v_i.
3. *Broadcast* : Chaque *nœud* diagonal (i, i) diffuse (broadcast) son fragment v_i à tous les nœuds de la colonne i.

Dans la figure 4.2, nous présentons un exemple d'une exécution du schéma précédent pour 4 nœuds. La matrice A est divisée en 4 sous-matrices. Dans la figure, les 4 nœuds, colorés en gris, sont numérotés de 0 à 3.

Dans la partie gauche de la figure, sont indiqués les 4 sous-matrices, les fragments du vecteur d'entrée u_0 et u_1 et les fragments du vecteur de sortie v_0 et v_1 qui vérifient :

$$\begin{cases} v_0 &= A_{00}u_0 + A_{01}u_1 \\ v_1 &= A_{10}u_0 + A_{11}u_1 \end{cases}$$

Dans la partie centrale, nous montrons la répartition des calculs sur la grille.

Dans la partie droite, nous détaillons les données intermédiaires présentes dans chaque *nœud* après chaque étape.

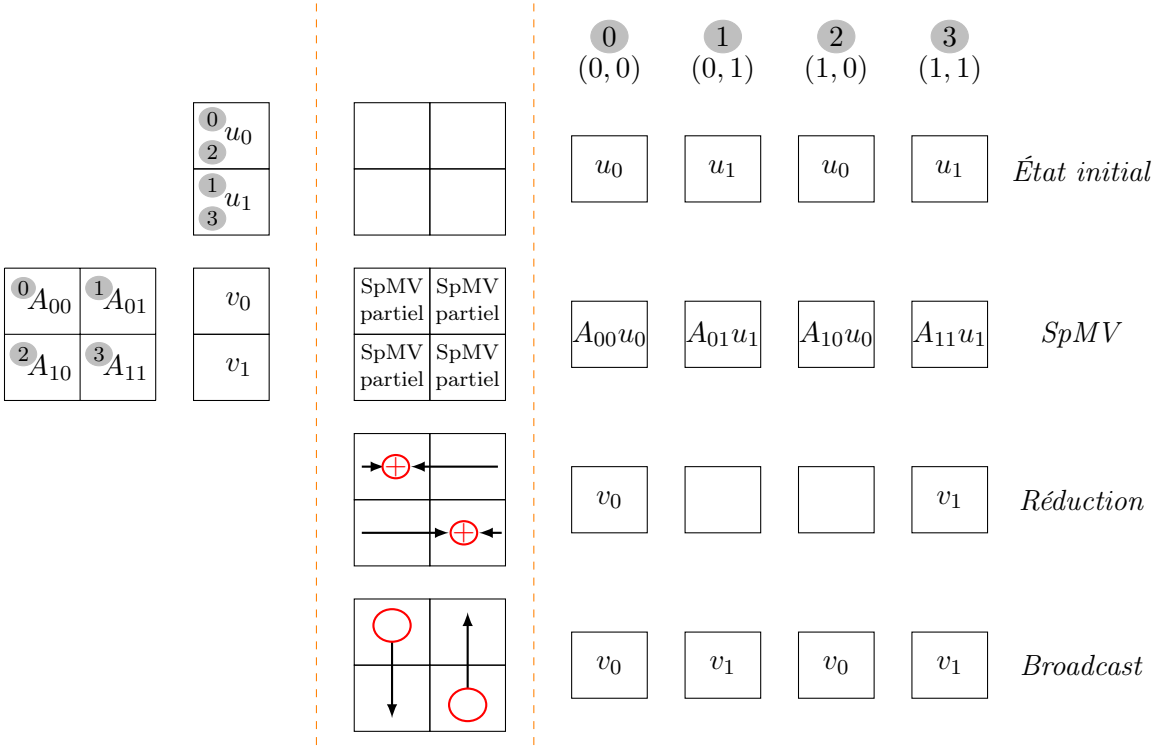


FIGURE 4.2 – Schéma de calcul/communication pour une division 2×2 de la matrice, en utilisant les opérations *Reduction/Broadcast*.

Dans le schéma précédent, pour une ligne donnée, un *nœud* collecte les $(t-1)$ autres résultats partiels et pour une colonne donnée, un *nœud* diffuse le fragment vers les $(t-1)$ autres nœuds. Ce schéma souffre ainsi du fait que les charges de communication ne sont pas distribuées d'une manière équilibrée.

Il est possible de paralléliser les opérations *Réduction/Broadcast*, typiquement en utilisant les opérations *ReduceScatter/AllGather*. On remplacera l'opération de *Réduction*, qui met le résultat combiné dans un nœud, par l'opération *ReduceScatter*, qui divisera le résultat combiné en plusieurs parties, chacune combinée dans un nœud [Tho12b, section 13.2].

Cette parallélisation apporte deux avantages : le premier est que tous les nœuds participeront aux opérations de communication, le second est que la taille des fragments à communiquer dans les étapes analogues à la réduction et au broadcast sera divisée par t. Toutefois, en utilisant un schéma parallèle, la sortie de l'itération sera permutée, i.e., les fragments du vecteur de sortie v ne seront pas distribués de la même manière que l'étaient les fragments du vecteur d'entrée u, au début de l'itération [Tho12a].

Reprenons dans la figure 4.3 l'exemple de la matrice distribuée sur 4 nœuds et détaillons l'exécution du schéma de Calcul/Communication avec les opérations *ReduceScatter/AllGather*. Maintenant, dans la partie gauche de la figure, nous considérons t^2 fragments pour chaque vecteur tels que :

$$u_0 = \begin{bmatrix} u_{00} \\ u_{01} \end{bmatrix}, \; u_1 = \begin{bmatrix} u_{10} \\ u_{11} \end{bmatrix}, \; v_0 = \begin{bmatrix} v_{00} \\ v_{01} \end{bmatrix} \text{ et } v_1 = \begin{bmatrix} v_{01} \\ v_{11} \end{bmatrix}$$

Nous observons qu'à la sortie de l'itération, les fragments du vecteur v ont été permutés. Ainsi, au lieu d'avoir effectué $v = Au$, nous avons calculé $v = \Sigma_t Au$, où Σ_t est une matrice de permutation qui dépend du paramètre t, ce qui n'affecte pas la résolution du système, comme nous l'avons déjà vu avec l'équilibrage des charges de travail dans la section 4.2.

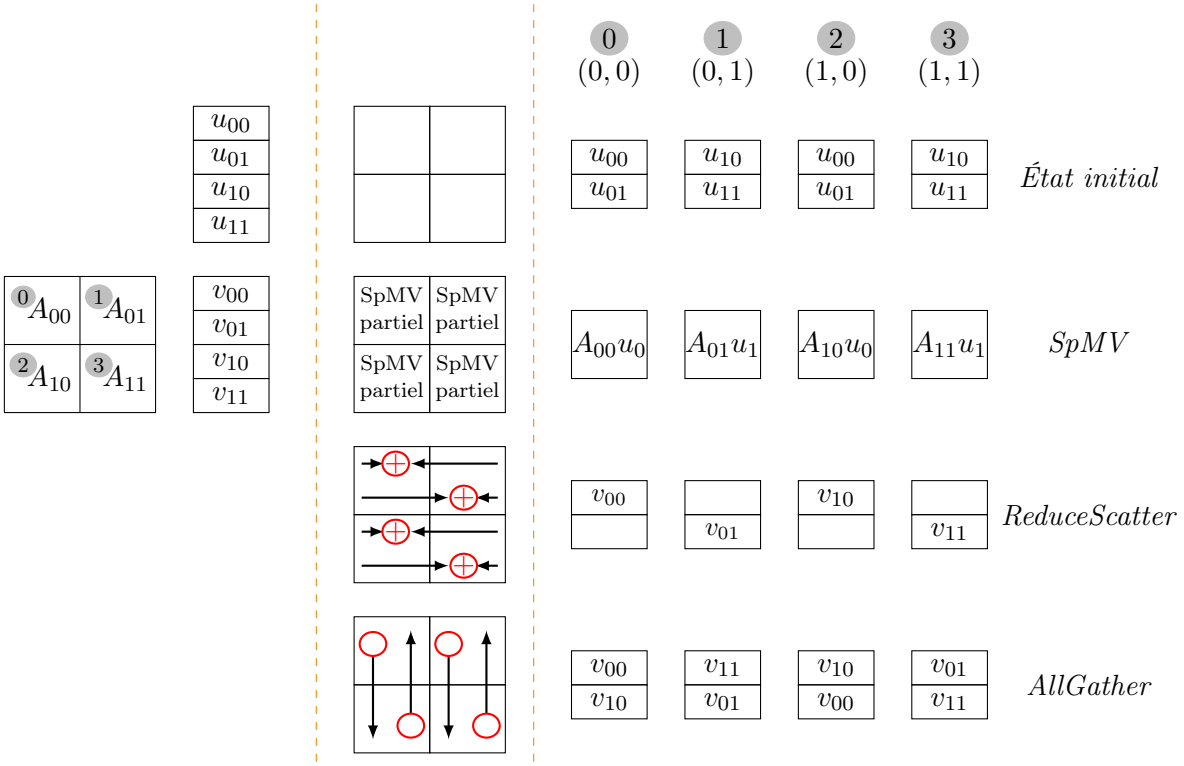


FIGURE 4.3 – Schéma de calcul/communication pour une division 2×2 de la matrice, en utilisant les opérations *ReduceScatter/AllGather*.

Dans ce chapitre, nous considérons le produit partiel sous-matrice–vecteur comme une *boîte noire*, c'est-à-dire une sous-routine, qui prend une sous-matrice et un fragment et qui retourne leur produit. Le chapitre 5 détaille comment cette sous-routine est implémentée. Maintenant, nous nous intéressons au partage des données entre les nœuds de calcul.

4.4 Communication entre les nœuds de calcul

Nous utilisons le modèle de passage de messages (avec l'interface MPI) pour implémenter la parallélisation du produit matrice-vecteur. Le choix de MPI est justifié par le fait qu'il est adapté pour des communications aussi bien dans le cas où les nœuds de calcul sont des GPU que dans le cas où les nœuds correspondent à des CPU. Toutefois, on souligne que MPI n'est pas le modèle le plus efficace pour les communications CPU intra-nœud (nous avons déjà abordé ce point dans la sous-section 2.4.2).

Chaque processus MPI est exécuté sur un *nœud* et effectue un produit partiel. Nous utilisons les *Communications Collectives* pour l'échange de données entre les processus. Nous rappelons qu'une *Communication Collective* est une méthode de communication qui fait intervenir tous les processus MPI qui appartiennent à un ensemble donné, appelé *Communicateur*. Ici, nous utilisons en particulier les collectives :

- `MPI_Reduce` : un processus collecte et combine des données réparties sur plusieurs processus.
- `MPI_Bcast` : un processus diffuse des données vers plusieurs processus.
- `MPI_Reduce_Scatter` : il s'agit d'une réduction, où le résultat final est dispersé sur plusieurs processus.
- `MPI_AllGather` : il s'agit de collecter, par plusieurs processeurs, des données réparties.

4.5 Répartition des processus MPI

Nous avons déjà introduit dans la section 4.1 la notion abstraite de *nœud* de calcul, qui correspond à une unité qui exécute un processus MPI et qui peut correspondre à un GPU ou un cœur d'un CPU. Maintenant, nous allons utiliser la notion de nœud du cluster, qui est plutôt une définition matérielle et qui correspond à une machine dans le cluster et qui peut contenir plusieurs GPU ou plusieurs cœurs et qui peut par conséquent exécuter plusieurs processus MPI.

Si nous considérons les communications entre deux processus MPI qui sont exécutés en parallèle sur le cluster, nous distinguons deux cas, le cas *intra-nœud* où les deux processus sont exécutés sur un même nœud du cluster et le cas *inter-nœud* où les deux processus sont exécutés sur deux nœuds distincts. Les communications *intra-nœud* sont plus efficaces que les communications *inter-nœud*, vu que les communications *inter-nœud* sont effectuées à travers le réseau, alors que les communications *intra-nœud* sont effectuées à travers la mémoire partagée, s'il s'agit de cœurs d'un CPU ou à travers le PCI Express (PCIe), s'il s'agit de GPU (voir section 2.4). Lorsque des opérations de communication sont effectuées entre plusieurs processus MPI, il est possible d'avoir simultanément les deux cas. Dans cette section, nous allons discuter la stratégie de répartition des processus MPI de sorte à tirer avantage le plus possible de l'efficacité des communications *intra-nœud* par rapport aux communications *inter-nœud*.

Pour illustrer cette stratégie, nous considérons un exemple, celui d'une parallélisation d'un produit matrice-vecteur sur un ensemble de 4 nœuds ; chaque nœud est composé de 2 processeurs Intel Xeon E5-2650 (2 GHz) avec 8 cœurs dans chaque processeur, soit 16 cœurs par nœud. Les

nœuds sont connectés par des connexions InfiniBand FDR à 56 Gbit/s (voir sous-section 2.4.1). Le calcul du produit matrice-vecteur est effectué en parallèle par 64 processus MPI. Nous utilisons la matrice décrite dans la table 7.8, qui contient 7 M lignes et colonnes, et la bibliothèque OpenMpi-1.7.3 et nous allons mesurer les latences des opérations de collecte dans une ligne et de diffusion dans une colonne. Le but est de minimiser les temps de communications, par conséquent la somme des ces deux opérations.

Dans la sous-figure 4.4a, est illustrée la répartition par défaut des 64 processus. Avec cette configuration, nous avons obtenu les temps suivants :

— temps de la collecte : 0.18 s ;
— temps de la diffusion : 0.65 s.

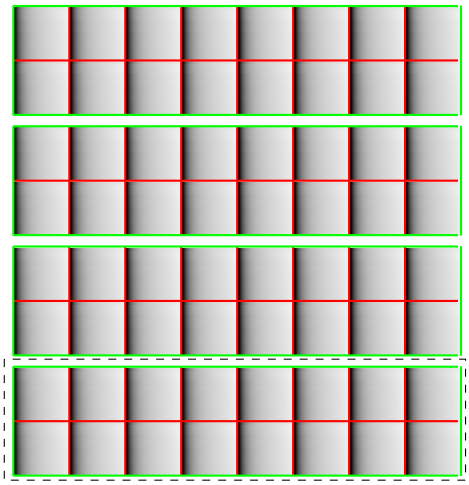
La collecte des résultats dans une ligne est « rapide » vu que tous les processus s'exécutent sur le même nœud, alors que la diffusion dans une colonne est « lente », vu qu'elle se fait à travers le réseau par des processus s'exécutant sur les 4 nœuds.

Nous proposons de répartir les processus MPI de sorte à ce que les processus qui sont exécutés sur des cœurs appartenant à un même nœud soient groupés en blocs carrés, c'est-à-dire lorsqu'on minimise les frontières entre les processus s'exécutant sur des nœuds distincts (voir la sous-figure 4.4b). Avec cette répartition, nous diminuons les communications *inter-nœuds* au profit des communications *intra-nœuds*. Ce qui donne les mesures suivantes :

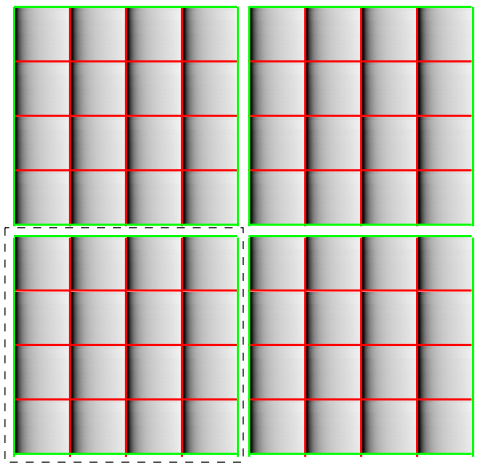
— temps de la collecte : 0.22 s ;
— temps de la diffusion : 0.18 s.

Que ce soit pour la collecte dans une ligne ou la diffusion dans une colonne, les 8 processus MPI qui communiquent appartiennent à deux groupes, où chaque groupe est exécuté sur un nœud.

La seconde répartition donne une accélération d'un facteur 2.1 sur la somme des latences de deux opérations de communication, par rapport à la répartition par défaut.



(a) Répartition par défaut des processus.



(b) Répartition choisie des processus.

FIGURE 4.4 – Répartition des processus MPI sur les cœurs pour une matrice divisée en 8×8 et distribuée sur 4 nœuds, chacun contenant 16 cœurs.

4.6 Conclusion

Dans ce chapitre, nous avons discuté de la parallélisation du produit matrice-vecteur sur plusieurs nœuds de calcul, ce qui correspond au second niveau de parallélisme pour le calcul de l'algèbre linéaire. Nous avons détaillé le produit matrice-vecteur parallèle et les stratégies pour l'accélérer, en considérant le produit partiel du *nœud* comme une boîte noire. Dans les chapitres suivants, nous allons étudier comment effectuer ce produit partiel et détailler les niveaux de parallélisme qui lui sont associés.

Chapitre 5

Produit matrice-vecteur

Dans le chapitre précédent, nous avons étudié la problématique de la parallélisation du produit matrice-creuse–vecteur sur plusieurs nœuds. Dans ce chapitre, nous allons nous intéresser au produit (partiel) sur un nœud. En particulier, nous allons nous intéresser aux questions suivantes :

— comment représenter la matrice creuse ?
— comment effectuer le produit matrice-vecteur ?

Le format de stockage de la matrice est important, d'une part parce qu'il permet de réduire la mémoire nécessaire pour représenter la matrice, d'autre part parce qu'il est lié à la stratégie de traitement des coefficients non nuls. Ainsi, les deux questions sont reliées.
Dans ce chapitre, nous allons présenter les formats classiques de stockage des matrices creuses et les produits matrice-creuse–vecteur (Sparse-Matrix–Vector product, SpMV) correspondants sur GPU et sur CPU. Nous allons expliquer comment adapter ces SpMV au contexte des corps finis de grande caractéristique et présenter des améliorations qui tiennent compte des spécificités de la matrice et des caractéristiques des architectures utilisées.
Les résultats de ce chapitre ont été présentés dans [Jel14a].

Sommaire

Dans la littérature, la problématique du produit matrice-creuse–vecteur (SpMV) a été étudiée pour différentes architectures :
— les CPU standards [BBC+94, WOL+07, GKA+09] ;
— les GPU [BG08, VGM+09] ;
— les architectures *many-cœurs* (MIC) [EKÇ13, LSC+13].

Les difficultés liées à cette problématique sont :
— comment réduire la mémoire nécessaire à la représentation de la matrice ?
— comment réduire les pénalités dues au caractère creux de la matrice, qui engendrent des accès irréguliers au vecteur d'entrée ?
— comment optimiser l'utilisation des caches ?
— comment équilibrer les charges de travail dans le cas où il y a plusieurs unités de calcul (typiquement dans le cas GPU) ?

Le cas que nous étudions a des spécificités (voir chapitre 3) qui font que les travaux du contexte numérique ne s'appliquent pas bien, même si certaines préoccupations sont communes.

5.1 Travaux et bibliothèques pour l'algèbre linéaire

Il existe un nombre significatif de travaux, de spécifications et de réalisations logicielles pour optimiser les briques de base d'algèbre linéaire. Ces travaux et réalisations ont d'abord ciblé le contexte numérique avec des matrices denses. Plus tard, le spectre a été élargi au contexte exact (entier, rationnel, polynômes) en considérant aussi des matrices creuses.

Parmi ces réalisations sur CPU, on peut donner quelques exemples :
— L'interface BLAS (Basic Linear Algebra Subprograms) qui présente un ensemble de sous-routines de l'algèbre linéaire pour optimiser les opérations sur les vecteurs, les produits matrice-vecteur et les produits matrice-matrice [LHK+79]. Les BLAS ont été mises en

œuvre à travers différentes implémentations de référence, particulièrement pour les langages C et Fortan.
— La bibliothèque LinBox qui est une une bibliothèque C++ qui fournit différentes primitives d'algèbre linéaire symbolique, y compris le produit matrice-vecteur sur un corps fini avec des matrices denses, creuses ou structurées [LinBox].
— La bibliothèque FFLAS-FFPACK (Finite Field Linear Algebra Subroutines) est une bibliothèque qui utilise les BLAS pour implémenter des opérations d'algèbre linéaire sur les corps finis [FFLAS].
— La bibliothèque MUMPS (MUltifrontal Massively Parallel sparse direct Solver) qui résout en parallèle des systèmes linéaires dans le contexte numérique, en utilisant les méthodes directes [MUMPSa, MUMPSb].

Il existe un certain nombre de réalisations logicielles spécifiques à l'algèbre linéaire issue des calculs de logarithme discret. On peut mentionner, sans prétendre l'exhaustivité, le module d'algèbre linéaire dans le logiciel CADO-NFS [CADO] et le travail de Giorgi et Vialla qui est en cours d'intégration dans la bibliothèque FFLAS-FFPACK [GV14].

Sur GPU, on mentionne la bibliothèque CUSP qui optimise les calculs des graphes et de l'algèbre linéaire creuse dans le contexte numérique pour les plateformes NVIDIA [CUSP]. Cette bibliothèque propose différents formats de représentation de la matrice creuse et algorithmes de produit matrice-creuse–vecteur, qui nous ont beaucoup inspiré. Ces structures de données et algorithmes sont à disposition de l'utilisateur qui choisit le format et l'algorithme les plus adaptés à son type de matrice.

Les objectifs du développement des BLAS et des implémentations logicielles correspondantes sont l'optimisation des routines de l'algèbre linéaire en garantissant une portée généraliste (par rapport au contexte applicatif des systèmes linéaires considérés) et la portabilité (par rapport aux architectures sur lesquelles les calculs sont effectués). Notre approche se distingue du fait qu'elle privilégie l'efficacité pour résoudre un type spécifique de systèmes linéaires sur deux types d'architecture donnés. Toutefois, pour pouvoir positionner notre approche par rapport à l'existant, nous proposons dans la section 5.8 une comparaison des performances de notre implémentation avec certaines des bibliothèques qui fournissent un SpMV sur un corps fini $\mathbb{F}_\ell$, avec ℓ tenant sur plusieurs mots machine.

5.2 Formats de représentation de la matrice creuse

Notations

Les entrées sont une matrice creuse A et un vecteur dense u. La sortie est un vecteur dense v tel que $v = Au$. La matrice A correspond à la matrice complète du système linéaire qu'on cherche à résoudre, si on n'exploite pas de parallélisme sur plusieurs nœuds comme nous l'avons décrit dans le chapitre 4. Dans le cas où on parallélise le calcul sur plusieurs nœuds, la matrice A correspond à la sous-matrice associée au nœud. On note que la densité dans la sous-matrice est la même que pour la matrice totale. N désigne la dimension de A et n_{NZ} le nombre total de ses coefficients non nuls. Les lignes et les colonnes sont indexés de 0 à $N-1$ (plutôt que de 1 à N). Cette remarque est aussi valable pour les éléments des vecteurs u et v.

5.2.1 Coordinate (COO)

Le format COO est composé de 3 tableaux `row_id`, `col_id` et `data` de n_{NZ} éléments. Les indices de ligne/colonne ainsi que la valeur sont explicitement stockés pour définir un coefficient non nul de la matrice. Les coefficients non nuls peuvent être dans n'importe quel ordre. Ici, on propose de les ordonner par leur indice de ligne.

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) matrice creuse A

$$\begin{aligned} \texttt{data} &= \begin{bmatrix} a_{01} & a_{03} & a_{11} & a_{14} & a_{15} & a_{20} & a_{22} & a_{23} & \dots \end{bmatrix} \\ \texttt{row\_id} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & \dots \end{bmatrix} \\ \texttt{col\_id} &= \begin{bmatrix} 1 & 3 & 1 & 4 & 5 & 0 & 2 & 3 & \dots \end{bmatrix} \end{aligned}$$

(b) sa représentation COO

5.2.2 Compressed Sparse Row (CSR)

Le format CSR stocke les indices de colonne et les valeurs des coefficients non nuls de A dans 2 tableaux : `id` et `data`, chacun de longueur n_{NZ}. Un troisième tableau de pointeurs, `ptr`, de taille $N + 1$, indique le début et la fin d'une ligne. Les coefficients non nuls sont ordonnés par leur indice de ligne. Le format CSR élimine le stockage explicite de l'indice de ligne, et de fait réduit la quantité de mémoire nécessaire au stockage de la matrice.

Ce format est convenable pour un accès direct à n'importe quelle ligne de la matrice, vu que `ptr` indique où chaque ligne commence et se termine dans les deux autres tableaux.

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) matrice creuse A

$$\begin{aligned} \texttt{data} &= \begin{bmatrix} a_{01} & a_{03} & a_{11} & a_{14} & a_{15} & a_{20} & a_{22} & a_{23} & \dots \end{bmatrix} \\ \texttt{id} &= \begin{bmatrix} 1 & 3 & 1 & 4 & 5 & 0 & 2 & 3 & \dots \end{bmatrix} \\ \texttt{ptr} &= \begin{bmatrix} 0 & 2 & 5 & 8 & \dots \end{bmatrix} \end{aligned}$$

(b) sa représentation CSR

5.2.3 ELLPACK (ELL)

Le format ELL étend les tableaux du format CSR à des tableaux N-par-K, où K correspond au nombre maximal de coefficients non nuls par ligne. Pour les lignes qui ont moins de K coefficients non nuls, on procède à un *padding* (remplissage). Les éléments sont ordonnés par leurs indices de colonne. Comme les lignes ont toutes la même longueur (la longueur correspond au nombre de coefficients non nuls) après le *padding*, les indices de ligne peuvent être retrouvés à partir de la position de l'élément. Seuls les indices de colonne sont explicitement stockés.

Ce format souffre du surcoût dû au *padding*, lorsque le nombre moyen de coefficients non nuls par ligne est très petit devant K. Une optimisation a été proposée par Vázquez et al. avec le

format dit ELLPACK-R (ELL-R) [VGM+09]. Cette variante rajoute un tableau `len` de longueur N qui indique le nombre de coefficients non nuls dans chaque ligne. Ainsi, les éléments rajoutés par le *padding* ne sont pas considérés lorsque le produit matrice-vecteur est effectué, mais on continue à avoir le surcoût du *padding* en mémoire.

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) matrice creuse A

$$\texttt{data} = \begin{bmatrix} a_{01} & a_{03} & * \\ a_{11} & a_{14} & a_{15} \\ a_{20} & a_{22} & a_{23} \\ a_{31} & a_{34} & * \\ a_{41} & a_{42} & a_{45} \\ a_{52} & a_{55} & * \end{bmatrix} \quad \texttt{id} = \begin{bmatrix} 1 & 3 & * \\ 1 & 4 & 5 \\ 0 & 2 & 3 \\ 1 & 4 & * \\ 1 & 2 & 5 \\ 2 & 5 & * \end{bmatrix} \quad \texttt{len} = \begin{bmatrix} 2 & 3 & 3 & 2 & 3 & 2 \end{bmatrix}$$

(b) sa représentation ELL-R

5.2.4 Sliced Coordinate (SLCOO)

Le but de ce format est d'améliorer l'utilisation du cache qui limite les performances des autres formats. Ce format s'inspire du logiciel `CADO-NFS` [CADO] pour l'algèbre linéaire sur CPU et a été introduit pour les GPU par Schmidt et al. dans le contexte de la factorisation d'entiers, où les matrices sont sur $\mathbb{F}_2$ [SAD11].

La matrice est divisée en tranches horizontales, où les coefficients non nuls sont ordonnés par leur indice de colonne dans le but de réduire les accès irréguliers au vecteur d'entrée u, par rapport aux accès si les coefficients étaient ordonnés par leur indice de ligne. Comme le format COO, le format SLCOO stocke explicitement les indices de ligne et de colonne et la valeur. Un quatrième tableau `ptrSlice` indique le début et la fin de chaque tranche. On appelle ce format SLCOO-σ, où le paramètre σ désigne le nombre de lignes de chaque tranche.

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) matrice creuse A

$$\begin{aligned} \texttt{data} &= \begin{bmatrix} a_{01} & a_{11} & a_{03} & a_{14} & a_{15} & a_{20} & a_{31} & a_{22} & a_{23} & a_{34} & \dots \end{bmatrix} \\ \texttt{row\_id} &= \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 2 & 3 & 2 & 2 & 3 & \dots \end{bmatrix} \\ \texttt{col\_id} &= \begin{bmatrix} 1 & 1 & 3 & 4 & 5 & 0 & 1 & 2 & 3 & 4 & \dots \end{bmatrix} \\ \texttt{ptrSlice} &= \begin{bmatrix} 0 & 5 & 10 & \dots \end{bmatrix} \end{aligned}$$

(b) sa représentation SLCOO-2

5.2.5 Diagonal (DIA)

Ce format s'applique dans le cas où les coefficients non nuls sont sur les diagonales de la matrice. Le format est représenté par deux tableaux : `data` qui stocke les valeurs et `offset` qui stocke le décalage de chaque diagonale par rapport à la diagonale principale.

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) matrice creuse A

$$\texttt{data} = \begin{bmatrix} * & * & * & a_{01} & a_{03} & * \\ * & * & a_{11} & * & a_{14} & a_{15} \\ * & a_{20} & a_{22} & a_{23} & * & * \\ * & a_{31} & * & a_{34} & * & * \\ a_{41} & a_{42} & * & a_{45} & * & * \\ a_{52} & * & a_{55} & * & * & * \end{bmatrix} \quad \texttt{offset} = \begin{bmatrix} -3 & -2 & 0 & 1 & 3 & 4 \end{bmatrix}$$

(b) sa représentation DIA

L'efficacité d'un format par rapport aux autres, en terme d'efficacité des accès (des schémas d'accès qui sont *cache-friendly*), dépend de la distribution des coefficients non nuls. La structure que peut avoir la matrice peut privilégier un format. Par exemple, le format DIA a les meilleures performances sur des matrices structurées, où les coefficients non nuls s'alignent sur des diagonales. Une matrice extrêmement creuse sera plutôt adaptée au format COO. Une matrice dont le nombre d'éléments non nuls par ligne ne varie pas beaucoup aura de meilleures performances avec le format ELL. Il est aussi envisageable de combiner deux ou plusieurs formats pour une même matrice en fonction de la densité des parties.

5.3 Produits matrice-vecteur sur GPU

À présent, nous discutons la stratégie de traitement des coefficients non nuls pour effectuer le produit matrice-vecteur. Dans cette section, on se restreint aux implémentations du SpMV pour les GPU NVIDIA (voir la section 2.3 pour un rappel de la programmation avec les GPU NVIDIA). On sera confronté aux problématiques des accès mémoire irréguliers, du déséquilibre entre les unités calculatoires du GPU et de l'utilisation efficace du cache.

Pour chaque format, on décrit comment s'effectue le produit matrice-vecteur. Pour certains formats, on donne le pseudo-code correspondant. Des éléments quantitatifs de comparaison sont donnés et détaillés dans la section 5.5.

Notations

Les entrées du SpMV sont les données relatives à la matrice, ainsi que le vecteur u. La sortie est le vecteur v. On note x_i la i^{e} composante (élément) d'un vecteur x. Les entrées et la sortie sont placées en mémoire globale, vu que leur taille est importante pour les autres mémoires disponibles sur le GPU. Les résultats temporaires sont stockés dans les registres. La mémoire partagée est utilisée quand on a besoin de combiner des résultats partiels de plusieurs threads, ces opérations sont désignées parfois par le terme *réduction*.

Dans cette section, nous supposons que les éléments de la matrice, ainsi que ceux des vecteurs u et v, sont dans un corps $\mathbb{K}$ (réels, corps finis, ...). À ce niveau, on fait abstraction de la nature du corps, pour simplifier la présentation des algorithmes. Plus loin, dans la section 5.4, nous allons détailler les considérations liées au fait de travailler dans un corps fini de grande caractéristiques. Les données relatives à la position des éléments (indice de ligne, indice de colonne, pointeur de début de ligne, ...) sont des entiers positifs (16, 32 ou 64 bits, selon la taille de la matrice). Les opérations arithmétiques sont désignées dans le pseudo-code par la fonction `addmul()` (`d←addmul(a,b,c)` correspond à `d←a+b×c` dans le corps $\mathbb{K}$).

5.3.1 COO

La manière typique pour utiliser le format COO sur GPU est que les 32 *threads* d'un *warp* traitent 32 coefficients consécutifs non nuls. Ainsi, ils itèrent sur un intervalle de 32 coefficients non nuls. Ceci implique que des *threads* appartenant à plusieurs *warps* travaillent sur une même ligne, lorsqu'il existe plus de 32 coefficients non nuls par ligne.

Pour combiner les résultats des threads, deux possibilités sont envisageables :

— La première possibilité est d'effectuer des mises à jour atomiques dans la mémoire globale, ce qui dégrade considérablement les performances.

— La seconde possibilité est que chaque *thread* calcule son résultat partiel, ensuite une *réduction* segmentée [SHZ+07, BHZ93] est effectuée pour sommer les résultats des *threads* appartenant à un même *warp* et travaillant sur une même ligne. Nous nous sommes inspiré du schéma proposé dans la bibliothèque CUSP [CUSP] qui effectue une *réduction* segmentée dans la mémoire partagée, avec les indices des lignes comme descripteurs de segment. Quand tous les *warps* ont terminé, nous combinons leurs résultats partiels. On rappelle que la synchronisation entre les *warps* n'est pas supportée et qu'on a besoin d'attendre la fin d'exécution de tous les *warps* (voir la sous-section 2.3.6).

Les principaux inconvénients du SpMV COO sont le coût de la combinaison des résultats partiels et l'utilisation excessive de la mémoire globale. Son avantage est que les *warps* sont équilibrés, vu qu'ils itèrent sur un intervalle de longueur constante.

5.3.2 CSR

Pour paralléliser le produit pour le format CSR, une façon simple est d'associer un *thread* à une ligne. Cette approche est dite *scalaire* [NBG+08] et on la note CSR-S. Dans le pseudo-code suivant, on illustre le produit matrice-vecteur CSR-S. Le *thread* travaille sur la ligne d'indice `row`. Pour chaque élément non nul de la ligne, le *thread* effectue une lecture de la mémoire globale, une opération arithmétique `addmul` et une écriture dans des registres. Ainsi, le résultat temporaire est stocké dans des registres et à la fin de la ligne, le résultat final est écrit en mémoire globale.

Algorithm 5.1 SpMV CSR-S pour la ligne `row`, exécuté par un *thread*.

Entrées : `data` : tableau de n_{NZ} éléments de $\mathbb{K}$
`id` : tableau de n_{NZ} entiers positifs
`ptr` : tableau de N entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

`sum` $\leftarrow 0$ // élément de $\mathbb{K}$ en registres
Pour $i \leftarrow$ `ptr`$_{\texttt{row}}$ **à** `ptr`$_{\texttt{row+1}}-1$ **faire**
| `sum` $\leftarrow$ `addmul(sum,data`$_i$`,`$u_{\texttt{id}_i}$`)`
$v_{\texttt{row}} \leftarrow$ `sum` // écriture en mémoire globale

Une autre approche, dite *vectorielle*, consiste à associer un *warp* à une ligne [BG08]. On note ce produit CSR-V. Les *threads* d'un même *warp* accèdent à des coefficients voisins, ce qui fait que les accès à `id` et `data` deviennent contigus. Chaque *thread* calcule sa somme partielle, ensuite une *réduction* dans la mémoire partagée est effectuée pour combiner tous les résultats des *threads*. On n'a pas besoin de spécifier une barrière de synchronisation, vu que les *threads* appartenant à un même *warp* sont exécutés physiquement en même temps. Passer par une

barrière de synchronisation aurait été nécessaire si nous avions associé un *block* de plusieurs *warps* à une ligne.

Algorithm 5.2 SpMV CSR-V pour la ligne `row`, exécuté par le *thread* d'indice `lane` dans son *warp*.

Entrées : `data` : tableau de n_{NZ} éléments de $\mathbb{K}$
`id` : tableau de n_{NZ} entiers positifs
`ptr` : tableau de N entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

```
sum ← 0
i ← ptr_row + lane                          // position de début pour chaque thread
Tant que i < ptr_row+1 faire
    sum ← addmul(sum,data_i,u_id_i)
    i ← i + 32
réduction_csr_vec(sum,lane)                 // réduction en mémoire partagée
Si lane= 0 alors                            // premier thread du warp écrit en mémoire globale
    v_row ← sum
```

Par rapport au SpMV COO, les deux SpMV CSR réduisent l'utilisation de la mémoire globale et simplifient la combinaison des résultats partiels. De plus, l'exécution entre les différents *warps* est non-synchronisée. En effet, chaque *warp* peut terminer son calcul sans être synchronisé avec les autres *warps*, que ce soit avec le produit CSR-S où chaque *thread* calcule un élément du vecteur v ou bien avec le produit CSR-V où chaque *warp* calcule un élément du vecteur v. Toutefois, le SpMV COO garde l'avantage de l'équilibre de charge par rapport au CSR. Le déséquilibre de charge du CSR s'accentue si les lignes ont des longueurs très variables. Pour pallier le déséquilibre de charge du CSR, une possibilité est d'ordonner les lignes par leur longueur, les *warps* lancés simultanément ont ainsi quasiment la même charge.

Maintenant, comparons les deux SpMV CSR entre eux. Les *threads* du SpMV CSR-S ont des accès non contigus aux tableaux `data` et `id`, vu qu'ils ne travaillent pas sur les mêmes lignes. Ainsi, leurs accès mémoire ne sont pas aussi efficaces que ceux du CSR-V. Par contre, le SpMV CSR-V requiert une combinaison des résultats partiels qui augmente l'utilisation des registres et de la mémoire partagée (voir la sous-section 5.5.2).

5.3.3 ELL

Le partitionnement du travail se fait en associant un *thread* à une ligne de la matrice (voir l'algorithme 5.3). Le SpMV ELL-R prend avantage du fait que les éléments sont ordonnés par leur indice de colonne, ce qui améliore les accès au vecteur u. Toutefois, il a l'inconvénient de consommer beaucoup de mémoire, à cause du padding.

5.3.4 SLCOO

Pour le SpMV SLCOO, chaque *warp* travaille sur une tranche. Ainsi, chaque *thread* travaille sur plus d'une ligne. Soit il possède un stockage individuel pour chaque ligne, soit il a un accès exclusif à une ressource commune. Dans [SAD11], où un *block* (voir le chapitre 2) a été assigné à une tranche, trois possibilités ont été mentionnées pour résoudre cette question :

Algorithm 5.3 SpMV ELL-R pour la ligne `row`, exécuté par un *thread*.

Entrées : `data` : tableau de $K \times N$ éléments de $\mathbb{K}$
`id` : tableau de $K \times N$ entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

`sum` $\leftarrow 0$
Pour $i \leftarrow 0$ **à** $\texttt{len}_{\texttt{row}}-1$ **faire**
  `sum` $\leftarrow$ `addmul(sum,data`$_{N \times i+\texttt{row}}$`,`$u_{\texttt{id}_{N \times i+\texttt{row}}}$`)`
$v_{\texttt{row}} \leftarrow$ `sum`

— un *thread* a une entrée exclusive dans la mémoire partagée pour stocker le résultat partiel pour chaque ligne, on appelle cette approche SLCOO-*petit* ;
— les *threads* ayant le même *lane* (indice dans le *warp*) partagent une même entrée par ligne dans la mémoire partagée et accèdent à cette entrée par une opération XOR atomique (seulement sur $\mathbb{F}_2$), on appelle cette approche SLCOO-*moyen* ;
— tous les *threads* partagent une entrée pour chaque ligne, on appelle cette approche SLCOO-*grand*.

Comme la mémoire partagée est limitée, si on suppose que le SpMV SLCOO-*petit* nous permet d'avoir des tranches d'épaisseur σ, le SpMV SLCOO-*moyen* permet de mettre k fois plus de lignes par tranche, où k est le nombre de *warps* dans un *block*. D'une manière similaire, le SpMV SLCOO-*grand* permet de mettre 32 fois plus de lignes que le format SLCOO-*moyen* (32 étant le nombre de *threads* par *warp*). Ainsi, de la variante SLCOO-*petit* à la variante SLCOO-*grand*, on arrive à mettre plus de lignes par tranche. Par conséquent, on améliore le taux de succès du cache, ce qui compense les inconvénients des accès atomiques. De ce fait, comme le mentionnent Schmidt et al. dans [SAD11], il est intéressant de réordonner les lignes selon leur poids et d'utiliser le SpMV SLCOO-*petit* dans les parties les plus denses (où la probabilité d'avoir deux accès atomiques simultanés à une même ressource est grande), SLCOO-*grand* dans les parties les moins denses et le SpMV SLCOO-*moyen* pour les lignes de poids moyen.

Algorithm 5.4 SpMV SLCOO-*petit* pour la tranche `slice`, exécuté par le *thread* d'indice `lane` dans son *warp*.

Entrées : `data` : tableau de n_{NZ} éléments de $\mathbb{K}$
`row_id` : tableau de n_{NZ} entiers positifs
`col_id` : tableau de n_{NZ} entiers positifs
`ptrSlice` : tableau de N entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

Déclarer le tableau `sum` $\leftarrow \{0\}$ // tableau de σ éléments de $\mathbb{K}$
$i \leftarrow \texttt{ptrSlice}_{\texttt{slice}} + \texttt{lane}$ // position de début pour chaque thread
Tant que $i < \texttt{ptrSlice}_{\texttt{slice+1}}$ **faire**
  $\texttt{sum}_{\texttt{row\_id}_i \bmod \sigma} \leftarrow \texttt{addmul(sum}_{\texttt{row\_id}_i \bmod \sigma}\texttt{,data}_i\texttt{,}\ u_{\texttt{col\_id}_i \bmod \sigma}\texttt{)}$
  $i \leftarrow i + 32$
`réduction_slcoo(sum,lane)` // réduction en mémoire partagée
Si `lane`$= 0$ **alors** // premier *thread* du *warp* écrit en mémoire globale
  Pour $j \leftarrow 0$ **à** σ **faire**
    $v_{\texttt{slice}\times\sigma+j} \leftarrow \texttt{sum}_j$

5.3.5 Exécution des SpMV sur une matrice-jouet

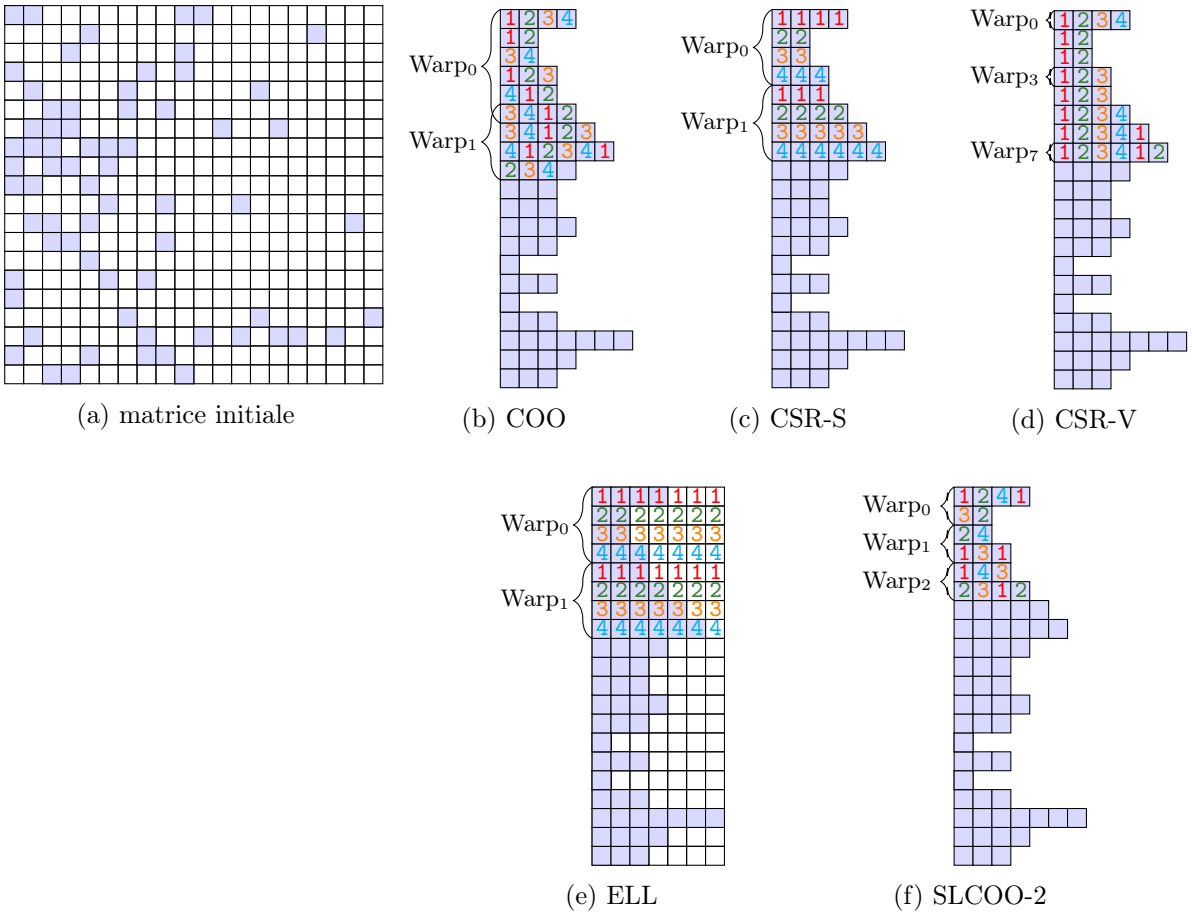
Dans la figure 5.6, nous prenons une petite matrice creuse composée de 20 lignes et colonnes et nous allons illustrer l'exécution des SpMV précédemment décrits sur cette matrice. Par souci de simplicité, on suppose qu'un *warp* contient 4 *threads* ; chaque *thread* est représenté par un nombre parmi {1, 2, 3, 4} et une couleur parmi {rouge, vert, orange, bleu}.

Pour le SpMV COO, on suppose qu'un *warp* effectue 4 itérations, c'est-à-dire traite 16 coefficients. Le premier *warp* traite les 4 premiers coefficients de la matrice, par la suite passe aux 4 suivants et ainsi de suite. On voit que la charge de travail est équilibrée entre les *threads* et les warps, mais qu'une ligne peut être traitée par plusieurs *warps* (par exemple la 6^{e}), ce qui complexifie la *réduction*.

Pour le SpMV CSR-S, on voit qu'un *thread* traite une ligne, alors qu'un *warp* traite une ligne avec le SpMV CSR-V.

Le SpMV ELL traite les coefficients de la même façon que le SpMV CSR-S, sauf que les coefficients sont ordonnés suivant leur indice de colonne, ainsi les accès mémoire du SpMV ELL sont contigus.

Avec le SpMV SLCOO-2, une tranche est composée de 2 lignes où les coefficients sont ordonnés par leur indice de colonne.



(a) matrice initiale (b) COO (c) CSR-S (d) CSR-V

(e) ELL (f) SLCOO-2

FIGURE 5.6 – Une matrice creuse et l'exécution des SpMV pour les formats COO, CSR-S, CSR-V, ELL et SLCOO-2

5.4 Produits matrice-vecteur pour les corps finis de grande caractéristique

Dans la littérature, les produits matrice-creuse–vecteur ont été étudiés et comparés pour les cas numériques (c'est-à-dire lorsque les éléments de A, u et v sont des réels qu'on représente par des nombres flottants en précision simple ou double) et les cas exacts, où l'arithmétique est modulo un petit premier qui tient sur un seul mot machine (qu'on représente dans un entier ou un flottant à précision simple ou double). En numérique, on peut citer les travaux de Bell et Garland [BG08] et de Vázquez at al. [VGM+09]. En exact, Schmidt et al. ont étudié le cas $\mathbb{F}_2$ [SAD11] et Boyer et al. se sont intéressés aux petits corps finis $\mathbb{Z}/m\mathbb{Z}$, où ils utilisent un flottant double précision pour représenter un élément du corps [BDG10].

Dans le contexte de notre application, les éléments de la matrice A sont généralement « petits » (c'est-à-dire qu'ils tiennent sur un seul mot machine) et les éléments des vecteurs u et v sont « grands » (c'est-à-dire qu'ils tiennent sur plusieurs mots machine) (voir sous-section 3.2.1). Dans cette section, nous étudions comment adapter les SpMV pour ce contexte. Étudier cette question va nous amener à invoquer le parallélisme au niveau arithmétique, que nous allons détailler dans le chapitre 6.

Notations

On suppose que le SpMV est effectué dans un corps fini $\mathbb{Z}/\ell\mathbb{Z}$. On note n le nombre de mots machine occupés par un élément de $\mathbb{Z}/\ell\mathbb{Z}$. Ainsi, traiter un coefficient non nul λ à la ligne i à la colonne j de la matrice A implique la lecture des n mots qui composent le j^{e} élément de vecteur d'entrée u, les multiplier par λ et les additionner aux n mots qui composent le i^{e} élément du vecteur de sortie v. Dans les pseudo-codes suivants, on désignera l'opération arithmétique qui s'applique à un mot par la fonction `addmul_mot()`. À ce niveau, on fait abstraction du système de représentation des nombres et de l'arithmétique qui en découle. Ces considérations sont détaillées dans le chapitre 6.

5.4.1 Schéma *séquentiel*

Une première approche serait qu'un *thread* traite un élément non nul. On appelle cette approche l'approche *séquentielle*. Pour illustrer cette approche, reprenons le SpMV CSR-V et appliquons cette approche (voir l'algorithme 5.5). Cette approche souffre de plusieurs problèmes. Le premier est que le *thread* traite les n mots machine correspondant à un élément non nul, c'est-à-dire lit et écrit les n mots machine et effectue les n opérations arithmétiques. Ainsi, le *thread* est amené à consommer plusieurs registres. Le second est que les *threads* d'un même *warp* accèdent à des zones non contiguës des vecteurs u et v, vu que leurs accès sont toujours espacés de n mots. Toutefois, on peut régler cet inconvénient en entrelaçant les données.

5.4.2 Schéma *parallèle*

Pour corriger ces faiblesses, un meilleur schéma serait qu'un élément non nul soit traité par n threads. On appelle cette approche l'approche *parallèle*. On organise les *threads* d'un même *warp* en n_{GPS} groupes de n threads, où $n_{\text{GPS}} \times n$ est le plus proche de 32, le nombre des *threads* dans un warp. Chaque groupe traite un coefficient non nul de la ligne. Par exemple, pour $n = 5$, on prend $n_{\text{GPS}} = 6$, ainsi les 5 premiers *threads* travaillent en parallèle sur les mots du 1^{er} élément

Algorithm 5.5 SpMV CSR-V-*séquentiel* pour la ligne `row`, exécuté par le *thread* d'indice `lane` dans son warp

Entrées : `data` : tableau de n_{NZ} entiers signés
`id` : tableau de n_{NZ} entiers positifs
`ptr` : tableau de N entiers positifs
u : vecteur de $N \times n$ mots machine
Sortie : v : vecteur de $N \times n$ mots machine

Déclarer le tableau `sum` $\leftarrow \{0\}$ // n mots machine initialisés à 0
$i \leftarrow \texttt{ptr}_{\texttt{row}} + \texttt{lane}$
Tant que $i < \texttt{ptr}_{\texttt{row+1}}$ **faire**
  Pour $j \leftarrow 0$ **à** $n-1$ **faire**
    $\texttt{sum}_j \leftarrow \texttt{addmul\_mot}(\texttt{sum}_j, \texttt{data}_i, u_{\texttt{id}_i \times n + j})$ // traiter le j^{e} mot machine
  $i \leftarrow i + 32$
`réduction_csr_vec_séquentiel(sum,lane)`
Si `lane`$= 0$ **alors**
  Pour $j \leftarrow 0$ **à** $n-1$ **faire**
    $v_{\texttt{row} \times n + j} \leftarrow \texttt{sum}_j$ // écrire le j^{e} mot machine en mémoire globale

du vecteur source, les *threads* 5 à 9 traitent le 2e élément, et ainsi de suite. Le *warp* aura deux *threads* inactifs.

Dans l'algorithme 5.6, on applique ce schéma au SpMV CSR-V. Ce schéma permet de réduire la complexité de la *réduction* et l'utilisation de la mémoire partagée, vu que dans le schéma *séquentiel*, on combine les résultats partiels de 32 threads, alors que dans le second schéma, on combine les résultats uniquement des *threads* appartenant à des groupes différents et travaillant sur le même mot machine, c'est-à-dire n_{GPS} *threads*.

Algorithm 5.6 SpMV CSR-V-*parallèle* pour la ligne `row`, exécuté par le *thread* d'indice `lane` dans son warp

Entrées : `data` : tableau de n_{NZ} entiers signés
`id` : tableau de n_{NZ} entiers positifs
`ptr` : tableau de N entiers positifs
u : vecteur de $N \times n$ mots machine
Sortie : v : vecteur de $N \times n$ mots machine

`sum` $\leftarrow 0$ // 1 mot machine initialisé à 0
$i \leftarrow \texttt{ptr}_{\texttt{row}} + \lfloor\, \texttt{lane} \,/\, n \,\rfloor$ // position de début pour chaque thread
Tant que $i < \texttt{ptr}_{\texttt{row+1}}$ **faire**
  `sum` $\leftarrow \texttt{addmul\_mot}(\texttt{sum}, \texttt{data}_i, u_{\texttt{id}_i \times n + \texttt{lane} \bmod n})$ // traiter 1 mot machine
  $i \leftarrow i + n_{\text{GPS}}$
`réduction_csr_vec_parallèle(sum,lane)`
Si `lane`$< n$ **alors** // premier groupe du *warp* écrit en mémoire globale
  $v_{\texttt{row} \times n + \texttt{lane}} \leftarrow \texttt{sum}$

Ici, nous avons détaillé comment appliquer les schémas *séquentiel* et *parallèle* au SpMV CSR-V. Pour les autres algorithmes de produit, les deux schémas sont applicables et le schéma *parallèle* présente toujours des améliorations considérables par rapport au schéma *séquentiel*.

5.5 Analyse comparative des produits matrice-vecteur sur GPU

Dans cette section, nous comparons les performances des SpMV et des schémas que nous avons présentés. L'objectif est de minimiser le temps du produit matrice-vecteur.

On se munit d'une matrice de taille raisonnable par rapport aux calculs qu'on effectue dans notre contexte. Cette matrice a été obtenue pour la résolution du logarithme discret dans un sous-groupe du corps $\mathbb{F}_{2^{619}}$ d'ordre un nombre premier ℓ de 217 bits, en utilisant l'algorithme FFS. Le nombre premier ℓ occupe 4 mots machine 64 bits. La table 5.1 résume les propriétés de cette matrice. On utilise la carte graphique GeForce GTX 680 (voir table 7.9) pour exécuter le produit matrice-vecteur.

Calcul	FFS pour $\mathbb{F}_{2^{619}}$
Taille de la matrice (N)	650k
Nombre de coefficients non nuls (n_{NZ})	65M
Nombre moyen de coefficients non nuls par ligne	100
Nombre maximal de coefficients non nuls par ligne	418
Pourcentage de ± 1	92.7%
Taille de ℓ (bits)	217

TABLE 5.1 – Caractéristiques de la matrice FFS pour $\mathbb{F}_{2^{619}}$.

Chaque SpMV a été exécuté 100 fois. On reporte le temps moyen du SpMV et le débit calculatoire en GOP/s, qu'on détermine en divisant le nombre d'opérations requises (2 fois le Nombre de coefficients non nuls de A multiplié par le nombre de mots de 32 bits) par le temps d'exécution. Les mesures n'incluent pas les temps de transfert entre le CPU et le GPU, puisqu'on n'a pas besoin de transférer la matrice et les vecteurs entre les itérations SpMV.

En plus des mesures de latence et de débit, nous allons considérer des métriques qui permettent d'expliquer les performances et les facteurs limitants des SpMV (voir sous-section 2.3.8) :

- le nombre de registres par *thread* ;
- la quantité de mémoire partagée par SM ;
- le taux de remplissage des SM ;
- le taux de divergence de branches ;
- l'efficacité des accès à la mémoire globale : on mesure cette efficacité en calculant le ratio des transactions mémoire demandées par rapport aux transactions mémoire effectuées, ceci reflète si les accès mémoire sont parfaitement fusionnés (efficacité de 100%) ou non ;
- le taux de succès du cache.

Les mesures pour ces métriques ont été obtenues avec les outils d'analyse et de profilage : CUDA Occupancy Calculator et `nvvp` (voir sous-section 2.3.9).

5.5.1 Comparaison des schémas *séquentiel* et *parallèle*

Pour comparer les schémas *séquentiel* et *parallèle*, nous appliquons ces deux schémas sur le SpMV CSR-V et comparons leurs performances.

D'après la table 5.2, nous notons que le SpMV *séquentiel* consomme plus de registres et de mémoire partagée, ce qui limite le nombre maximal de *warps* pouvant être lancés sur un SM à 24, alors qu'un SM peut en contenir 64. Ceci est reporté dans la colonne *remplissage théorique*. Le faible taux de remplissage dégrade les performances.

En terme d'efficacité des accès mémoire, la colonne *efficacité lecture / écriture* indique que les accès mémoire du SpMV *séquentiel* sont peu efficaces, parce qu'ils ne sont pas fusionnés. Des accès non fusionnés entraînent une perte de la bande passante et une dégradation des performances. Le SpMV *parallèle* atteint 100% d'efficacité en écriture, mais en lecture, son efficacité est limitée à 47% à cause des accès irréguliers sur le vecteur source, vu le caractère creux de la matrice.

	Registres par *thread*	Mém. Partagée par SM	Remplissage (théorique)	Efficacité lecture/écriture	Temps en ms	Débit en GOP/s
Séquentiel	27	49152	35.1% (37.5%)	7.5%/26%	141.1	11.1
Parallèle	**21**	**15360**	**70.3% (100%)**	**47.2%/100%**	**41.4**	**37.7**

TABLE 5.2 – Comparaison des schémas *séquentiel* et *parallèle* pour le SpMV CSR-V, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

Il est clair que le schéma *parallèle* est plus adapté à notre contexte de grands entiers.

5.5.2 Comparaison des SpMV CSR, COO et ELL

Maintenant, nous appliquons le schéma *parallèle* sur les formats CSR, COO et ELL et comparons leurs performances (voir table 5.3).

À cause de la *réduction* segmentée, le SpMV COO est le SpMV qui effectue le plus d'instructions et consomme le plus grand nombre de registres. La divergence de *threads* est plus fréquente, à cause des différentes branches que peuvent prendre les *threads* appartenant à un même *warp*.

En ce qui concerne le SpMV ELL, les lignes ont le même nombre de coefficients non nuls, grâce au *padding*. Les *warps*, ainsi que les *threads* sont alors bien équilibrés (voir les colonnes *remplissage* et *divergence de branches*). Le fait que les éléments sont ordonnés par leur indice de colonne fait que le SpMV ELL atteint le taux de succès du cache le plus élevé.

Le SpMV CSR-S souffre d'une faible efficacité d'accès mémoire par rapport au SpMV CSR-V. Ceci est dû au fait que dans le SpMV CSR-S, les *threads* d'un même *warp* travaillent sur plusieurs lignes en même temps. Le SpMV CSR-V satisfait le mieux les caractéristiques architecturales du GPU et atteint par conséquent les meilleures performances.

	Reg.	Diverg. branches	Remplissage (théorique)	Efficacité lecture/écriture	Taux succès du cache	Temps en ms	Débit en GOP/s
COO	25	47.1%	65.2% (66.7%)	34.3%/37.8%	34.4%	88.9	17.6
CSR-S	18	28.1%	71.8% (100%)	29.2%/42.5%	35.4%	72.3	21.6
CSR-V	**21**	**36.7%**	**70.3% (100%)**	**47.2%/100%**	**35.4%**	**41.4**	**37.7**
ELL-R	18	44.1%	71.8% (100%)	38.1%/42.5%	40.1%	45.5	34.3

TABLE 5.3 – Comparaison des performances des SpMV CSR-V, CSR-S, COO et ELL-R, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

5.5.3 Comparaison des SpMV SLCOO et CSR-*vectoriel*

Schmidt et al. ont pu dans [SAD11] implémenter et comparer pour le cas de l'arithmétique sur $\mathbb{F}_2$ les trois SpMV SLCOO : SLCOO-*petit*, SLCOO-*moyen* et SLCOO-*grand*. Toutefois, dans notre cas, les opérations atomiques disponibles sur CUDA ne sont pas suffisantes pour faire

des opérations atomiques dans $\mathbb{Z}/\ell\mathbb{Z}$. Ainsi, seul le SpMV SLCOO-*petit*, qui ne requiert pas d'opérations atomiques, peut être implémenté.

Dans la table 5.4, on compare les performances du SpMV CSR-V avec celles du SpMV SLCOO-*petit* : SLCOO-2, SLCOO-4, SLCOO-8, pour des tranches de 2, 4 et 8 lignes. On note que le CSR-V peut être considéré comme un SLCOO-1.

On remarque qu'en augmentant l'épaisseur de la tranche, le taux de succès du cache s'améliore, vu que les accès au vecteur source sont moins irréguliers. Toutefois, en élargissant la tranche, on augmente l'utilisation de la mémoire partagée proportionnellement à l'épaisseur, ce qui diminue le nombre maximal de *blocks* pouvant s'exécuter en parallèle. Cette limitation du remplissage impacte directement les performances.

	Mém. Partagée par Bloc	# *Blocks* par SM	Remplissage (théorique)	Taux de succès du cache	Temps en ms	Débit en GOP/s
CSR-V	**1920**	**8**	**70.3% (100%)**	**35.4%**	**41.4**	**37.7**
SLCOO-2	3840	4	48.4% (50%)	36.1%	46.9	33.3
SLCOO-4	7680	2	24.5% (25%)	36.9%	58.6	26.6
SLCOO-8	15360	1	12.3% (12.5%)	37.8%	89.9	17.4

TABLE 5.4 – Comparaison des performances des SpMV CSR-V et de plusieurs SpMV SLCOO-σ pour différentes épaisseurs de tranche σ, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

La quantité de mémoire partagée nécessaire est proportionnelle au nombre de mots nécessaires pour représenter ℓ. Dans l'expérience précédente, le nombre de mots pour représenter la caractéristique du corps ℓ était 4. On propose de voir l'évolution des performances de ces SpMV, en variant la taille de ℓ et par conséquent le nombre de mots nécessaire (voir figure 5.7).

On voit que pour une taille de ℓ de 2 ou de 3 mots machine, le SpMV SLCOO-2 consomme moins de mémoire partagée. Par conséquent, il arrive à atteindre un meilleur remplissage et ses performances sont meilleures que celles du CSR, vu que son taux de succès du cache est toujours supérieur. Ceci est valable pour le SpMV SLCOO-4 pour des ℓ de 2 mots machine, alors que le SpMV SLCOO-8 consomme beaucoup de mémoire partagée et n'est par conséquent jamais compétitif avec le SpMV CSR. Nous n'avons pas montré les résultats pour des tailles de ℓ plus grandes que 250 bits, car à partir de cette taille, tous les SpMV SLCOO sont pénalisés par l'utilisation excessive de la mémoire partagée et ne sont pas compétitifs avec le SpMV CSR.

La comparaison et les résultats précédents sont valables pour les cartes graphiques NVIDIA issues des générations Fermi et Kepler. En effet, la mémoire partagée est limitée à 16 ko (on peut l'étendre à 48 ko, au dépens du cache L1). Pour la nouvelle génération Maxwell, les cartes disposent d'une mémoire partagée de 64 ko et de 96 ko. Ainsi, on peut s'attendre, avec l'augmentation de la mémoire partagée, à ce que le format SLCOO devienne plus compétitif avec le format CSR.

De ce qui précède, on voit que le SpMV CSR-V avec le schéma *parallèle* permet d'atteindre les meilleures performances (sauf pour des ℓ de taille petite, où le SLCOO-2 est plus rapide). Toutefois, on observe que ce SpMV souffre toujours de la divergence de *threads* (37%), d'un remplissage réel (70%) plus faible que le remplissage théorique (100%) et d'une efficacité en lecture limitée à 47%. Nous présentons dans la prochaine section des améliorations qui réduiront ces problèmes.

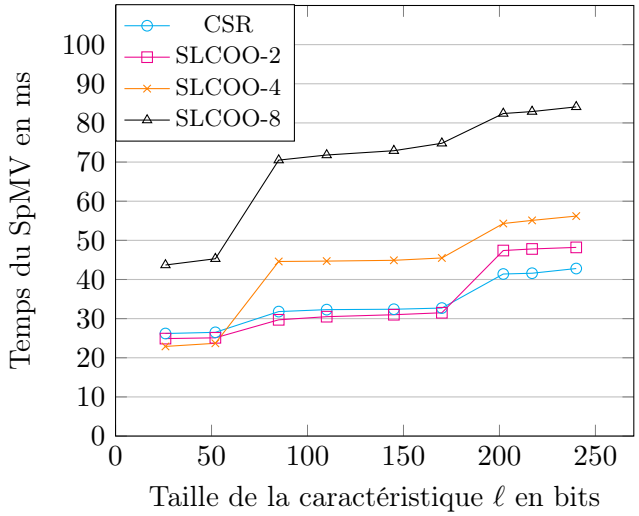


FIGURE 5.7 – Performances des SpMV CSR et SLCOO-σ en fonction de la taille de la caractéristique, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

5.6 Améliorations pour le produit CSR-V

Certaines des améliorations que nous présentons dans cette section sont applicables pour n'importe quel SpMV, d'autres sont spécifiques au SpMV CSR-V. Nous reportons dans la table 5.5 les impacts et les accélérations relatives à ces améliorations.

5.6.1 Cache texture

Même si notre SpMV souffre d'accès irréguliers en lecture, les *threads* d'un même groupe font des lectures dans des zones voisines. C'est pourquoi il est intéressant de placer le vecteur source dans la mémoire texture et d'exploiter le cache texture en remplaçant les lectures par des *fetches* (voir paragraphe 2.3.5). Ceci améliore l'efficacité de lecture.

5.6.2 Réordonner les coefficients non nuls d'une ligne

Dans les matrices issues du calcul de logarithme discret, la plupart des coefficients non nuls sont ± 1 (voir table 5.1). Ainsi, il paraît intéressant de les traiter différemment des autres coefficients, vu que les additions et soustractions sont moins coûteuses que les multiplications. De plus, nous n'avons pas le même code pour les coefficients positifs et négatifs. Toutes ces séparations entraînent de la divergence de *threads*. Pour limiter cette divergence, on propose de réordonner les coefficients non nuls d'une ligne de sorte à ce que les coefficients de la même catégorie $(+1, -1, > 0, < 0)$ soient contigus. Ceci diminue la divergence.

5.6.3 Compresser le tableau de valeurs `data`

Ayant réordonné les coefficients non nuls de la ligne, comme nous avons une majorité de ± 1, il est possible de remplacer les ± 1 par le nombre de leurs occurrences. D'une part, ceci divise la longueur du tableau `data` par un facteur 10 (si on a $\sim$90% de ± 1), ce qui diminue la consommation de la mémoire globale (très important quand on arrive à des grandes matrices).

D'autre part, on diminue le nombre de lectures. Nous aurons besoin d'un autre tableau `ptr_data`, de longueur $N + 1$, qui indique le début et la fin de chaque ligne dans le tableau `data`.

5.6.4 Améliorer l'équilibre des warps

Dans le SpMV CSR-V, chaque *warp* traite une ligne. Ceci nécessite qu'on lance un grand nombre de warps. Par conséquent, il y a un délai pour ordonnancer les *warps* lancés. Nous proposons qu'un *warp* itère sur un certain nombre de lignes. Toutefois, ceci fait augmenter le nombre de registres utilisés (de 19 par *thread* à 24). Pour continuer à améliorer le remplissage, on fait une permutation des lignes de sorte à ce que chaque *warp* ait quasiment la même charge de travail.

Optimisation	Effets	Temps en ms	Débit en GOP/s (accélération)
Cache texture	Efficacité lecture : 47.2% → 84%	32	48.8 (+30%)
Réordonner les coefficients	Divergence de branches : 36.7% → 12.9%	30.5	51.2 (+5%)
Compresser `data`	Nombre d'instructions exécutées : $5.8 \times 10^8 \rightarrow 5.72 \times 10^8$	27.6	56.6 (+11%)
Plusieurs lignes par warp	Remplissage (théorique) : 70.3% (100%) → 74.9% (83.3%)	27.4	56.9 (+0.5%)
Permutation des lignes	Remplissage (théorique) : 74.9% (83.3%) → 81.8% (83.3%)	**27.1**	**57.7** (+1%)

TABLE 5.5 – Effets des améliorations pour le SpMV CSR-V et leurs accélérations, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

5.7 Produits matrice-vecteur sur CPU

Dans cette section, nous supposons disposer d'un seul CPU et nous explorons comment réaliser le produit matrice-vecteur. Les questions relatives à comment diviser le travail sur plusieurs CPU et/ou GPU ont été traitées dans le chapitre 4.

Le produit matrice-vecteur sur CPU souffre principalement des accès mémoire irréguliers et indirects. Pour optimiser ce produit, on cherche un format compact qui minimise le nombre d'accès mémoire et qui utilise efficacement le cache, en réduisant les accès irréguliers au vecteur v.

5.7.1 SpMV COO, CSR et ELL

Les formats précédemment étudiés s'appliquent sur CPU. Nous nous intéressons particulièrement aux formats COO, CSR, ELL et un format non mentionné jusqu'à maintenant, CSR-bloc (BCSR). Les autres formats ne présentent pas d'intérêt sur CPU.

Dans les pseudo-codes qui suivent, on détaille le produit matrice-vecteur pour les formats COO, CSR et ELL-R.

Algorithm 5.7 SpMV COO

Entrées : `data` : tableau de n_{NZ} éléments de $\mathbb{K}$
`row_id` : tableau de n_{NZ} entiers positifs
`col_id` : tableau de n_{NZ} entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

$v \leftarrow \{0\}$
Pour $j \leftarrow 0$ **à** $n_{\text{NZ}} - 1$ **faire**
  $v_{\texttt{row\_id}_j} \leftarrow \texttt{addmul}(v_{\texttt{row\_id}_j}, \texttt{data}_j, u_{\texttt{col\_id}_j})$

Algorithm 5.8 SpMV CSR

Entrées : `data` : tableau de n_{NZ} éléments de $\mathbb{K}$
`id` : tableau de n_{NZ} entiers positifs
`ptr` : tableau de $N+1$ entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

Pour $i \leftarrow 0$ **à** $N - 1$ **faire**
  `sum` $\leftarrow 0$
  Pour $j \leftarrow \texttt{ptr}_i$ **à** $\texttt{ptr}_{i+1} - 1$ **faire**
    `sum` $\leftarrow \texttt{addmul}(\texttt{sum}, \texttt{data}_j, u_{\texttt{id}_j})$
  $v_i \leftarrow$ `sum`

Algorithm 5.9 SpMV ELL-R

Entrées : `data` : tableau de $K \times N$ éléments de $\mathbb{K}$
`id` : tableau de $K \times N$ entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

Pour $i \leftarrow 0$ **à** $N - 1$ **faire**
  `sum` $\leftarrow 0$
  Pour $j \leftarrow 0$ **à** $\texttt{len}_i - 1$ **faire**
    `sum` $\leftarrow \texttt{addmul}(\texttt{sum}, \texttt{data}_{N \times j + i}, u_{\texttt{id}_{N \times j + i}})$
  $v_i \leftarrow$ `sum`

5.7.2 SpMV BCSR

Le format BCSR est en quelque sorte une généralisation du format CSR. Il remplace un élément non nul par un bloc dense $r \times c$. Pour $r = c = 1$, on retrouve le format CSR. Le format BCSR divise la matrice en $\frac{N}{r}$ lignes de blocs, chaque ligne est composée d'un certain nombre de blocs denses $r \times c$. Chaque bloc est stocké dans un format de matrice dense (les éléments sont ordonnés par rapport à leur indice de ligne ou par rapport à leur indice de colonne). Les blocs sont ordonnés par leur indice de ligne et sont stockés dans un tableau `data`. Un tableau `id` indique l'indice de colonne du premier élément d'un bloc et un tableau `ptr` indique le début et la fin d'une ligne de blocs.

$$\begin{pmatrix} a_{00} & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{20} & 0 & 0 & 0 & a_{24} & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) matrice creuse A

$$\texttt{data} = \begin{bmatrix} a_{00} & a_{01} & 0 & a_{11} & 0 & a_{03} & a_{12} & a_{13} & a_{20} & 0 & 0 & a_{31} & \dots \end{bmatrix}$$

$$\texttt{id} = \begin{bmatrix} 0 & 2 & 0 & \dots \end{bmatrix}$$

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & \dots \end{bmatrix}$$

(b) sa représentation BCSR-2x2

Le but de cette structure de données est de pouvoir diviser le produit creux matrice-vecteur en beaucoup de petits produits denses matrice-vecteur, ce qui permet de réutiliser les éléments des vecteurs u et v. Le format BCSR stocke les indices de colonne d'un bloc, alors que le format CSR stocke les indices de colonne d'un coefficient. Si les blocs obtenus ne contiennent pas beaucoup de zéros (c'est-à-dire $K_{rc} < n_{\text{NZ}}$), le BCSR réduit les accès de mémoire, vu que le tableau *id* est plus court qu'avec le format CSR.

Algorithm 5.10 SpMV BCSR-2x2

Entrées : `data` : tableau de $K_{22} \times 4$ éléments de $\mathbb{K}$
`id` : tableau de K_{22} entiers positifs
`ptr` : tableau de $\frac{N}{2}$ entiers positifs
u : vecteur de N éléments de $\mathbb{K}$
Sortie : v : vecteur de N éléments de $\mathbb{K}$

Pour $i \leftarrow 0$ **à** $\frac{N}{2} - 1$ **faire**
  $\texttt{sum}_0 \leftarrow 0$
  $\texttt{sum}_1 \leftarrow 0$ **Pour** $j \leftarrow \texttt{ptr}_i$ **à** $\texttt{ptr}_{i+1} - 1$ **faire**
    $\texttt{src}_0 \leftarrow u_{\texttt{id}_j}$
    $\texttt{src}_1 \leftarrow u_{\texttt{id}_{j+1}}$
    $\texttt{sum}_0 \leftarrow \texttt{addmul}(\texttt{sum}_0, \texttt{data}_{4j}, \texttt{src}_0)$
    $\texttt{sum}_1 \leftarrow \texttt{addmul}(\texttt{sum}_1, \texttt{data}_{4j+2}, \texttt{src}_0)$
    $\texttt{sum}_0 \leftarrow \texttt{addmul}(\texttt{sum}_0, \texttt{data}_{4j+1}, \texttt{src}_1)$
    $\texttt{sum}_1 \leftarrow \texttt{addmul}(\texttt{sum}_1, \texttt{data}_{4j+3}, \texttt{src}_1)$
  $v_{2i} \leftarrow \texttt{sum}_0$
  $v_{2i+1} \leftarrow \texttt{sum}_1$

5.7.3 Comparaison des SpMV COO, CSR, ELL et BCSR

Nous reprenons la même matrice test (voir table 5.1) et le même nombre premier de taille 217 bits et comparons les performances des produits matrice-vecteur sur un processeur Intel Core i5-4570 (3.2 GHz). La table 5.6 montre que c'est le SpMV CSR qui est le plus rapide. Nous avons espéré que le format BCSR sera plus efficace sur les parties denses, où le taux de remplissage des blocs BCSR est élevé. Toutefois, ses performances ne dépassent pas celles du format CSR.

	Temps en ms	Débit en GOP/s
COO	634	2.46
CSR	**584**	**2.66**
ELL	811	1.92
BCSR-2×2	725	2.15

TABLE 5.6 – Comparaison des performances des SpMV COO, CSR, ELL et SLCOO sur CPU, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

5.7.4 Optimisations pour le produit CSR

Encodage Delta. Les indices de colonne sont stockés sur des entiers 16 bits, si la taille de la matrice est inférieur à 2^{16}, sur des entiers 32 bits si la taille est entre 2^{16} et 2^{32} et sur des entiers 64 bits si la matrice a une taille entre 2^{32} et 2^{64}, et ainsi de suite. Goumas et al. proposent de stocker les écarts entre les coefficients non nuls, plutôt que les indices de colonne [GKA+09]. Comme les écarts sont plus petits que les indices, dans certains cas, ils peuvent être stockés dans des entiers plus petits. Par exemple, pour une matrice de taille 2^{20} dont l'écart maximal est inférieur à 2^{14}, cet encodage permet d'utiliser des entiers 16 bits au lieu d'entiers 32 bits, et par conséquent réduit la mémoire.

Optimisation de la boucle. Le SpMV CSR contient deux boucles imbriquées. La boucle extérieure itère sur toutes les lignes, alors que la boucle intérieure itère sur les coefficients d'une ligne. On voit que la fin d'une ligne est directement suivie par le début de la ligne suivante. Williams et al. proposent d'itérer du premier coefficient de la première ligne au dernier de la dernière ligne [WOL+07]. De cette manière, on utilise un seul compteur de boucle. Sur notre processeur Intel, ceci offre une amélioration des performances d'autour de 3%.

Compresser le tableau de valeurs `data`. De même que pour le SpMV GPU, on réordonne les coefficients non nuls de la matrice en trois catégories (+1, −1 et les autres) et ainsi on peut compresser le tableau `data` et on a une arithmétique spécifique à chaque catégorie. Ceci diminue le nombre de lectures et fournit une accélération d'environ 21%.

La proportion des ±2 représente un peu moins que 5% des coefficients non nuls de la matrice. Il est par conséquent intéressant d'appliquer sur les ±2 le procédé que nous avons appliqué sur les ±1 :

- Au niveau du format de représentation de la matrice : à la place de stocker les ±2 dans le tableau `data`, on stocke leur nombre. Ainsi, le format final pour une ligne est comme suit :
  - Nombre des coefficients non nuls de la ligne
  - Nombre des 1
  - Nombre des -1
  - Nombre des 2
  - Nombre des -2
  - Liste des valeurs pour les coefficients plus grands que 2
  - Liste des valeurs pour les coefficients plus petits que -2
  - Liste des indices de colonne des coefficients non nuls dans l'ordre (1,-1,2,-2,>2,<-2).

— Au niveau de l'arithmétique : en traitant les ± 2 séparément des autres coefficients, il est intéressant d'avoir une arithmétique spécifique aux ± 2. En effet, il est possible d'avoir une routine `addmul` optimisée pour le produit par ± 2 et donc plus rapide que lorsqu'on multiplie par un nombre quelconque.

Ces deux optimisations fournissent une accélération du SpMV d'environ 5% avec la matrice-test précédente. Avec d'autres matrices, le facteur d'accélération est entre 3 et 7%.

Comme nous avons en moyenne une centaine de coefficients par ligne, le procédé de compression est intéressant pour une valeur lorsque la proportion de celle-ci est au moins 2%. Sinon, nous allons utiliser dans chaque ligne deux entiers (un pour les positifs et un pour les négatifs) pour indiquer le nombre d'occurrence, alors que cette valeur apparaît en moyenne moins que deux fois dans cette ligne. La proportion des ± 3 est 1%, c'est pourquoi il n'est pas intéressant d'appliquer le précédé précédent sur les ± 3.

Augmenter le taux de succès du cache. On divise la matrice en bloc verticaux, où chaque bloc est stocké en format CSR, sauf le dernier, le moins dense, qui est stocké en COO. Ceci diminue l'irrégularité des accès sur le vecteur source et fournit une accélération d'environ 4%.

5.8 Comparaison avec des bibliothèques existantes

La bibliothèque LinBox fournit un solveur de systèmes creux définis sur un corps fini qui tient sur plusieurs mots machine. Cependant, LinBox suppose que tous les coefficients non nuls de la matrice sont « grands » et sont donc stockés et traités en multi-précision en utilisant la couche `mpz` de GMP (GNU Multiple Precision Arithmetic Library) [GMP], ce qui rajoute un surcoût calculatoire et de mémoire par rapport à une implémentation qui exploite le fait que la grande majorité de ces coefficients tiennent sur un mot machine et qu'une importante proportion d'entre eux valent en plus des ± 1. Dans [GV14], Giorgi et Vialla reportent dans la figure 3 un facteur d'accélération entre 12 et 32 en comparant les performances de leur implémentation par rapport à celles de LinBox en utilisant les matrices obtenues pour la résolution du logarithme discret dans les corps $\mathbb{F}_{2^{619}}$ et $\mathbb{F}_{2^{809}}$ (voir tables 7.1 et 7.4). Nous avons aussi mesuré un facteur d'accélération de cet ordre de grandeur (autour de 50) en faveur de notre implémentation.

Dans la table 5.7, nous comparons les performances de notre implémentation CPU avec celles de la bibliothèque FFLAS-FFPACK et celles du module d'algèbre linéaire dans CADO-NFS. Nous mesurons le temps du SpMV, avec la matrice issue du calcul de logarithme discret dans $\mathbb{F}_{2^{619}}$, exécuté sur une machine multi-cœurs qui contient 4 processeurs un processeur Intel Core i5-4570 (3.2 GHz). Nous avons testé 2 configurations de parallélisation du produit matrice-vecteur, en variant t. On rappelle que t correspond à la dimension de la grille carrée de parallélisation du SpMV (voir chapitre 4). Les mesures faites avec FFLAS-FFPACK correspondent à un code fourni par Vialla et qui est en cours d'intégration dans la bibliothèque FFLAS-FFPACK. Ce code repose sur la bibliothèque `Cilk++` pour le multi-threading.

	Parallélisation $(t \times t)$	Temps en ms	Débit en GOP/s
Notre implémentation	1×1	**584**	**2.66**
CADO-NFS		2910	0.53
FFLAS-FFPACK		837	1.86
Notre implémentation	2×2	**225**	**6.9**
CADO-NFS		750	2.07
FFLAS-FFPACK		293	5.3

TABLE 5.7 – Comparaison des performances du SpMV de notre implémentation avec d'autres bibliothèques pour la matrice FFS pour $\mathbb{F}_{2^{619}}$.

5.9 Conclusion

Dans ce chapitre, nous avons exploré les structures de données pour effectuer le produit matrice-creuse–vecteur, principalement pour les GPU. Nous avons adapté ces produits au contexte des corps finis de grande caractéristique et ajouté des optimisations qui tiennent compte du caractère creux de la matrice et des spécificités du modèle de programmation. Il apparaît que le produit CSR-V est le plus efficace et que le produit SLCOO pose des difficultés matérielles qui annulent sa contribution à améliorer le taux de succès du cache.

Chapitre 6

Arithmétique sur les corps finis

Dans ce chapitre, nous nous intéressons aux aspects arithmétiques de l'algèbre linéaire. Nous rappelons que les calculs s'effectuent sur des corps finis de grande caractéristique. Nous cherchons un système de représentation et des algorithmes qui nous permettent d'implémenter une arithmétique efficace sur des architectures qui effectuent des calculs en parallèle telles que les cartes graphiques ou les processeurs avec extensions SIMD.
En particulier, nous allons étudier l'utilisation de l'arithmétique RNS (Residue Number System), mais nous serons aussi amenés à nous intéresser à d'autres formes de représentation des grands nombres telles que la représentation multiprécision (MP) et la représentation MRS (Mixed Radix System), ne serait-ce que pour évaluer l'apport de RNS dans notre contexte.
Les résultats de ce chapitre ont été présentés dans [Jel14a].

Sommaire

6.1 Système modulaire de représentation (RNS)

Le système modulaire de représentation (RNS) est basé sur le théorème des restes chinois (Chinese Remainder Theorem, CRT).

6.1.1 Théorème des restes chinois

Prenons une liste de n entiers premiers entre eux deux à deux, qu'on note $\mathcal{B} = (m_1, m_2, \ldots, m_n)$. Leur produit est noté $M = \prod_{i=1}^{n} m_i$. Le théorème des restes chinois résout le système de congruences suivant :

$$\begin{cases} x & \equiv & x_1 & (\text{mod } m_1) \\ x & \equiv & x_2 & (\text{mod } m_2) \\ \vdots & & & \\ x & \equiv & x_n & (\text{mod } m_n). \end{cases}$$

Théorème 1. *Le système de congruences précédent admet une unique solution modulo M.*

La preuve permet de construire une solution du système :

Preuve. Pour chaque $i \in [1, n]$, on définit $M_i = \dfrac{M}{m_i}$. Les entiers M_i et m_i étant premiers entre eux, l'égalité de Bézout indique qu'il existe un couple d'entiers (u_i, v_i) tel que $m_i u_i + M_i v_i = 1$.

Ainsi, $M_i v_i \equiv 1 \pmod{m_i}$, $\forall i \in [1, n]$ et $M_j v_j \equiv 0 \pmod{m_i}$, $\forall j \neq i$.
On a alors :

$$v_1 M_1 x_1 + v_2 M_2 x_2 + \ldots + v_n M_n x_n \equiv x_i \pmod{m_i},\ \forall i \in [1, n] \tag{6.1}$$

Le nombre $x = v_1 M_1 x_1 + v_2 M_2 x_2 + \ldots + v_n M_n x_n$ est donc solution du système.
Prouvons maintenant l'unicité de la solution modulo M. Supposons qu'il existe une deuxième solution y du système de congruences. On voit que $x - y$ est divisible par chaque m_i. Tous les m_i sont premiers entre eux. Ainsi, on déduit par le théorème de Gauss que $x - y$ est divisible par le produit des m_i, donc par M. Par conséquent, les deux nombres sont congrus modulo M. □

En remplaçant les coefficients de Bézout par leurs valeurs, la somme s'écrit

$$\sum_{i=1}^{n} x_i M_i v_i = \sum_{i=1}^{n} x_i M_i \left| {M_i}^{-1} \right|_{m_i}. \tag{6.2}$$

Les notations $|.|_{m_i}$ et $\left|.^{-1}\right|_{m_i}$ signifient respectivement le reste de la division euclidienne par m_i et l'inverse multiplicatif modulo m_i.

La majoration suivante de la somme

$$\sum_{i=1}^{n} x_i M_i \left| {M_i}^{-1} \right|_{m_i} < \sum_{i=1}^{n} m_i M \tag{6.3}$$

nous conduit à effectuer une réduction modulo M afin de nous assurer que le résultat ne déborde pas de l'intervalle $[0, M[$.

L'expression complète de la solution du système est alors :

$$x = \left| \sum_{i=1}^{n} x_i M_i \left| {M_i}^{-1} \right|_{m_i} \right|_M. \tag{6.4}$$

6.1.2 Le système RNS

Le système modulaire de représentation utilise la formulation algébrique suivante du théorème des restes chinois.

Théorème 2. *L'application*

$$\begin{array}{rccc} \varphi : & \mathbb{Z}/M\mathbb{Z} & \to & \mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_n\mathbb{Z} \\ & x & \mapsto & (|x|_{m_1}, \ldots, |x|_{m_n}) \end{array}$$

est un isomorphisme d'anneaux.

Un résultat pratique de ce théorème est que tout entier de $[0, M-1]$ peut être représenté d'une manière unique par le n-uplet $(x_1, x_2, \ldots, x_n)$, où $x_i = |x|_{m_i}$.

Vocabulaire et notations. La liste $\mathcal{B}$ est dite *base RNS*. On appelle un élément m_i de la liste *module RNS*, alors que le reste x_i de la division euclidienne par rapport à un module est appelé *résidu* ou *composante*. Le n-uplet $(x_1, x_2, \ldots, x_n)$, qu'on notera aussi $\vec{x}$, est la *représentation RNS* ou *vecteur RNS* de l'entier x dans la base $\mathcal{B}$.

6.1.3 Intérêt pour le système RNS

Prenons deux entiers x, y et leurs représentations RNS dans la base $\mathcal{B}$ notées $\vec{x}, \vec{y}$. On suppose que $x, y < M$. L'addition en RNS, notée $\vec{+}$, est effectuée en réalisant une addition modulaire (addition dans $\mathbb{Z}/m_i\mathbb{Z}$) dans chaque composante RNS :

$$\vec{x}\vec{+}\vec{y} = (|x_1 + y_1|_{m_1}, \ldots, |x_n + y_n|_{m_n}) \tag{6.5}$$

De même, la multiplication RNS, notée $\vec{\times}$, est effectuée à travers n multiplications modulaires :

$$\vec{x}\vec{\times}\vec{y} = (|x_1 \times y_1|_{m_1}, \ldots, |x_n \times y_n|_{m_n}) \tag{6.6}$$

Le résultat des opérations arithmétiques est valide si le résultat attendu est dans $[0, M-1]$. Sinon, il sera réduit modulo M. En effet, l'entier correspondant à la représentation RNS $\vec{x}\vec{+}\vec{y}$ est égal à la somme $x + y$ si et seulement si $0 \leq x + y < M$. De même, l'entier correspondant à la représentation RNS $\vec{x}\vec{\times}\vec{y}$ est égal au produit $x \times y$ si et seulement si $0 \leq x \times y < M$.

Ce système de représentation est particulièrement intéressant pour l'arithmétique sur des grands entiers, vu qu'il permet de distribuer le calcul sur plusieurs petits résidus. Dans le cas où nous disposons de plusieurs unités calculatoires, les opérations sur les composantes RNS peuvent être parallélisées. Le système RNS ne propage pas de retenue : ainsi, les unités calculatoires qui vont traiter les composantes RNS n'ont pas besoin de communiquer et peuvent être asynchrones [Tay84].

Une autre propriété intéressante de la représentation RNS est qu'il s'agit d'une représentation non-positionnelle, dans le sens où les composantes RNS ont toutes une place équivalente. Dans le cas d'une exécution séquentielle, on peut changer l'ordre de calcul des composantes. Dans le cas d'une exécution parallèle, on peut changer l'association entre les unités calculatoires et les composantes RNS. L'introduction d'aléa dans le traitement des composantes RNS permet de protéger contre certaines attaques physiques de type attaques par canaux cachés.

Nous avons vu que les opérations telles que l'addition, la soustraction et la multiplication sont intéressantes en utilisant le système RNS. Cependant, son caractère non-positionnel rend

difficile l'information sur l'ordre de grandeur d'un nombre représenté en RNS. Ainsi, il n'y a pas de moyen simple pour détecter le dépassement de capacité, et les implémentations en RNS d'opérations telles que la division ou la comparaison sont plutôt subtiles. Pour la comparaison RNS, plus de détails peuvent être consultés dans [Sou07]. Pour la division, nous nous intéresserons dans ce recueil à comment effectuer cette opération en RNS, plus particulièrement au calcul du reste d'une division. Parmi les premiers articles qui se sont intéressés à cette question, on peut mentionner les travaux de [Gam91, LC92, HK95, BDM98].

6.2 RNS pour l'arithmétique sur les corps finis

Nous avons déjà vu dans les chapitres précédents que l'algèbre linéaire est formée de SpMV (Sparse-Matrix–Vector product) de type $v \leftarrow Au$ sur un corps fini de grande caractéristique $\mathbb{Z}/\ell\mathbb{Z}$, où A est la matrice creuse et u et v sont deux vecteurs denses.

Nous rappelons que les éléments des vecteurs u et v sont « grands » (c'est-à-dire que leur représentant dans $[0, \ell[$ occupe autant de mots machine que ℓ). Pour les coefficients de la matrice A, nous distinguons deux cas (voir la sous-section 3.2.1) :

- les matrices FFS : leurs coefficients sont « petits », dans le sens où ils sont bornés par une valeur maximale B inférieur à 2^{10} (B peut être encore beaucoup plus petite que cette borne, nous avons dans la sous-section 3.2.1 que B était égal à 36). Nous notons que la grande majorité (autour de 90%) de ces coefficients sont des ± 1 ;
- les matrices NFS : en plus des coefficients « petits », la matrice contient des coefficients « grands ».

L'objectif est de représenter le corps $\mathbb{Z}/\ell\mathbb{Z}$ en RNS et d'effectuer les opérations arithmétiques du SpMV en RNS. On commence par convertir le vecteur u de sa représentation entière à sa représentation RNS. Ensuite, on effectue en RNS le produit $v \leftarrow Au$ et on réduit le vecteur v modulo ℓ. Enfin, on passe de la représentation RNS du vecteur v à sa représentation entière. Dans le cas de plusieurs produits itérés $v \leftarrow A^i u$ (ce qui est le cas de l'algorithme de résolution de Wiedemann, voir sous-section 3.5.2), on préfère n'effectuer qu'une seule conversion de l'entier en RNS (pour u) et une seule conversion de RNS en entier (pour v). En d'autres termes, il est préférable de rester en RNS pendant le calcul de tous les produits itérés, vu que les opérations de conversion d'une représentation à une autre sont coûteuses. De plus, au lieu d'avoir une réduction modulo ℓ après chaque produit matrice-vecteur, on envisage d'accumuler un certain nombre de produits matrice-vecteur avant de faire une réduction modulo ℓ. Le nombre de produits accumulés dépend de la taille de la base RNS.

Si on suppose que x et y sont des éléments des vecteurs v et u respectivement, nous avons besoin des opérations suivantes :

1. calculer $\vec{x}$ à partir de x ;
2. calculer x à partir de $\vec{x}$.

Ces deux opérations ne sont pas critiques vu que, dans le contexte de produits itérés, chaque opération n'est effectuée qu'une seule fois.

Nous avons aussi besoin d'effectuer les opérations arithmétiques suivantes en restant en RNS, c'est-à-dire en ne manipulant que des représentations RNS :

3. $x \leftarrow x \pm y$;
4. $x \leftarrow x + \lambda \times y$, où λ est un coefficient « petit » de A ;

5. $x \leftarrow x + \Lambda \times y$, où Λ est un coefficient « grand » de A ;
6. $x \leftarrow x \bmod \ell$.

Aux opérations précédemment mentionnées, on ajoute l'opération suivante dont on verra l'utilité plus loin dans la sous-section 6.4.2 :

7. convertir la représentation RNS de x d'une base RNS à une autre.

On se munit d'une base RNS $\mathcal{B} = (m_1, m_2, \ldots, m_n)$. On suppose que les m_i ont la même taille en bits notée k (c'est-à-dire $m_i < 2^k$).

6.2.1 Convertir un entier en RNS

Prenons un entier représentable dans la base $\mathcal{B}$, c'est-à-dire dans l'intervalle $[0, M-1]$. L'algorithme 6.1 détaille comment calculer sa représentation RNS.

Algorithm 6.1 Conversion entier en RNS

Entrées : x : un entier dans $[0, M-1]$.
Sortie : $\vec{x} = (x_1, x_2, \ldots, x_n)$: représentation RNS de x

Pour $i \in [1, n]$ **faire**
| $x_i \leftarrow |x|_{m_i}$

Trouver la représentation RNS requiert n réductions modulo m_i. La taille de x est majorée par nk bits. Chaque réduction consiste en une division dont le quotient est de taille $(n-1)k$. Si on utilise l'algorithme naïf de division, cette opération est effectuée en

$$\mathcal{O}((n-1)k \cdot k) = \mathcal{O}((n-1)k^2).$$

Ainsi, le coût total est

$$\text{coût}(\text{conversion entier} \to \text{RNS}) = \mathcal{O}(n(n-1)k^2). \tag{6.7}$$

Dans la pratique, il existe des choix de modules pour lesquels le calcul de la réduction modulo m_i est simplifié et coûte moins qu'une division. Par exemple, si m_i est un nombre pseudo-Mersenne. Ces considérations seront discutées dans la sous-section 6.3.1.

6.2.2 Convertir une représentation RNS en entier

Pour calculer le sens inverse, c'est-à-dire celui de trouver l'entier correspondant à une représentation RNS, nous avons deux méthodes possibles : soit en utilisant le théorème des restes chinois, soit en passant par un autre système de représentation.

Via le théorème des restes chinois. L'algorithme 6.2 récapitule les étapes de ce calcul.

Algorithm 6.2 Conversion représentation RNS en entier (Thé. des restes chinois)

Pré-calcul : M_i et $|{M_i}^{-1}|_{m_i}, \forall i \in [1, n]$.
Entrées : $\vec{x} = (x_1, x_2, \ldots, x_n)$: représentation RNS d'un entier x.
Sortie : x : l'entier dans $[0, M-1]$ qui correspond à la représentation

$$x \leftarrow \sum_{i=1}^{n} x_i \times M_i \times |{M_i}^{-1}|_{m_i}$$
$x \leftarrow x \bmod M$

En supposant que M_i et $|M_i^{-1}|_{m_i}$ sont pré-calculés, le calcul nécessite les étapes suivantes :

1. calculer les termes additifs γ_i ;
2. sommer ces termes ;
3. réduire la somme modulo M.

Les tailles en bits des entiers M_i et M sont respectivement $(n-1)k$ et nk. Multiplier x_i par M_i donne un entier inférieur à M, donc de taille au plus nk. Le coût de cette multiplication est

$$\mathcal{O}(k \cdot (n-1)k).$$

Multiplier le résultat précédent par $|M_i^{-1}|_{m_i}$ donne un entier dont la taille est majorée par $(n+1)k$. Ce produit s'effectue en

$$\mathcal{O}(nk \cdot k).$$

Ainsi, calculer les n termes additifs prend $\mathcal{O}(n(2n-1)k^2)$.
Sommer n termes de longueur $(n+1)k$ coûte :

$$\mathcal{O}((n-1) \cdot (n+1)k).$$

La réduction modulo M consiste à diviser un entier de taille $(n+1)k+n$ par un entier de taille nk. Elle s'effectue en

$$\mathcal{O}(k \cdot nk).$$

Au total, le coût de la conversion est

$$\text{coût(conversion RNS} \to \text{entier)} = \mathcal{O}(2n^2k^2). \tag{6.8}$$

Via MRS. Le système MRS, aussi connu sous le nom de bases de Cantor, est un système de représentation de nombres à numération de position [Can69].

Étant donnés la base RNS $\mathcal{B} = (m_1, \ldots, m_n)$ et un entier x, la représentation MRS de x est la liste $(r_1, r_2, \ldots, r_n)$ telle que

$$x = r_1 + r_2m_1 + r_3m_2m_1 + \cdots + r_nm_{n-1}m_{n-2}\ldots m_1, \text{ où } 0 \leq r_i < m_i. \tag{6.9}$$

La représentation MRS est une représentation de position de poids 1, m_1, m_2m_1 jusqu'à $m_{n-1}m_{n-2}\ldots m_1$. Cette représentation est souvent complémentaire de la représentation RNS. Par exemple, pour comparer deux nombres représentés en RNS, il est envisageable de calculer leurs représentations MRS qui peuvent être facilement comparées. Ici, on peut utiliser le système MRS pour passer de la représentation RNS à la représentation MRS, puis à l'entier correspondant.

On procède en convertissant la représentation RNS $(x_1, \ldots, x_n)$ en représentation MRS $(r_1, \ldots, r_n)$ dans la même base $\mathcal{B}$:

$$\begin{cases} r_1 &= |x_1|_{m_1}, & \\ r_2 &= |(x_2 - r_1)C_2|_{m_2}, & C_2 = |m_1^{-1}|_{m_2} \\ r_3 &= |((x_3 - r_1) - m_1r_2)C_3|_{m_3}, & C_3 = |m_1^{-1}m_2^{-1}|_{m_3} \\ \vdots & & \\ r_n &= |((x_n - r_1) - m_1(r_2 - m_2(r_3 \ldots)))C_n|_{m_n} & C_n = |m_1^{-1}\ldots m_{n-1}^{-1}|_{m_n}. \end{cases}$$

L'algorithme 6.3 récapitule les étapes de conversion d'une représentation RNS en entier via le système MRS. Cet algorithme est aussi connu sous le nom d'algorithme de Garner [Gar59].

Algorithm 6.3 Conversion représentation RNS en entier (Garner)

Pré-calcul : $C_i = \left| \prod_{j=1}^{i-1} m_j^{-1} \right|_{m_i}$ et $D_i = \prod_{j=1}^{i-1} m_j, \forall i \in [2, n]$.
Entrées : $\vec{x} = (x_1, x_2, \ldots, x_n)$: représentation RNS d'un entier x.
Sortie : x : l'entier dans $[0, M-1]$ qui correspond à la représentation

$t \leftarrow x_1, x \leftarrow x_1$
Pour $i \in [2, n]$ **faire**
  $t \leftarrow |(x_i - x)C_i|_{m_i}$
  $x \leftarrow x + tD_i$

Comparé à l'algorithme 6.2, cet algorithme ne requiert pas de réduction modulo M. Si on néglige les coûts des pré-calculs C_i et D_i, la complexité de l'algorithme 6.3 est de $\mathcal{O}(nk^2)$. Cette méthode de conversion est séquentielle. Bajard propose dans [BP04] une version parallèle de la conversion de RNS à MRS.

À présent, nous explorons comment implémenter les opérations arithmétiques en RNS. On suppose, par souci de simplicité des algorithmes, que la matrice contient uniquement des coefficients positifs. Les algorithmes pour les coefficients négatifs peuvent être déduits facilement.

6.2.3 Addition en RNS

Nous avons déjà vu que l'addition est effectuée par une addition modulaire dans chaque composante. Les deux opérandes d'entrée étant des éléments de $\mathbb{Z}/\ell\mathbb{Z}$, le résultat final est valide si et seulement si $2 \times (\ell - 1) < M$.

Algorithm 6.4 Addition RNS

Entrées : $\vec{x}$ et $\vec{y}$: deux représentations RNS de x et y, éléments de $\mathbb{Z}/\ell\mathbb{Z}$.
Sortie : $\vec{z}$: représentation RNS de $z = x + y$

Pour chaque *composante* i **faire**
  $z_i \leftarrow |x_i + y_i|_{m_i}$

6.2.4 AddMul en RNS

L'opération AddMul se réfère à une multiplication par un élément non nul λ (appartenant à $[1, B-1]$) de la matrice A, suivie d'une addition. En pratique, on est concerné par le cas $B = 2^{10}$. D'une manière similaire à l'addition, l'opération est effectuée dans chaque composante RNS. L'algorithme requiert que $B \times (\ell - 1) < M$.

Algorithm 6.5 AddMul RNS

Entrées : $\vec{x}$ et $\vec{y}$: deux représentations RNS de x et y, éléments de $\mathbb{Z}/\ell\mathbb{Z}$.
  λ : un entier dans $[1, B-1]$.
Sortie : $\vec{z}$: représentation RNS de $z = x + \lambda \times y$

Pour chaque *composante* i **faire**
  $z_i \leftarrow |x_i + \lambda \times y_i|_{m_i}$

6.2.5 Multiplication en RNS

Tout comme les deux cas précédents, l'opération est effectuée sur les n composantes RNS. L'algorithme donne un résultat valide si et seulement si $(\ell-1)^2 < M$. Cette condition est bien entendu contraignante vu qu'il en suit que pour pouvoir multiplier deux éléments de $\mathbb{Z}/\ell\mathbb{Z}$ en RNS, on a besoin d'une base RNS deux fois plus grande que la base RNS qui permet de représenter un élément de $\mathbb{Z}/\ell\mathbb{Z}$.

Algorithm 6.6 Multiplication RNS

Entrées : $\vec{x}$ et $\vec{y}$: deux représentations RNS de x et y, éléments de $\mathbb{Z}/\ell\mathbb{Z}$.
Sortie : $\vec{z}$: représentation RNS de $z = x \times y$

Pour chaque *composante* i **faire**
  $z_i \leftarrow |x_i \times y_i|_{m_i}$

Dans une architecture multi-threadée (ou pourvue de plusieurs unités vectorielles) où on dispose de plus que n threads (ou n unités vectorielles), on peut associer pour les trois opérations mentionnées un *thread* (ou une unité vectorielle) à chaque composante RNS. Ainsi, les opérations élémentaires (opérations dans les composantes RNS) peuvent être effectuées en parallèle par les n *threads* (ou n unités vectorielles) et la complexité parallèle est égale à la complexité d'une opération élémentaire.

6.2.6 Réduction modulo ℓ en RNS

Le but est de réduire modulo ℓ une représentation RNS d'un entier x dans $[0, M-1]$ en restant dans le domaine RNS.

Dans la littérature, cette question a été plus généralement étudiée dans les travaux qui cherchent à optimiser la multiplication modulaire avec RNS, qui est une opération centrale dans plusieurs cryptosystèmes. Plusieurs travaux se sont intéressés à la multiplication de Montgomery, parmi lesquels on peut citer les travaux de Bajard et al. [BDK98] et de Kim et al. [KPH04].

La méthode que nous détaillons ci-dessous et qui a été proposée par Shenoy et Kumaresan, permet d'effectuer la réduction modulo ℓ [SK89].

Nous commençons à partir de la construction du théorème des restes chinois (voir sous-section 6.1.1) :

$$x = \left| \sum_{i=1}^{n} \gamma_i M_i \right|_M, \text{où nous définissons } \gamma_i = \left| x_i M_i^{-1} \right|_{m_i}. \tag{6.10}$$

Maintenant, si on définit l'entier α comme suit :

$$\alpha = \left\lfloor \sum_{i=1}^{n} \frac{\gamma_i M_i}{M} \right\rfloor = \left\lfloor \sum_{i=1}^{n} \frac{\gamma_i}{m_i} \right\rfloor, \tag{6.11}$$

alors l'entier x peut être écrit comme $\sum_{i=1}^{n} \gamma_i M_i - \alpha M$ et, comme $\gamma_i < m_i$, nous avons $0 \leq \alpha < n$.

Maintenant, si on suppose que α est connu, on définit $z = \sum_{i=1}^{n} \gamma_i \left|M_i\right|_\ell - \left|\alpha M\right|_\ell$. On peut

facilement vérifier que z est congru à x modulo ℓ et qu'il est dans l'intervalle $\left[0, \ell \sum_{i=1}^{n} m_i\right[$, qu'on pourrait approcher par l'intervalle $[0, n2^k\ell[$.

Il reste à déterminer l'entier α. Soit on procède à un calcul de la valeur exacte de α à partir de l'équation 6.11, soit on passe par un estimé dont le calcul est plus rapide que le calcul de α.

Shenoy et Kumaresan proposent dans [SK89] d'utiliser un module supplémentaire. Posh et Posh proposent une approche avec des nombres en virgule flottante [PP92]. Bernstein présente dans [Ber95] un calcul d'estimé de α qui est basé sur l'hypothèse que les m_i sont proches de 2^k. Dans la suite, nous allons détailler et utiliser l'approche de Bernstein.

Si $m_i \approx 2^k$, on peut approcher le quotient γ_i/m_i par le quotient $\gamma_i/2^k$, qui est plus rapide à calculer. Il est aussi possible de pousser encore plus l'approximation en choisissant un paramètre entier s dans $[1, k]$ et d'approcher γ_i/m_i par les s bits de poids fort de $\gamma_i/2^k$. Ainsi, un estimé de α est donné par :

$$\hat{\alpha} = \left\lfloor \sum_{i=1}^{n} \frac{\left\lfloor \frac{\gamma_i}{2^{k-s}} \right\rfloor}{2^s} + \Delta \right\rfloor, \tag{6.12}$$

où Δ est un terme correctif dans $]0, 1[$, qui compense les erreurs induites par les approximations.

Si $0 \leq x < (1-\Delta)M$ et $(\varepsilon + \delta) \leq \Delta < 1$ où $\varepsilon = \sum_{i=1}^{n} \frac{2^k - m_i}{2^k}$ et $\delta = n\frac{2^{k-s}-1}{2^k}$, alors $\alpha = \hat{\alpha}$. Les quantités ε et δ sont reliées aux erreurs d'approximation.

Une fois α déterminé, nous sommes capables d'effectuer un calcul complet de z en RNS. L'algorithme 6.7 récapitule les étapes du calcul.

Algorithm 6.7 Réduction approximative modulo ℓ en RNS

Pré-calcul : Vecteur $(\left|{M_i}^{-1}\right|_{m_i})$ pour $i \in \{1, \ldots, n\}$
Table de représentations RNS $\overrightarrow{|M_j|_\ell}$ pour $j \in \{1, \ldots, n\}$
Table de représentations RNS $\overrightarrow{|\alpha M|_\ell}$ pour $\alpha \in \{1, \ldots, n-1\}$
Entrée : $\vec{x}$: représentation RNS de x, avec $0 \leq x < (1-\Delta)M$
Sortie : $\vec{z}$: représentation RNS de $z \equiv x \pmod{\ell}$, avec $z < \ell \sum_{i=1}^{n} m_i$

Pour chaque *composante* i **faire**
  $\gamma_i \leftarrow \left|x_i \times \left|{M_i}^{-1}\right|_{m_i}\right|_{m_i}$ // 1 multiplication RNS

calcul de $\alpha \leftarrow \left\lfloor \sum_{j=1}^{n} \frac{\left\lfloor \frac{\gamma_j}{2^{k-s}} \right\rfloor}{2^s} + \Delta \right\rfloor$ // addition de n termes de s-bit

Pour chaque *composante* i **faire**
  $z_i \leftarrow \left| \sum_{j=1}^{n} \gamma_j \times \left||M_j|_\ell\right|_{m_i} \right|_{m_i}$ // $(n-1)$ additions RNS & n multiplications RNS
  $z_i \leftarrow \left|z_i - \left||\alpha M|_\ell\right|_{m_i}\right|_{m_i}$ // 1 soustraction RNS

Dans cet algorithme, nous avons un grand degré de parallélisme. Les opérations peuvent être évaluées en séquentiel, mais aussi en parallèle sur les n composantes. La première multiplication est indépendante pour chaque composante. Toutefois, la connaissance de tous les γ_j est nécessaire pour le calcul de α. Par la suite, chaque composante requiert tous les γ_j, ainsi que le α pour le calcul de son z_i.

En l'occurrence, si on implémente cet algorithme sur GPU, il est intéressant d'associer une composante RNS à un thread. Après le calcul de γ_i, les différents *threads* doivent se synchroniser et communiquer tous leurs γ_i respectifs. Si tous les *threads* appartiennent à un même *warp* (ce qui est possible si le nombre des composantes RNS est inférieur à 32), il n'y a pas de surcoût lié à cette synchronisation et la diffusion des données se fait en utilisant la mémoire partagée (voir sous-section 2.3.6). Le calcul de α peut être soit effectué par un *thread* et transmis aux autres, ou bien effectué par chaque *thread*. Les deux possibilités sont identiques en terme de coût.

Si on néglige le calcul de α et les additions devant les multiplications, la complexité parallèle de cet algorithme est de $n+1$ multiplications RNS. Sa complexité séquentielle est $n(n+1)$ multiplications RNS. La complexité de cet algorithme est certes importante comparé aux algorithmes de réduction modulaire. Cependant, il est envisageable de diminuer la fréquence à laquelle une réduction modulo ℓ est nécessaire pour des SpMV itérés, afin d'amortir son coût. Ceci est typiquement possible en ajoutant un module de plus dans la base RNS.

Idéalement, nous cherchions un algorithme qui prend une entrée x dans $[0, M-1]$ et retourne une sortie z dans $[0, \ell-1]$. Cependant, l'algorithme présenté ci-dessus prend une entrée x dans $[0, (1-\Delta)M[$ et ne permet pas de trouver la réduction exacte modulo ℓ, mais plutôt calcule un résultat z dans $[0, n2^k\ell[$ congru à x modulo ℓ. Concernant l'intervalle sur l'entrée, en fixant les paramètres, on peut diminuer significativement Δ de sorte à ce que $(1-\Delta)M$ soit proche de $M-1$. Concernant la sortie de l'algorithme, notre besoin pour les calculs de SpMV n'est pas d'avoir une réduction exacte, mais plutôt d'avoir des résultats qui ne dépassent pas la limite du plus grand entier représentable dans la base. Ainsi, si on fixe $n2^k\ell < (1-\Delta)M$, on peut se contenter de cet algorithme de réduction approximative pour garantir que tous les résultats intermédiaires restent représentables dans la base RNS.

6.2.7 Conversion de RNS à RNS

Considérons deux bases RNS $\mathcal{B} = (m_1, \ldots, m_n)$ et $\tilde{\mathcal{B}} = (\tilde{m_1}, \ldots, \tilde{m_{\tilde{n}}})$ telles que $M = \prod_{i=1}^{n} m_i$ et $\tilde{M} = \prod_{i=1}^{\tilde{n}} \tilde{m_i}$ sont premiers entre eux. Nous prenons un entier x dont on connaît la représentation RNS $(x_1, \ldots, x_n)$ dans la base $\mathcal{B}$. Nous cherchons sa représentation RNS $(\tilde{x_1}, \ldots, \tilde{x}_{\tilde{n}})$ dans l'autre base $\tilde{\mathcal{B}}$.

Il existe deux approches possibles pour obtenir la solution :
— utiliser le théorème des restes chinois, ou
— utiliser le système de base mixte (MRS pour Mixed Radix System), méthode de Shenoy et Kumaresan [SK89].

Via le théorème des restes chinois. L'approche est similaire à celle utilisée pour la réduction modulo ℓ. Nous adaptons la méthode de Szabo et Tanaka à l'approche utilisée pour la réduction

modulo ℓ. À partir du théorème des restes chinois

$$x = \sum_{i=1}^{n} \left| x_i \left| M_i^{-1} \right|_{m_i} \right|_{m_i} M_i - \alpha M = \sum_{i=1}^{n} \gamma_i M_i - \alpha M, \tag{6.13}$$

il suffit de réduire l'équation précédente modulo chaque module $\tilde{m}_j$ de la nouvelle base

$$\tilde{x}_j = |x|_{\tilde{m}_j} = \left| \sum_{i=1}^{n} \gamma_i \left| M_i \right|_{\tilde{m}_j} - |\alpha M|_{\tilde{m}_j} \right|_{\tilde{m}_j}, \text{ pour } j \in \{1, \ldots, \tilde{n}\} \tag{6.14}$$

Pour calculer la représentation dans la nouvelle base, on groupe les calculs de tous les nouveaux modules dans l'algorithme suivant.

Algorithm 6.8 Conversion RNS de $\mathcal{B}$ à $\tilde{\mathcal{B}}$

Pré-calcul : Vecteur $(\left| M_i^{-1} \right|_{m_i})$ pour $i \in \{1, \ldots, n\}$
Table de représentations RNS de M_i pour $i \in \{1, \ldots, n\}$ dans $\tilde{\mathcal{B}}$
Table de représentations RNS de αM pour $\alpha \in \{1, \ldots, n-1\}$ dans $\tilde{\mathcal{B}}$
Entrée : Représentation RNS de x dans $\mathcal{B}$, avec $0 \leq x < (1 - \Delta)M$
Sortie : Représentation RNS de x dans $\tilde{\mathcal{B}}$

Pour chaque *composante* i **faire**
  $\gamma_i \leftarrow \left| x_i \times \left| M_i^{-1} \right|_{m_i} \right|_{m_i}$ // 1 multiplication RNS

calcul de $\alpha \leftarrow \left\lfloor \sum_{j=1}^{n} \frac{\left\lfloor \frac{\gamma_j}{2^{k-s}} \right\rfloor}{2^s} + \Delta \right\rfloor$ // addition de n termes s-bit

Pour chaque *composante* j **faire**
  $\tilde{x}_j \leftarrow \left| \sum_{i=1}^{n} \gamma_i \times |M_i|_{\tilde{m}_j} \right|_{\tilde{m}_j}$ // $(n-1)$ additions RNS & n multiplications RNS
  $\tilde{x}_j \leftarrow \left| \tilde{x}_j - |\alpha M|_{\tilde{m}_j} \right|_{\tilde{m}_j}$ // 1 soustraction RNS

On désigne les composantes de la base $\mathcal{B}$ par l'indice i et les composantes de la base $\tilde{\mathcal{B}}$ par l'indice j. Pour implémenter efficacement cet algorithme en parallèle sur une architecture *multi-threadée*, il est intéressant que les deux bases aient le même nombre de modules, ainsi les mêmes *threads* travaillant sur la première partie de l'algorithme calculent la deuxième partie. Si on néglige le calcul de α et les additions devant les multiplications, la complexité séquentielle de cet algorithme est $(n + n\tilde{n})$ multiplications RNS. Avec $\max(n, \tilde{n})$ threads, sa complexité parallèle est $n + 1$ multiplications RNS.

Via MRS. Il est possible d'utiliser le système MRS pour passer d'une base RNS à une autre. On procède en convertissant la représentation RNS $(x_1, \ldots, x_n)$ en représentation MRS $(r_1, \ldots, r_n)$ dans la même base $\mathcal{B}$, puis on calcule les résidus de x dans la nouvelle base $\tilde{\mathcal{B}}$. Nous avons déjà vu dans le paragraphe 6.2.2 comment effectuer la conversion de RNS à MRS. La conversion de MRS vers la nouvelle base RNS se fait suivant un schéma de Horner. Ces algorithmes sont détaillés dans [BP04].

6.3 Implémenter RNS sur GPU et CPU

On choisit les modules RNS d'une certaine forme dans la perspective d'accélérer l'opération de réduction modulaire effectuée après chaque opération arithmétique. Certains choix ont été par le passé plus utilisés que d'autres.

6.3.1 Choix des modules RNS

Au début des travaux sur les implémentations RNS, Merrill a proposé de prendre le premier module de la forme 2^k et les autres modules de la forme $2^{k'} - 1$, pour différentes valeurs de k' (dont k). Ce choix fait que les opérations peuvent être implémentées efficacement en utilisant les circuits classiques d'arithmétique binaire [Mer64].

Cependant, ce choix fait que l'on est rapidement limité en nombre de modules. D'autre part, les modules n'ont pas la même longueur et suggèrent d'avoir un circuit spécifique à chaque module, ce qui n'est pas judicieux sur des architectures comme les cartes graphiques ou les CPU.

Modules pseudo-Mersenne

Les nombres de Mersenne sont de la forme $2^n - 1$. Les nombres pseudo-Mersenne s'écrivent sous la forme $2^n - c$, où c est pris petit devant 2^n. L'orientation vers les nombres pseudo-Mersenne pour le choix des modules RNS se justifie par le fait qu'avec ces nombres, la réduction modulo m_i nécessaire après une opération RNS est simplifiée. Tout au long de cette section, on illustrera cet argument par des exemples.

Forme $2^{k-1} - c_i$. Une première approche est de prendre des modules de la forme pseudo-Mersenne $2^{k-1} - c_i$, où k est une taille de mots machine $(16, 32, 64)$ et c_i petit devant 2^k.

À titre d'exemple, observons ce qu'implique ce choix pour l'addition RNS de deux résidus x_i et y_i, dont le module correspondant est m_i. On cherche à calculer $z_i = (x_i + y_i) \bmod m_i$. Si m_i s'écrit sous la forme $2^{63} - c_i$, avec $c_i < 2^4$ et qu'on calcule sur des entiers 64 bits, (c'est-à-dire implicitement modulo 2^{64}), alors

$$z_i = (x_i + y_i) \bmod (2^{63} - c_i) = \begin{cases} x_i + y_i & \text{si } (x_i + y_i) < 2^{63} - c_i \\ x_i + y_i - (2^{63} - c_i) & \text{si } (x_i + y_i) \geq 2^{63} - c_i \end{cases} \tag{6.15}$$

Ainsi, l'addition RNS revient à faire une addition et une soustraction conditionnelle, qu'on pourrait remplacer par une affectation conditionnelle (`CMOV`) et une soustraction, c'est-à-dire que l'on remplacerait le test par une soustraction dont le 2^e opérande est à 0 si $(x_i + y_i) \leq 2^{63} - c_i$, ou à $(2^{63} - c_i)$ si $(x_i + y_i) > 2^{63} - c_i$.

Forme $2^k - c_i$. Un autre choix possible est de prendre des modules de la forme $2^k - c_i$, aussi avec k est une taille de mots machine et c_i un nombre petit devant 2^k. La réduction modulo m_i est alors légèrement différente. En effet, si on reprend l'exemple de l'addition RNS, avec $k = 64$, alors

$$z_i = (x_i + y_i) \bmod (2^{64} - c_i) = \begin{cases} x_i + y_i & \text{si } x_i + y_i < 2^{64} - c_i \\ x_i + y_i - (2^{64} - c_i) & \text{si } 2^{64} - c_i \leq x_i + y_i < 2^{64} \\ x_i + y_i + c_i & \text{si } x_i + y_i \geq 2^{64} \end{cases} \tag{6.16}$$

Les deux premiers cas sont équivalents au choix précédent des modules. Le troisième cas ($x_i + y_i \geq 2^{64}$) est un sous-cas du cas $x_i + y_i \geq m_i$, où il y a un dépassement de capacité. Comme on calcule implicitement modulo 2^{64}, soustraire $(2^{64} - c_i)$ est équivalent à ajouter c_i. Maintenant, pour simplifier l'équation 6.16, on peut envisager de fusionner les 2 premiers cas :

$$z_i = (x_i + y_i) \bmod (2^{64} - c_i) \approx \begin{cases} x_i + y_i & \text{si } x_i + y_i < 2^{64} \\ x_i + y_i + c_i & \text{si } x_i + y_i \geq 2^{64} \end{cases} \tag{6.17}$$

Ainsi, on remplace toutes les comparaisons par des détections de dépassement de capacité, qui sont des opérations très peu chères. L'inconvénient de cette approche est que le résultat final peut être non normalisé (c'est-à-dire pas complètement réduit modulo m_i), s'il est dans l'intervalle $[m_i, 2^{64}[$. Toutefois, la probabilité d'avoir un résultat dans cet intervalle est assez faible, d'autant plus faible que m_i est choisi proche de 2^{64}.

On peut se contenter d'opérandes non normalisés à la sortie des fonctions RNS et envisager de temps en temps des normalisations. Notons que pour l'addition RNS, l'algorithme reste correct si l'un des opérandes d'entrée est non normalisé. Ainsi, nous arrivons à construire une routine qui prend des entrées $x_i < 2^{64}$ et $y_i < m_i$ et qui génère une sortie $z_i < 2^{64}$ telle que $z_i \equiv (x_i + y_i) \pmod{m_i}$. En l'occurrence, dans le contexte du SpMV $v \leftarrow Au$, on garantit que les éléments de u sont normalisés (qui correspondent à l'opérande y_i), alors que ceux de v ne le sont pas (qui correspondent aux opérandes x_i et z_i).

Algorithmes RNS avec les modules pseudo-Mersenne

Le corps fini $\mathbb{Z}/\ell\mathbb{Z}$ est représenté par l'intervalle entier $[0, \ell - 1]$. Une représentation RNS correspond à un tableau de résidus RNS. Chaque résidu est représenté par un type de données primitif : un entier 32 ou 64 bits, un flottant simple ou double précision, etc.

On s'intéresse à une opération RNS fréquente dans le SpMV qui est $z_i \leftarrow (x_i + \lambda \times y_i) \bmod m_i$, où $0 \leq x_i, y_i, z_i < m_i$ sont des résidus RNS et λ un coefficient positif de la matrice. Ainsi, l'opération consiste en un AddMul (une multiplication par λ et une addition) suivi par une réduction modulo m_i. Supposons que le module m_i est de la forme $2^k - c_i$.

On définit $t_i = x_i + \lambda \times y_i$ comme le résultat intermédiaire avant la réduction modulaire. On peut écrire t_i comme

$$t_i = t_{iL} + 2^k \times t_{iH}, \text{où } t_{iL} = t_i \bmod 2^k, t_{iH} = \lfloor t_i / 2^k \rfloor. \tag{6.18}$$

Comme $2^k \equiv c_i \pmod{m_i}$, nous avons $t_i \equiv t_{iL} + t_{iH} \times c_i \pmod{m_i}$. Ainsi, on calcule $t_i \leftarrow t_{iL} + t_{iH} \times c_j$, alors nous avons deux cas à considérer :

- si $t_i < 2^k$, nous avons « presque » réduit $(x_i + \lambda \times y_i)$ modulo m_i ;
- sinon, nous avons réduit t_i par approximativement k bits. Il suffit de répéter la procédure précédente avec $t_i \leftarrow t_{iL} + c_i \times t_{iH}$.

RNS avec des entiers. La taille k est choisie comme un multiple de la taille des mots machine, de sorte à ce que le calcul des parties hautes et basses et les comparaisons avec 2^k soient peu chères.

L'algorithme 6.9 récapitule les étapes de calcul pour le cas $\lambda = 1$.

Algorithm 6.9 RNS Add (entiers k bits)

Entrées : $0 \leq x_i < 2^k$, $0 \leq y_i < m_i$ et $c_i < 2^{k_0}$
Sortie : $0 \leq z_i < 2^k$ tel que $z_i \equiv x_i + y_i \pmod{m_i}$
$z_i \leftarrow |x_i + y_i|_{2^k}$ // $x_i + y_i < 2^k + m_i$
Si *Dépassement* **alors**
    $z_i \leftarrow |z_i + c_i|_{2^k}$

L'algorithme 6.10 traite le cas $\lambda > 1$. L'algorithme est valide tant que $|z_{iH} \times c_i| < 2^k$. Ce qui garantit l'exactitude du résultat pour $|\lambda|$ inférieur à 2^{k-k_0}, si on majore c_i par 2^{k_0}.

Algorithm 6.10 RNS AddMul (entiers k bits)

Entrées : $0 \leq x_i < 2^k$, $0 \leq y_i < m_i$, $c_i < 2^{k_0}$ et $\lambda < 2^{k-k_0}$
Sortie : $0 \leq z_i < 2^k$ tel que $z_i \equiv x_i + \lambda \times y_i \pmod{m_i}$
$z_{iL} \leftarrow |x_i + \lambda \times y_i|_{2^k}$ // les k bits de poids faible
$z_{iH} \leftarrow (x_i + \lambda \times y_i)/2^k$ // les k bits de poids fort
$z_i \leftarrow |z_{iL} + c_i \times z_{iH}|_{2^k}$ // $c_i \times z_{iH} < 2^k$
Si *Dépassement* **alors**
    $z_i \leftarrow |z_i + c_i|_{2^k}$

RNS avec des nombres en virgule flottante. Il est possible d'utiliser un flottant pour représenter un entier. Dans ce cas, seule la mantisse sera utilisée. Ainsi, la taille des modules avec une représentation flottante est plus petite qu'avec une représentation entière (à nombre de bits équivalents) et par conséquent le nombre nécessaire de modules est plus important. On verra aussi que les algorithmes sont plus compliqués que ceux avec des entiers. On s'attend à ce qu'il y ait un surcoût lié à l'utilisation des flottants à la place des entiers. Cependant, dans certaines architectures, en l'occurrence les GPU, les débits des instructions flottantes est plus grand que celui des instructions entières. Avec certains jeux d'instructions, en l'occurrence avec AVX (voir section 2.2) seules les instructions flottantes sont disponibles, les instructions entières dont nous avons besoin pour effectuer l'arithmétique RNS ne sont supportées qu'à partir de AVX2. Ces cas justifient notre motivation pour explorer l'utilisation des flottants pour faire de l'arithmétique entière.

On prend pour la valeur de k le nombre de bits dans la mantisse moins un. Ainsi, l'opération d'addition en virgule flottante donne la somme exacte de deux résidus.

Algorithm 6.11 RNS Add (flottants à double précision)

Entrées : $m_i < 2^{52}$, $0 \leq x_i < 2^{52}$ et $0 \leq y_i < m_i$
Sortie : $0 \leq z_i < 2^{52}$ tel que $z_i \equiv x_i + y_i \pmod{m_i}$
$z_i \leftarrow x_i + y_i$
Si $z_i \geq m_i$ **alors**
    $z_i \leftarrow z_i - m_i$

Pour effectuer un AddMul en RNS avec des flottants (voir algorithme 6.12), on commence par calculer les parties basse et haute du produit $y_i \times \lambda$, qu'on note t_{iH} et t_{iL} respectivement. Le calcul peut être accéléré en utilisant un FMA (*Fused-Multiply-and-Add*). Si le FMA n'est pas supporté par l'architecture, on peut utiliser la méthode classique du Two-Product. Ensuite, on

utilise l'algorithme de Veltkamp [Dek71] pour séparer les 52 bits de poids faible et les 52 bits de poids fort. La réduction de la partie haute est effectuée de la même manière qu'avec les entiers en multipliant par c_i.

Algorithm 6.12 RNS AddMul (flottants à double précision)

Entrées : $m_i < 2^{52}$ et $c_i < 2^8$ tel que $m_i = 2^{52} - c_i$
$0 \leq x_i < 2^{52}$, $0 \leq y_i < m_i$ et $0 \leq \lambda < 2^{52-8}$
Sortie : $0 \leq z_i < 2^{52}$ tel que $z_i \equiv x_i + \lambda \times y_i \pmod{m_i}$
$(t_{iH}, t_{iL}) \leftarrow y_i \times \lambda$
$(t_{iHH}, t_{iHL}) \leftarrow \text{VeltkampSplit}\ (t_{iH}, 52)$
$z_i \leftarrow x_i + (t_{iHH} \gg 52) \times c_i + t_{iHL} + t_{iL}$
Si $z_i \geq m_i$ **alors**
  $z_i \leftarrow z_i - m_i$

6.3.2 RNS sur GPU

Sur les GPU NVIDIA, les types de données entiers possibles sont les entiers 8, 16 ou 32 bits et les vecteurs entiers qui dérivent du type `int` et qui l'étendent en vecteurs 64, 96 et 128 bits. Par exemple, le vecteur `int2` correspond à un entier 64 bits et est formé de deux composantes 32 bits `x` et `y`. Les types de données flottants sont les flottants à simple et à double précision (`float` et `double`) et les vecteurs qui en dérivent : `float2`, `float3`, `float4` et `double2` [CUDAa].

Pour implémenter les briques arithmétiques RNS, on utilise le langage d'assembleur PTX pour écrire des fonctions en assembleur dans un code CUDA (voir sous-section 2.3.7).

Nous avons implémenté et comparé l'arithmétique RNS pour différentes tailles des modules :
— 32 bits sur `uint` (entier non signé) ;
— 64 bits sur `uint2` ;
— 96 bits sur `uint3` ;
— 128 bits sur `uint4`.

Nous avons aussi utilisé les types flottants, où nous avons implémenté :
— des modules 23 bits sur `float` ;
— des modules 52 bits sur `double`.
Pour des nombres entre 100 et 1000 bits (c'est-à-dire l'intervalle qui nous intéresse dans le contexte de l'algèbre linéaire), les modules 64 bits sur `uint2` permettent d'atteindre les meilleures performances.

6.3.3 RNS sur CPU avec extensions SIMD

Nous avons implémenté les versions suivantes sur les différents jeux d'instructions SIMD (voir section 2.2) :
— Scalaire (en utilisant les instructions scalaires x68_64) : une version avec des modules 64 bits avec des entiers et une version avec des modules 52 bits avec des flottants.
— SSE2 : une version avec des entiers où un registre 128 bits contient 2 modules de 63 bits et une version avec des flottants, où un registre contient 2 modules de 52 bits.
— AVX2 : une version avec des entiers où un registre 256 bits contient 4 modules de 63 bits et une version avec 4 composantes flottantes, chacune correspondant à un module de 52 bits.

Nous comparons dans la table 6.1 les performances de ces versions sur un seul cœur d'un CPU Intel i5-4570 (3.2 GHz). On se munit d'une matrice exemple (voir table 5.1) et d'un nombre premier ℓ de 332 bits et on mesure le temps nécessaire pour effectuer un produit matrice-vecteur avec chaque implémentation.

Dans la première colonne, nous spécifions la base RNS nécessaire pour faire de l'arithmétique RNS avec le nombre ℓ. On utilise la notation $\mathcal{B}(n, k)$ pour désigner une base RNS de n modules, chaque module de taille k bits. Le nombre des modules pour les versions SSE est pair. Celui pour les versions AVX est multiple de 4.

Dans les trois colonnes suivantes, nous rapportons le temps en cycles par opération de chaque opération arithmétique (Add/Sub, AddMul/SubMul avec un coefficient petit (λ) et un coefficient grand (Λ) et la réduction modulo ℓ). Pour obtenir ces mesures, on considère deux vecteurs dont les éléments sont dans $\mathbb{Z}/\ell\mathbb{Z}$ représentés en RNS. Nous mesurons le temps moyen de l'exécution de chaque type d'opération, qu'on divise par la fréquence du CPU.

La dernière colonne rapporte le temps d'exécution du SpMV en ms. Dans la première ligne, nous indiquons les poids relatifs typiques de ces opérations dans un SpMV. Nous précisons les données relatives à l'opération AddMul avec un grand coefficient Λ. Toutefois, il est à noter que cette opération n'est présente que lorsque nous considérons des matrices NFS. Nous spécifions en gras l'implémentation qui permet de minimiser le temps du SpMV.

Opération	$x \pm y$	$x \pm \lambda y$	$x \pm \Lambda y$	$x \bmod \ell$	SpMV
Ratio d'occurrence dans un SpMV	88-91%	6-8%	0-4%	0.05-1%	100%
Temps avec Scalaire (entier) et $\mathcal{B}$(7,64)	17.3 cycles	83.1 cycles	438 cycles	1779 cycles	1561 ms
Temps avec Scalaire (flottant) et $\mathcal{B}$(8,52)	19.4 cycles	108 cycles	552 cycles	2267 cycles	1795 ms
Temps avec SSE2 (entier) et $\mathcal{B}$(8,63)	11.1 cycles	51.3 cycles	271 cycles	1183 cycles	1007 ms
Temps avec SSE2 (flottant) et $\mathcal{B}$(8,52)	12.5 cycles	66 cycles	339 cycles	1352 cycles	1146 ms
Temps avec AVX2 (entier) et $\mathcal{B}$(8,63)	**7.9 cycles**	**27.1 cycles**	**141 cycles**	**643 cycles**	**598 ms**
Temps avec AVX2 (flottant) et $\mathcal{B}$(8,52)	10 cycles	41.5 cycles	211 cycles	846 cycles	691 ms

TABLE 6.1 – Comparaison des performances des implémentations RNS sur différents jeux d'instructions SIMD.

On observe que pour tous les jeux d'instruction, utiliser des entiers est plus efficace qu'utiliser des nombres en virgule flottante. Avec un nombre ℓ de 332 bits, l'implémentation SSE2 est 55% plus rapide que l'implémentation MMX, l'implémentation AVX2 est 68% plus rapide que l'implémentation SSE2. Pour d'autres tailles de module, les pourcentages d'accélération peuvent être plus petits, vu que les conditions sur le nombre des modules (pair pour SSE et multiple de 4 pour AVX) peuvent alourdir les calculs. Toutefois, la version AVX2 avec des entiers est toujours la plus performante pour l'intervalle qui nous intéresse, celui entre 100 et 1000 bits.

La vectorisation fournie par le SIMD peut être utilisée autrement, pour effectuer en parallèle plusieurs produits matrice-vecteurs (exploiter le parallélisme de l'algorithme de Wiedemann par blocs, voir sous-section 3.5.3). Typiquement, au lieu d'utiliser un registre AVX2 pour tenir quatre résidus d'un élément d'un vecteur, on l'utilise pour contenir quatre résidus de quatre éléments

de quatre vecteurs. Ceci permet de passer outre la contrainte que la longueur de la base soit multiple du nombre de composantes dans le registre AVX2. D'autre part, effectuer un produit matrice–quatre-vecteurs est plus rapide que quatre produits matrice-vecteur.

À titre illustratif, reprenons l'exemple précédent. Un SpMV $v \leftarrow A \times u$ est effectué en 598 ms en AVX2 (entier), alors que si la vectorisation est utilisée pour effecteur 4 SpMV simultanément, 1651 ms sont nécessaires pour effectuer $(v_0, v_1, v_2, v_3) \leftarrow A \times (u_0, u_1, u_2, u_3)$. Ainsi, faire 4 produits ne coûte que 2.75 fois plus qu'un seul produit. Les quatre produits simultanés sont efficaces car la matrice A n'est lue qu'une seule fois.

6.4 RNS pour l'algèbre linéaire

Dans cette section, on s'intéresse à la question de comment choisir la base RNS pour effectuer un produit matrice-vecteur. Plus spécifiquement, quelle est la longueur minimale de la base RNS qui garantit que les calculs restent représentables (ne dépassent pas la borne maximale) ?

Nous allons considérer les deux types de matrices issues des algorithmes FFS et NFS (voir chapitre 3 pour plus de détails sur les caractéristiques de chaque type de matrice).

On reprend les mêmes notations que précédemment et on note r la norme maximale d'une ligne dans la matrice, définie comme la somme des valeurs absolues des coefficients de la ligne.

6.4.1 Matrices issues de FFS

On utilise une base RNS $\mathcal{B}(n, k)$, dont les modules sont proches de 2^k. Le vecteur d'entrée correspond au vecteur de sortie de l'itération précédente, donc les éléments du vecteur d'entrée sont dans $[0, n2^k\ell[$ (voir sous-section 6.2.6).

Proposition 1. *Étant donnés un nombre premier ℓ et une matrice FFS dont la norme maximale des lignes est r, le calcul d'un SpMV modulo ℓ est correct avec une base RNS $\mathcal{B}(n, k)$ si et seulement si*

$$rn2^k\ell < (1 - \Delta)M.$$

En choisissant $\Delta \ll 1$ et comme $M \approx 2^{nk}$, la condition précédente donne :

$$n \geq \left\lceil \frac{\log \ell + \log r + \log n + k}{k} \right\rceil. \tag{6.19}$$

En guise d'exemple, si on suppose que le logarithme de la norme maximale est égal à 10, ce qui est une valeur raisonnable pour les matrices FFS, on calcule dans la table suivante la taille minimale nécessaire de la base RNS de modules 64 bits, pour différentes tailles de ℓ.

Taille de ℓ en bits	91		217		511	
$\mathcal{B}(n, k)$	$(3, 64)$	$(4, 64)$	$(5, 64)$	$(6, 64)$	$(10, 64)$	$(11, 64)$
Fréquence de réduction modulo ℓ	1/2	1/8	1/2	1/9	1/5	1/11

Dans la table, on rapporte aussi la fréquence de la réduction modulo ℓ, si on envisage d'accumuler un certain nombre de SpMV avant de réduire modulo ℓ. Le nombre de produits accumulés avant une réduction est donné par $\left\lfloor \frac{nk - \log(n2^k\ell)}{\log r} \right\rfloor$ vu que, d'une itération à la suivante, les tailles des éléments des vecteurs augmentent d'au plus $\log r$ bits.

Nous avons déjà fait la remarque qu'en pratique, utiliser une base de taille minimale n'est pas nécessairement le choix optimal, puisqu'en prenant un module de plus, on diminue la fréquence de la réduction modulo ℓ qui est coûteuse.

6.4.2 Matrices issues de NFS

Une matrice issue de NFS est composée de 2 parties (voir sous-section 3.2.1) :
— Une partie similaire aux matrices issues de FFS composée de coefficients « petits », qu'on désigne par A_{SC} (SC pour *Small Coefficients*). On désigne toujours par r la norme maximale d'une ligne dans cette partie.
— Une partie composée de n_{SM} ($n_{\text{SM}} \ll N$) *colonnes de caractèress* et qui contient des coefficients « grands ». On désigne cette partie par A_{SM} (SM pour *Schirokauer Maps*).

Méthode naïve. Une première idée serait d'utiliser la même base $\mathcal{B}(n,k)$ pour les parties A_{SC} et A_{SM}. On note M le produit des modules de la base $\mathcal{B}$. Les éléments du vecteur d'entrée sont toujours $[0, n2^k\ell[$.

Proposition 2. *Étant donnés un nombre premier ℓ et une matrice NFS dont la norme maximale des lignes de sa partie A_{SC} est r et qui contient n_{SM}* colonnes de caractères*, le calcul d'un produit matrice-vecteur modulo ℓ est correct selon la méthode naïve avec une base RNS $\mathcal{B}(n,k)$ si et seulement si*

$$\begin{cases} rn2^k\ell & < & (1-\Delta)M & (\textit{le produit par } A_{SC} \textit{ ne déborde pas}) \\ n_{SM}\ell \times rn2^k\ell & < & (1-\Delta)M & (\textit{le produit par } A_{SM} \textit{ ne déborde pas}). \end{cases}$$

Ainsi,

$$n \geq \left\lceil \frac{2\log \ell + \log n_{\text{SM}} + \log r + \log n + k}{k} \right\rceil \tag{6.20}$$

Pour illustrer cette méthode concrètement, on prend une matrice NFS, dont la partie SC a une norme maximale égale à 10 et la partie SM est composée de 5 colonnes, et on calcule la taille nécessaire de la base pour différentes tailles de ℓ.

Taille de ℓ en bits	91	217	511
$\mathcal{B}(n,k)$	$(5,64)$	$(9,64)$	$(18,64)$

L'inconvénient de cette méthode est qu'on utilise une base qui est suffisamment grande pour la partie A_{SM} et trop grande pour la partie A_{SC}. Dans les deux prochains paragraphes, nous présentons d'autres méthodes qui permettent d'avoir des bases suffisamment grandes pour chaque partie.

Méthode par conversion. Approximativement, multiplier par la partie A_{SM} nécessite une base dont la taille est deux fois plus grande qu'une base suffisante pour la partie A_{SC}. On propose de prendre une base $\mathcal{B}(n,k)$ de taille minimale lorsqu'on multiplie par A_{SC} et une base $\tilde{\mathcal{B}}(\tilde{n}, 2k)$ dont les modules sont deux fois plus grands lorsqu'on multiplie par A_{SM}. On note M et $\tilde{M}$ les produits respectifs des modules des bases $\mathcal{B}$ et $\tilde{\mathcal{B}}$. Le calcul du produit $v = Au \bmod \ell$ s'effectue suivant ces étapes :

1. On calcule $v = A_{\text{SC}} \times u$ avec $\mathcal{B}(n,k)$.
2. On convertit v (représenté dans $\mathcal{B}(n,k)$) en $\tilde{v}$ (représenté dans $\tilde{\mathcal{B}}(\tilde{n}, 2k)$).
3. On calcule $\tilde{v} = (\tilde{v} + A_{\text{SM}} \times \tilde{u}) \bmod \ell$ dans $\tilde{\mathcal{B}}(\tilde{n}, 2k)$.
4. On convertit $\tilde{v}$ (représenté dans $\tilde{\mathcal{B}}(\tilde{n}, 2k)$) en v (représenté dans $\mathcal{B}(n,k)$).

Proposition 3. *Étant donnés un nombre premier ℓ et une matrice NFS dont la norme maximale des lignes de sa partie A_{SC} est r et qui contient n_{SM}* colonnes de caractères*, le calcul d'un SpMV modulo ℓ est correct selon la méthode par conversion avec les bases RNS $\mathcal{B}(n,k)$ et $\tilde{\mathcal{B}}(\tilde{n},2k)$ si et seulement si*

$$\begin{cases} r\tilde{n}2^{2k}\ell & < & (1-\Delta)M & (\textit{le produit par } A_{SC} \textit{ ne déborde pas}) \\ n_{SM}\ell \times r\tilde{n}2^{2k}\ell & < & (1-\Delta)\tilde{M} & (\textit{le produit par } A_{SM} \textit{ ne déborde pas}). \end{cases}$$

Ainsi,

$$n \geq \left\lceil \frac{\log \ell + \log r + \log \tilde{n} + 2k}{k} \right\rceil, \tilde{n} \geq \left\lceil \frac{2\log \ell + \log n_{\mathrm{SM}} + \log r + \log \tilde{n} + 2k}{2k} \right\rceil \tag{6.21}$$

Nous avons déjà vu dans la sous-section 6.2.7 un algorithme de conversion d'une base RNS à une autre.

Reprenons l'exemple précédent avec la méthode par conversion :

Taille de ℓ en bits	91	217	511
$\mathcal{B}(n,k)$	$(4,64)$	$(6,64)$	$(11,64)$
$\tilde{\mathcal{B}}(\tilde{n},2k)$	$(3,128)$	$(5,128)$	$(10,128)$

Méthode par extension. La méthode par conversion propose deux conversions lors de chaque SpMV. Le but de cette méthode est de remplacer les deux conversions par une seule extension. On utilise une base $\mathcal{B}(n,k)$ de taille minimale lorsqu'on multiplie par A_{SC}, qu'on étend en une base $(\mathcal{B}||\hat{\mathcal{B}})(n+\hat{n},k)$ lorsqu'on multiplie par A_{SM}. On note M et $\hat{M}$ les produits respectifs des modules des bases $\mathcal{B}$ et $\hat{\mathcal{B}}$. Le calcul du produit $v = Au \bmod \ell$ s'effectue suivant ces étapes :

1. On calcule $v = A_{\mathrm{SC}} \times u$ dans $\mathcal{B}(n,k)$.
2. On convertit v (représenté par $\mathcal{B}(n,k)$) en $\hat{v}$ (représenté par $\hat{\mathcal{B}}(\hat{n},k)$).
3. On calcule $v||\hat{v} = (v||\hat{v} + A_{\mathrm{SM}} \times u||\hat{u}) \bmod \ell$ dans $(\mathcal{B}||\hat{\mathcal{B}})(n+\hat{n},k)$, où l'opérateur $||$ désigne la concaténation des deux représentations RNS.

Proposition 4. *Étant donnés un nombre premier ℓ et une matrice NFS dont la norme maximale des lignes de sa partie A_{SC} est r et qui contient n_{SM}* colonnes de caractères*, le calcul d'un SpMV modulo ℓ est correct selon la méthode par extension avec les bases RNS $\mathcal{B}(n,k)$ et $\hat{\mathcal{B}}(\hat{n},k)$ si et seulement si*

$$\begin{cases} r(n+\hat{n})2^{k}\ell & < & (1-\Delta)M & (\textit{le produit par } A_{SC} \textit{ ne déborde pas}) \\ n_{SM}\ell \times r(n+\hat{n})2^{k}\ell & < & (1-\Delta)M\hat{M} & (\textit{le produit par } A_{SM} \textit{ ne déborde pas}). \end{cases}$$

Ainsi,

$$n \geq \left\lceil \frac{\log \ell + \log r + \log(n+\hat{n}) + k}{k} \right\rceil, n+\hat{n} \geq \left\lceil \frac{2\log \ell + \log n_{\mathrm{SM}} + \log r + \log(n+\hat{n}) + k}{k} \right\rceil \tag{6.22}$$

Regardons ce que donne cette méthode sur un exemple de matrice :

Taille de ℓ [bits]	91	217	511
$\mathcal{B}(n,k)$	$(3,64)$	$(5,64)$	$(10,64)$
$\mathcal{B} \cup \hat{\mathcal{B}}(n+\hat{n},k)$	$(5,64)$	$(9,64)$	$(18,64)$

La méthode par extension minimise le nombre de composantes RNS par rapport à la méthode naïve et réduit le nombre de conversions par rapport à la méthode par conversion.

Nous comparons à présent les performances de ces trois approches. Pour cela, nous considérons une matrice issue d'un calcul NFS (voir table 6.2) et mesurons le temps du SpMV sur une carte GeForce GTX 680 (voir la table 7.9).

Calcul	NFS pour $\mathbb{F}_{p_{155}}$
Taille de la matrice (N)	2.3M
Nombre de coefficients non nuls (n_{NZ})	230M
Nombre moyen de coefficients non nuls par ligne	100
Pourcentage de ± 1	89%
Nombre de *colonnes de caractères*	5

TABLE 6.2 – Caractéristiques de la matrice NFS pour $\mathbb{F}_{p_{155}}$.

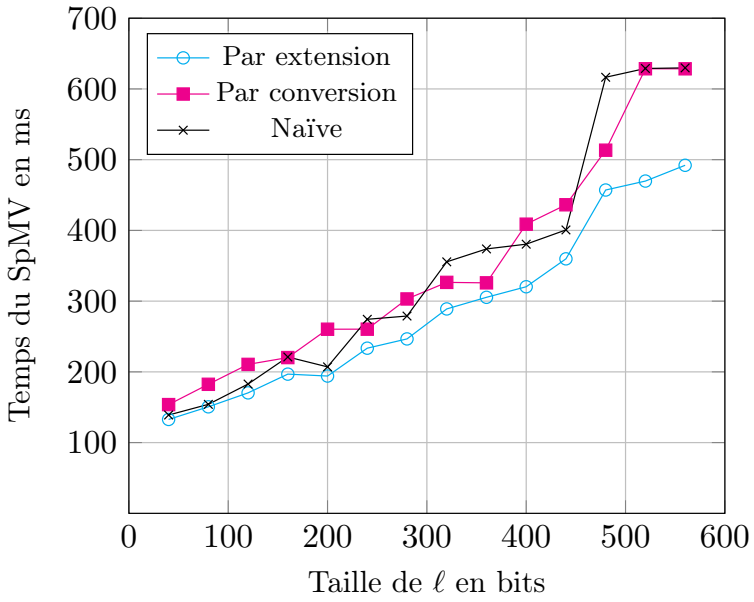


FIGURE 6.1 – Temps du SpMV avec les méthodes naïve, par conversion et par extension en fonction de la taille de ℓ.

Les méthodes naïve et par conversion sont compétitives. La première souffre d'un nombre trop grand de modules, alors que la seconde est ralentie par le conversions de RNS vers RNS. La méthode par extension est la plus efficace sur tout l'intervalle.

6.5 Comparaison des arithmétiques RNS et multi-précision (MP)

On rappelle que l'avantage de l'utilisation de l'arithmétique RNS par rapport à l'utilisation de l'arithmétique MP est qu'il n'y a pas de propagation de retenue avec l'arithmétique RNS. Dans cette section, nous rapportons quelques éléments quantitatifs de comparaison entre l'arithmétique RNS et Multi-précision (MP). Plus spécifiquement, on comparera les performances des deux

arithmétiques pour les opérations (Add/Sub, AddMul/SubMul avec un coefficient petit et avec un coefficient grand et la réduction modulo ℓ), ainsi que pour le SpMV.

6.5.1 RNS et MP sur GPU

Nous avons implémenté l'arithmétique MP en PTX. Pour l'addition et la multiplication, nous utilisons les algorithmes classiques en associant un *thread* à chaque mot machine. Pour la réduction modulo ℓ, on passe par un inverse pré-calculé de ℓ, ainsi la division par ℓ est obtenue par une seule multiplication. D'une manière similaire à ce qu'on a fait avec RNS, on choisit la base MP suffisamment large de sorte à pouvoir accumuler un certain nombre de produits matrice-vecteur avant de faire une réduction modulo ℓ.

Dans la table suivante, on compare les performances des implémentations RNS et MP sur la carte GeForce GTX 680 (voir table 7.9). On se munit d'un premier ℓ de 217 bits et d'une matrice-exemple (voir table 5.1).

Dans un premier temps, on mesure les temps de chaque opération arithmétique. De même que dans la sous-section 6.3.3, nous considérons deux vecteurs d'éléments dans $\mathbb{Z}/\ell\mathbb{Z}$, on mesure le temps moyen d'exécution de chaque type d'opération arithmétique sur un élément. Une opération arithmétique requiert 5 threads. Un *warp* effectue alors 6 opérations en parallèle. La carte GeForce GTX 680 possède 8 SM. Ainsi, 48 opérations sont exécutées en parallèle. Par conséquent, on divise le temps d'exécution total par le nombre d'éléments, puis on multiplie par 48. À partir du temps d'une opération, on obtient le nombre correspondant de cycles. La fréquence de la carte est 1 GHz. Dans un second temps, on mesure le temps du SpMV.

Opération	$x \pm y$	$x \pm \lambda y$	$x \pm \Lambda y$	$x \bmod \ell$	SpMV
Ratio d'occurrence dans un SpMV	88-91%	6-8%	0-4%	0.05-1%%	100%
Temps avec MP	2.9 cycles	**4.4 cycles**	36.5 cycles	**21.3 cycles**	31 ms
Temps avec RNS	**1.6 cycles**	5.1 cycles	**18.2 cycles**	81.5 cycles	**27.1 ms**
Accélération RNS/MP	1.8	0.87	2	0.26	1.14

L'arithmétique RNS permet de diminuer significativement le partage de données entre les *threads* et les instructions nécessaires à la génération et à la propagation des retenues. Sur un SpMV, l'implémentation RNS est approximativement 15% plus rapide.

Toutefois, si on observe les résultats de comparaison des deux arithmétiques pour chaque opération, on voit que l'arithmétique RNS est plus efficace pour les additions/soustractions et les multiplications par un coefficient grand, qu'elle est comparable à l'arithmétique MP pour la multiplication par un coefficient petit et qu'elle n'est pas du tout compétitive pour la réduction modulo ℓ.

L'accélération en faveur de RNS pour les additions/soustractions est annulée pour les multiplications par un coefficient petit, du fait que la réduction modulo m_i est peu chère dans le cas du RNS Add, alors qu'elle plus coûteuse dans le cas du RNS AddMul (voir algorithmes 6.9 et 6.10). Pour la multiplication par un coefficient grand, les complexités des opérations expliquent l'efficacité de RNS par rapport à MP. En effet, la multiplication MP est quadratique (ou sous-quadratique au mieux), alors que la multiplication RNS est linéaire. L'arithmétique RNS n'est pas du tout compétitive pour la réduction modulo ℓ. On observe très bien le coût quadratique de l'algorithme RNS de réduction, alors qu'avec la multi-précision, la réduction revient à faire un produit par l'inverse.

6.5.2 RNS et MP sur CPU

On reprend le premier de taille 217 bits et la même matrice. On mesure le temps que prend chaque opération arithmétique, ainsi que le produit matrice-vecteur. L'implémentation RNS utilise les AVX2 avec des entiers. Pour la version MP, on utilise la bibliothèque GMP, plus spécifiquement on se base sur la couche `mpn` qui regroupe les fonctions bas-niveau utiles pour des applications où le temps d'exécution est critique.

On reprend la même matrice que précédemment. Pour ℓ de 217 bits, la version MP utilise 4 mots machine, ainsi, une réduction modulo ℓ est effectuée à la fin de chaque produit matrice-vecteur, alors que la version RNS utilise 8 mots machine (5 mots machine auraient été suffisants, mais le nombre des modules doit être multiple de 4), la réduction modulaire est alors effectuée après une vingtaine de produits matrice-vecteur, son coût élevé est ainsi réparti sur les produits.

L'expérience a été réalisée sur un seul cœur d'un CPU Intel i5-4570 (3.2 GHz).

Opération	$x \pm y$	$x \pm \lambda y$	$x \pm \Lambda y$	$x \bmod \ell$	SpMV
Ratio d'occurrence dans un SpMV	88-91%	6-8%	0-4%	0.05-1%	100%
Temps avec MP (GMP `mpn`)	15 cycles	**26.8 cycles**	229.2 cycles	**184 cycles**	750 ms
Temps avec RNS	**7.9 cycles**	27.1 cycles	**139 cycles**	661.1 cycles	**584 ms**
Accélération RNS/MP	1.9	0.98	1.64	0.27	1.31

Les résultats de comparaison des deux versions CPU sont similaires à ce que nous avons observé sur GPU. L'implémentation RNS est plus rapide que la version MP pour l'addition et la multiplication avec un coefficient grand. La version MP est meilleure pour la multiplication par un coefficient petit et la réduction modulo ℓ. Comme autour de 90% des coefficients sont des ± 1, les opérations les plus utilisées sont les additions/soustractions. C'est pourquoi l'arithmétique RNS arrive à être autour de 30% plus rapide sur le SpMV.

Nous avons observé une accélération en utilisant l'arithmétique RNS plutôt que l'arithmétique MP, aussi bien sur GPU en comparant nos deux implémentations MP et RNS, que sur CPU en comparant notre implémentation RNS avec celle de GMP `mpn`. Néanmoins, il faut souligner qu'il est possible d'optimiser encore plus l'implémentation MP avec certaines techniques comme l'utilisation de la réduction Barrett ou la technique de *carry-save*. Un effort d'optimisation supplémentaire de l'arithmétique MP peut diminuer voire annuler l'accélération amenée par l'utilisation de la représentation RNS.

6.6 Conclusion

Dans ce chapitre, nous avons exploré l'intérêt d'utiliser une représentation qui fournit un degré significatif de parallélisme telle que la représentation RNS pour l'arithmétique sur des corps finis en grande caractéristique. Nous avons par ailleurs détaillé dans le cadre de notre contexte applicatif les choix et les approches pour implémenter efficacement cette représentation sur des architectures de type cartes graphiques ou processeurs avec SIMD.

Troisième partie

Résultats et conclusion

Chapitre 7

Calculs concrets de logarithme discret

Dans ce chapitre, nous présentons des détails sur trois calculs de logarithme discret. Au moment de leurs annonces, ces calculs étaient des records en terme de la taille des corps où le logarithme discret est calculé. À l'heure actuelle, le calcul dans $\mathbb{F}_{p_{180}}$ est toujours le record pour les corps finis premiers. Pour chaque record, nous rapportons des éléments quantitatifs sur toutes les étapes qui composent le calcul et nous détaillons le calcul d'algèbre linéaire. L'objectif est d'illustrer par des calculs concrets les approches et les choix que nous avons faits à travers ce mémoire.
Les résultats présentés dans ce chapitre ont fait l'objet des publications suivantes [Jel14a, Jel14b, BBD+14].

Sommaire

7.1 Calcul concret

Dans les chapitres précédents, nous avons détaillé l'application de différentes optimisations dans différents niveaux qui sont associés à un calcul d'algèbre linéaire. Cet effort est motivé par l'objectif de mener des calculs de logarithme discret rapides dans des groupes de plus en plus grands. Ceci nous a amené à réaliser deux implémentations de l'algorithme de Wiedemann par blocs :

— une implémentation adaptée à un cluster de GPU NVIDIA ;
— une implémentation adaptée à un cluster de CPU multi-cœurs.

Les codes des deux implémentations permettent de calculer les étapes de *Produits scalaires* et *Évaluation* de l'algorithme de Wiedemann par blocs et s'appuient sur le logiciel CADO-NFS pour calculer l'étape *Générateur linéaire*.

7.1.1 Implémentation GPU

L'implémentation GPU est actuellement sous la forme d'un package publié à l'adresse `http://www.loria.fr/~hjeljeli/`.

Le code a été optimisé pour les GPU NVIDIA de compute capability 3.0 et a été développé en utilisant principalement les langages CUDA et PTX. Les routines de l'algèbre linéaire et les fonctions de traitement de la matrice et des vecteurs sont écrites en CUDA. Les briques d'arithmétique RNS sont écrites en assembleur PTX. Pour les communications multi-GPU, nous avons combiné les directives CUDA et les directives *Cuda-aware MPI*.

7.1.2 Implémentation CPU

L'implémentation CPU est aussi sous la forme d'un package téléchargeable depuis la même adresse. La partie du code correspondant à l'arithmétique RNS est en cours d'intégration dans la bibliothèque MPFQ [MPFQ] qui permet de générer du code en langage C adapté à un corps fini (travail d'intégration réalisé avec Vialla et Sanselme).

Le code a été développé en utilisant le langage C et les directives MPI. L'implémentation fournit trois version possibles d'arithmétique :

— une version basée sur l'arithmétique multi-précision qui utilise la couche `mpn` de GMP ;
— une version qui implémente l'arithmétique RNS à travers les *intrinsics* SSE2 et qui nécessite un CPU qui supporte au moins le jeu d'instructions SSE4.2 ;
— une version qui implémente l'arithmétique RNS à travers les *intrinsics* AVX2 et qui nécessite une CPU qui supporte au moins le jeu d'instructions AVX2.

7.1.3 Organisation du calcul d'algèbre linéaire

Un calcul de Wiedemann ou de Wiedemann par blocs est organisé en 3 étapes (voir les sous-sections 3.5.2 et 3.5.3 pour Wiedemann et Wiedemann par blocs respectivement).

La première étape, *Produits scalaires*, consiste à calculer la séquence de Krylov ou une sous-séquence de Krylov s'il s'agit de la variante par blocs et que nous avons distribué le calcul sur autant de nœuds qu'il y a de séquences. Une itération typique d'un calcul de Wiedemann ou de Wiedemann par blocs, que nous indexons par j, correspond aux opérations suivantes :

— $v \leftarrow Av$ (SpMV) ;
— $a_j \leftarrow {}^t xv$ (produit scalaire).

La deuxième étape, *Générateur linéaire*, calcule le générateur linéaire de la séquence de sortie de l'étape *Produits scalaires*. La sortie de cette étape est le générateur F.

La troisième étape, *Évaluation*, calcule l'évaluation du polynôme F en la matrice A, multipliée par un vecteur z. Nous utilisons un schéma de type Hörner, de sorte à ce qu'on effectue exactement $\deg F$ produits. Notons $\hat{F}$ le polynôme réciproque de F. Si nous initialisons le vecteur w avec le vecteur nul, une itération d'indice j est alors effectuée comme suit :

— $w \leftarrow Aw + \hat{f}_j z$.

7.1.4 Gestion des erreurs et des pannes

Les calculs que nous allons exécuter vont prendre des temps relativement longs, qui peuvent aller de quelques heures à quelques mois. Il est possible d'avoir des interruptions accidentelles du calcul. C'est pourquoi nous mettons en place des sauvegardes périodiques des résultats intermédiaires sur un support de mémoire de masse. Ces sauvegardes permettent de reprendre le calcul. La périodicité de ces sauvegardes dépend de la durée du calcul. Par exemple, pour un calcul qui dure quelques heures, nous faisons des sauvegardes toutes les 120 minutes. Pour un calcul qui dure des jours, on peut envisager des sauvegardes quotidiennes.

Nous avons aussi observé des erreurs de calcul qui peuvent survenir avec un certain type d'architectures sur lesquelles nous effectuons les calculs, typiquement avec les cartes graphiques destinées pour les jeux vidéo. Ce qui nous a amené à mettre en place des vérifications périodiques pour pouvoir détecter et corriger une erreur survenue.

Le calcul de chacune des étapes *Produits scalaires* et *Évaluation* est alors divisé en tranches, une vérification de la cohérence du calcul est effectuée après chaque tranche. Pour cela, nous avons besoin de stocker les données de la tranche précédente ; ainsi, si le calcul s'avère erroné, il suffit de reprendre le calcul à partir de la tranche précédente qui a été vérifiée avec succès. Une tranche du calcul est composé d'un certain nombre d'itérations, par exemple toutes les 1000 ou 10000 itérations. Nous notons k le nombre d'itérations qui composent une tranche et nous allons expliqué le procédé de la vérification pour chaque étape.

Pendant l'étape *Produits scalaires*, nous vérifions la cohérence des vecteurs $A^i y$ calculés. Supposons que nous sommes sûrs du résultat de $A^i y$, pour un i donné. Après k itérations, nous voudrions vérifier que le vecteur $A^{i+k} y$ est correct. Admettons que nous avons deux vecteurs pré-calculés c_0 et c_k, tel que $c_k = ({}^t A)^k c_0$. Pour vérifier que $A^{i+k} y$ est correct, il suffit de vérifier l'égalité ${}^t c_k A^i y = {}^t c_0 A^{i+k} y$. Ce test assure avec une bonne probabilité que le calcul de $A^{i+k} y$ a réussi. Ce test ne nécessite que le pré-calcul des vecteurs c_0 et c_k et deux produits scalaires pour chaque vérification. Ainsi, il n'alourdit pas le calcul de l'étape.

Pour l'étape *Évaluation*, la vérification des résultats intermédiaires s'effectue avec un schéma similaire à celui de l'étape *Produits scalaires*. Pour cela, nous choisissons un nombre positif petit δ (typiquement < 10). Nous avons aussi besoin que z s'écrive comme une certaine puissance de la matrice A multipliée par y. Par souci de simplicité, prenons z égal à y. Nous sommes à l'itération j et nous voulons vérifier l'exactitude du calcul de w déjà effectué

$$w = \sum_{i=0}^{j} f_i A^i y.$$

En multipliant w par ${}^txA^\delta$, nous obtenons :

$$ {}^txA^\delta w = {}^txA^\delta \sum_{i=0}^{j} f_i A^i y = \sum_{i=0}^{j} f_i {}^txA^{\delta+i} y = \sum_{i=0}^{j} f_i a_{\delta+i}. $$

Par conséquent, le test de vérification de w revient à évaluer l'égalité ${}^txA^\delta w = \sum_{i=0}^{j} f_i a_{\delta+i}$. Ce test ne nécessite que la séquence de Krylov déjà calculée à la première étape.

7.2 Record de logarithme discret dans $\mathbb{F}_{2^{619}}$

Ce record a été réalisé par l'équipe Caramel et annoncé en octobre 2012 [BBD+12]. Ce calcul concerne les corps finis de caractéristique 2 et dont l'extension est un nombre premier. L'objectif est de résoudre le logarithme discret dans un sous-groupe multiplicatif du corps fini $\mathbb{F}_{2^{619}}$.

7.2.1 Record précédent

Quand ce calcul a été envisagé, le record de calcul de logarithme discret pour les corps binaires de degré d'extension premier était le calcul dans $\mathbb{F}_{2^{613}}$. Ce calcul a été effectué par Joux et Lercier et annoncé en septembre 2005 conjointement avec le calcul de logarithme discret dans $\mathbb{F}_{2^{607}}$ [JL05]. Dans ces deux calculs, l'algorithme FFS a été utilisé. L'objectif assumé de notre calcul était de battre le record par le plus petit incrément possible.

7.2.2 Les données du calcul

Le cardinal du groupe multiplicatif $\mathbb{F}_{2^{619}}^\times$ possède un diviseur premier ℓ de taille 217 bits (donné ci-dessous). L'objectif consistait à résoudre le problème du logarithme discret modulo ℓ dans le corps $\mathbb{F}_{2^{619}}$.

$$ \ell = 109378681671075297195692480234213908123642560192251038455204252439 $$

Comme pour les deux records précédents, l'algorithme FFS est utilisé. Le calcul est composé des 4 étapes suivantes :

1. **Sélection polynomiale :**
   Dans cette étape, la paire de polynômes (f, g) choisie est [9]

   $$ f(x,t) = x^6 + \texttt{0x7}x^5 + \texttt{0x6}x + \texttt{0x152a}, $$

   $$ g(x,t) = x + t^{104} + \texttt{0x6dbb}. $$

   On définit le corps $\mathbb{F}_{2^{619}}$ par le facteur $\varphi(t)$ de degré 619 du résultant de f et g en x. Le choix de la paire des polynômes est amplement détaillé dans [Bar13, chap. 9].
2. **Collecte de relations :**
   Le crible a pris un peu moins de 200 heures CPU pour collecter 20.5M relations faisant intervenir 10.5M idéaux [DGV13].

9. Nous représentons un polynôme de $\mathbb{F}_2$[t] par la valeur obtenue quand il est évalué à la valeur $t = 2$, notée en hexadécimal. Par exemple, `0x7` représente $t^2 + t + 1$.

3. **Filtrage :**
   Le filtrage a permis de générer à partir des relations une matrice carrée contenant 650k lignes et colonnes. L'article [Bou13] expose les méthodes utilisées et donne des données numériques sur les différentes étapes qui composent le filtrage.
4. **Algèbre linéaire :**
   Un élément du noyau de la matrice a été calculé avec l'algorithme de Wiedemann en utilisant une carte graphique. Ce calcul sera détaillé dans la sous-section suivante.

   À la fin de l'algèbre linéaire, le vecteur appartenant au noyau de la matrice contient les logarithmes discrets de certains idéaux. Pour un générateur z, nous avons obtenu :

$$\begin{aligned}\log_z(z+1) \quad &\equiv \quad 7204779598182633520945620695 1138 \setminus \\ &\quad\; 7255899891661996769519063164 26840 \pmod{\ell}.\end{aligned}$$

$$\begin{aligned}\log_z(z^2+z+1) \quad &\equiv \quad 3352748238664260008188488835101 \setminus \\ &\quad\; 4029057966752147859601106911 23980 \pmod{\ell}.\end{aligned}$$

7.2.3 Les données de l'algèbre linéaire

Les entrées de l'algèbre linéaire

Les entrées de l'algèbre linéaire sont le nombre premier ℓ et une matrice A qui contient 653358 lignes et colonnes et 65335817 coefficients non nuls ; ce qui nous donne une densité moyenne d'environ 100 coefficients non nuls par ligne. Le système est à résoudre sur le corps de base $(\mathbb{Z}/\ell\mathbb{Z})$ (voir table 7.1).

Taille de ℓ	217 bits
Taille de la matrice (N)	650k
Nombre de coefficients non nuls (n_{NZ})	65M
Pourcentage de ± 1	92.7%

TABLE 7.1 – Caractéristiques des entrées de l'algèbre linéaire pour le record dans $\mathbb{F}_{2^{619}}$.

Plate-forme de calcul

Au moment où le calcul a été envisagé, nous ne disposions que d'une seule carte graphique NVIDIA GeForce GTX 580 (voir table 7.2), installée sur une machine disposant d'un processeur Intel Xeon CPU E5440 et de 32 Go de RAM.

Génération	Fermi (compute capability 2.0)
Nombre de cœurs CUDA	512
Horloge processeur	1544 MHz
Mémoire globale	3072 Mo
Nombre de SM	16
Nombre maximal de *warps* par SM	48
Nombre maximal de *threads* par SM	1536

TABLE 7.2 – Spécifications GPU de la carte GeForce GTX 580.

Algorithme et paramètres optimaux

Comme nous disposons d'une seule unité de calcul, nous avons opté pour l'algorithme de Wiedemann simple, c'est-à-dire sans utiliser les blocs (voir sous-section 3.5.2). Toutefois, nous aurions aussi bien pu utiliser l'algorithme de Wiedemann par blocs avec les paramètres $n = 1$ et $m > 1$, typiquement $m = 2$ (voir sous-section 3.5.3). Rappelons que, le paramètre n correspond au nombre de séquences de calcul qui sont exécutées en parallèle (voir sous-section 3.5.4). C'est pourquoi prendre n égal à 1 s'imposait, alors que choisir m égal à 2, en l'occurrence, aurait permis de raccourcir la longueur de la séquence de Krylov à calculer. Rétrospectivement, le choix d'utiliser les blocs aurait été le meilleur, mais au moment de lancer le calcul, seul l'algorithme de Wiedemann simple avait été implémenté.

Ne disposant que d'une seule unité de calcul (le GPU), le SpMV n'est pas parallélisé. La matrice est représentée en utilisant le format qui a été expliqué dans le chapitre 5 et qui adapte le format classique CSR. Pour l'arithmétique, nous utilisons le système RNS, tel que décrit dans le chapitre 6.

Organisation du calcul

La figure 7.1 schématise le déroulement du calcul entre l'hôte (CPU) et le périphérique (GPU).

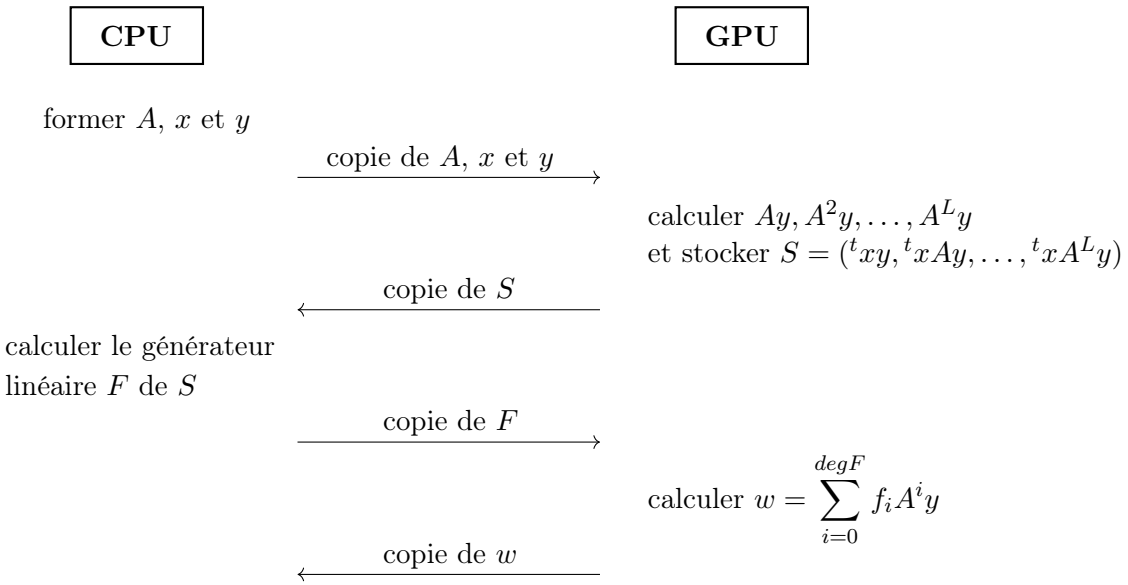


FIGURE 7.1 – Organisation du calcul de Wiedemann simple qui est exécuté sur un GPU et un CPU.

Nous commençons par construire les données sur le CPU : la matrice creuse A représentée dans un format de stockage adéquat et les deux vecteurs aléatoires x et y. Ces données sont ensuite copiées sur la mémoire globale du GPU.

L'étape *Produits scalaires* calcule la séquence de Krylov, qu'on note $S = (a_i)_{0 \leq i < L}$, avec $a_i = {}^txA^iy$. Cette étape est composée de L itérations, où $L = 2N + \varepsilon$, avec N la taille de la matrice.

Le résultat final, la séquence S, est transféré sur le CPU. Le calcul du générateur linéaire F de cette séquence est effectué sur CPU en utilisant le programme `plingen` du logiciel `CADO-NFS`

sur un processeur Intel i5-2500. Nous copions sur la mémoire globale du GPU le polynôme F représenté par ses coefficients f_i et un vecteur z. Ce vecteur peut correspondre au vecteur y ou à n'importe quel autre vecteur. Dans la figure 7.1, nous considérons que le vecteur z est égal au vecteur y.

La troisième étape, *Évaluation* calcule l'évaluation du polynôme F en la matrice A, multipliée par le vecteur z. Le calcul est composé de $\deg F$ itérations, où $\deg F$ est très proche de N.

Temps de calcul

Dans la figure 7.3 sont détaillés les temps de calcul des différentes étapes. Nous négligeons les temps des transferts entre le CPU et GPU qui sont inférieurs à la seconde.

Étape	Nombre d'itérations	Temps d'une itération	Débit en GOP/s	Temps total
Produits scalaires	1 306 816	27.8 ms	57.7	10.4 h GPU
Générateur linéaire	-	-	-	1 h CPU
Évaluation	653 458	30 ms	52	5.8 h GPU

TABLE 7.3 – Temps d'exécution des étapes de l'algèbre linéaire pour le record dans $\mathbb{F}_{2^{619}}$.

7.2.4 Conclusion

Ce calcul de logarithme discret a été fait *à la Bubka*, comme l'explicite Thomé dans [BBD+12], c'est-à-dire que la taille du record est juste supérieure à la taille du record précédent. Toutefois, il a permis de tester les différents bouts de code du logiciel `CADO-NFS`, qui ont été soit adaptés, soit écrits pour la première fois pour le contexte du logarithme discret. En effet, ce calcul a été le premier calcul complet de logarithme discret avec ce logiciel.

Plus spécifiquement à l'algèbre linéaire, ce calcul a permis de tester l'implémentation GPU de l'algorithme de Wiedemann simple. Même si les aspects liés au parallélisme tels que décrits dans les chapitres 3 et 4 n'ont pas été inclus, le calcul a permis de vérifier les choix et les approches utilisés pour la représentation de la matrice et pour l'arithmétique. La résolution d'un système linéaire de cette taille a permis aussi de se rendre compte des limites et des contraintes liées à l'utilisation des GPU, en particulier la taille limitée de la RAM. En effet, pour cette matrice, nous avons eu besoin d'environ 700 Mo de mémoire sur les 3 Go de RAM disponibles sur une carte GeForce GTX 580. Ainsi, nous avons pu conclure que sur un seul GPU, nous pourrons résoudre des matrices au plus 5 fois plus grandes. Pour des tailles plus grandes, nous avons besoin de nous orienter vers du multi-GPU ou vers d'autres architectures.

7.3 Record de logarithme discret dans $\mathbb{F}_{2^{809}}$

Ce calcul a été réalisé par l'équipe Caramel et annoncé en avril 2013 [BBD+13, BBD+14]. Ce record fait suite au record précédent et concerne aussi les corps finis de caractéristique 2 et de degré d'extension premier. L'objectif est de résoudre le logarithme discret dans un sous-groupe multiplicatif du corps $\mathbb{F}_{2^{809}}$.

7.3.1 Les données du calcul

Le sous-groupe de $\mathbb{F}_{2^{809}}$ dans lequel nous calculons le logarithme discret est d'ordre premier ℓ. L'ordre ℓ est un diviseur premier de $2^{809}-1$ de taille 202 bits et est donné par

$$\ell = 148386731260605647525186547488842396461625774241327567978137.$$

De même que pour le record précédent, l'algorithme FFS est utilisé. Quand ce calcul a été envisagé, les nouveaux algorithmes qui ont une meilleure complexité asymptotique que l'algorithme FFS venaient de voir le jour [Jou13a]. Nous savions que pour les petites tailles, l'algorithme FFS restait compétitif avec les nouveaux algorithmes. Mais, nous ignorions à partir de quelle taille les nouveaux algorithmes étaient plus efficaces en pratique que l'algorithme FFS. Par conséquent, une autre motivation de ce travail était de mieux quantifier cette limite, en particulier pour ce type de corps (de degré d'extension premier) qui pose plus de difficulté pour les nouveaux algorithmes.

Le calcul est composé des étapes suivantes :

1. **Sélection polynomiale** :
   Dans cette étape, la paire de polynômes (f, g) suivante, notée en hexadécimal, a été choisie :
   $$f(x,t) = x^6 + \texttt{0x7}x^5 + \texttt{0x6b}x^3 + \texttt{0x1ab}x^2 + \texttt{0x326}x + \texttt{0x19b3}$$
   $$g(x,t) = x + \texttt{0x80000000000000000000000000001e7eaa}$$
   Pour $f(x,t)$, le polynôme sélectionné a la meilleur valeur α de Murphy. Pour $g(x,t)$, il n'y a pas de contrainte particulière, un polynôme unitaire avec un terme constant creux est choisi. La sélection polynomiale a nécessité 115 jours CPU. Pour plus de détails, il est toujours recommandé de se référer à [Bar13, chap. 9].
2. **Collecte de relations** :
   Les principaux paramètres pour le crible sont la borne de friabilité B, qui est le plus grand degré de polynôme autorisé dans une relation et les paramètres I et J, qui correspondent aux dimensions de l'espace criblé.

   Pour ce calcul, deux ensembles de paramètres sont testés (voir table ci-dessous) [DGV13]. Dans le premier ensemble, $B = 27$ signifie que nous criblons sur tous les spécial-$\mathfrak{q}$ de degré entre 24 et 27 ; $I = J = 15$ signifie que nous criblons dans un carré de coté 2^{15}, ce qui fait que nous avons 2^{29} éléments à explorer par spécial-$\mathfrak{q}$. Le premier ensemble a donné 52M de relations et a nécessité 1550 jours CPU. Le second ensemble a nécessité 1121 jours CPU et a fourni 117M de relations. Ainsi, nous observons qu'avec une borne de friabilité plus petite ($B = 27$), il y a moins de relations à trouver, mais que chaque relation « coûte plus cher ».

B	I,J	Degré des spécial-$\mathfrak{q}$	#éléments explorés par spécial-$\mathfrak{q}$	#relations	Temps CPU (j CPU)
27	15	24 à 27	2^{29}	52M	1550
28	14	24 à 28	2^{27}	117.4M	1121

3. **Filtrage** : Nous orientons le lecteur vers le paragraphe 3.1 ou vers l'article [Bou13] pour une description détaillée des étapes du filtrage.

   Le filtrage est effectué sur les deux ensembles de relations produites par le crible avec les deux bornes de friabilité $B = 27$ et $B = 28$. Dans la table ci-dessous, nous détaillons l'évolution de le taille de la matrice pendant le filtrage.

B	27	28
#relations uniques	30.1M	67.4M
Taille de la matrice finale (N)	3.7M	4.8M

Il apparaît que le choix $B = 27$ s'avère plus judicieux. En effet, même si son crible a duré plus longtemps, ce choix nous a permis d'avoir une matrice plus petite et de donc diminuer la pression sur l'algèbre linéaire.

4. **Algèbre linéaire** :
   Un élément du noyau de la matrice a été calculé avec l'algorithme de Wiedemann par blocs en utilisant 8 cartes graphiques. Ce calcul sera détaillé dans la sous-section suivante.

   À la fin de l'algèbre linéaire, les logarithmes des éléments de la base de facteurs étaient déjà calculés, par exemple :

$$\log_t(t+1) \equiv 107082105171602535431582987436 \setminus 798986525914273094868488570257 4 \pmod{\ell}.$$

5. **Logarithme individuel** :
   Une descente classique par spécial-$\mathfrak{q}$ a été utilisée pour calculer le logarithme discret de n'importe quel élément, en l'occurrence ici RSA-1024 [10]. Nous avons écrit RSA-1024 en binaire et interprété cette écriture comme un polynôme de $\mathbb{F}_2[t]$. Nous avons réduit ce polynôme modulo le polynôme de définition φ pour obtenir un élément du corps $\mathbb{F}_{2^{809}}$. Le logarithme ci-dessous correspond au logarithme de cet élément.

$$\log_t(\text{RSA-1024}) \equiv 299978707191164348545002008342 \setminus 083497715498790833812541647079 6 \pmod{\ell}.$$

   Le calcul de logarithme individuel ne prend pas plus d'une heure CPU.

7.3.2 Les données de l'algèbre linéaire

Les entrées de l'algèbre linéaire

Les entrées de l'algèbre linéaire sont le nombre premier ℓ et une matrice A qui contient 3602667 lignes et colonnes, avec une densité moyenne d'environ 100 coefficients non nuls par ligne. Le système est à résoudre sur le corps de base $(\mathbb{Z}/\ell\mathbb{Z})$ (voir table 7.4).

Taille de ℓ	202 bits
Taille de la matrice (N)	3.6M
Nombre de coefficients non nuls (n_{NZ})	360M
Pourcentage de ± 1	92.8%

TABLE 7.4 – Caractéristiques des entrées de l'algèbre linéaire pour le record dans $\mathbb{F}_{2^{809}}$.

10. RSA-1024 = 135066410865995223349603216278805969938881475605667027524485143851526510604 \
859533833940287150571909441798207282164471551373680419703964191743046496589 \
274256239341020864383202110372958725762358509643110564073501508187510676594 \
629205563685529475213500852879416377328533906109750544334999811150056977236 \
890927563.

Plate-forme de calcul

Quand nous avons envisagé ce calcul, nous avions accès à un cluster composé de 4 nœuds ; chaque nœud contient deux cartes graphiques Tesla M2050 (voir table 7.5). Les nœuds sont reliés par des connexions InfiniBand QDR à 40 Gbit/s (voir sous-section 2.4.1) [11].

Génération	Fermi (compute capability 2.0)
Nombre de cœurs CUDA	448
Horloge processeur	1150 MHz
Mémoire globale	3072 Mo
Nombre de SM	14
Nombre maximal de *warps* par SM	48
Nombre maximal de *threads* par SM	1536

TABLE 7.5 – Spécifications GPU de la carte Tesla M2050.

Une partie du calcul qui s'exécute sur CPU a été effectuée sur un cluster de machines multi-cœurs. Le cluster était composé de 4 nœuds ; chaque nœud contenait un processeur Intel Xeon E5-2609 (2.4 GHz). Les nœuds étaient aussi reliés par des des connexions InfiniBand QDR.

Algorithme et paramètres optimaux

Dans cette algèbre linéaire, nous avons utilisé l'algorithme de Wiedemann par blocs, vu que nous disposions de plusieurs unités de calcul (voir sous-section 3.5.4).

Différents choix des paramètres (n, m) sont possibles. Le fait que nous disposons de 8 unités de calcul devrait nous orienter vers le choix le plus naturel, qui est de prendre $n = 8$; dans ce cas, 8 séquences seront exécutées en parallèle sur les 8 GPU, chaque séquence est exécutée d'une façon indépendante sur un GPU. Toutefois, la mémoire nécessaire pour représenter la matrice et les vecteurs d'entrée et de sortie est de 3.2 GB. Comme la mémoire disponible sur une carte graphique Tesla M2050 est seulement de 3 GB, la configuration $(n = 8, m = 16)$ n'est pas possible. Nous devons calculer une séquence sur plus d'une carte : en l'occurrence nous pouvons avoir les configurations $(n = 4, m = 8)$ et $(n = 2, m = 4)$. Dans la première configuration, 4 séquences sont exécutées en parallèle, chacune sur 2 GPU appartenant à un même nœud CPU ; le SpMV est donc parallélisé sur deux unités de calcul. Dans la seconde configuration, 2 séquences sont exécutées en parallèle, chacune sur 4 GPU appartenant à 2 nœuds CPU ; le SpMV est alors parallélisé sur 4 unités de calcul.

Théoriquement, la première configuration devrait convenir le mieux, vu que seuls 2 GPU installés sur le même nœud communiquent, alors que dans la seconde configuration, 4 GPU reliés par le réseau InfiniBand devraient communiquer. Dans la table 7.6, nous détaillons la description des deux configurations pour notre matrice de 3.6M lignes et colonnes. L'observation de la table nous montre que les communications GPU inter-nœuds pour la configuration $(n = 2, m = 4)$ ralentissent effectivement le temps total de la résolution, par rapport aux communications GPU intra-nœuds de la configuration $(n = 4, m = 8)$. Ceci peut être observé dans le ratio de la communication qui est plus important dans le cas $(n = 2, m = 4)$. Nous avons aussi ajouté des estimations du temps de la résolution pour une matrice plus petite (qui contient 3M lignes et

11. Travail effectué avec le support du Centre de compétences en calcul haute performance de la région Languedoc-Roussillon HPC@LR équipé d'un calculateur hybride IBM, financé par la région Languedoc-Roussillon, l'Europe et l'Université Montpellier 2 Sciences et Techniques.

colonnes) pour laquelle les trois configurations sont possibles et où il est possible de confirmer la prévision théorique, à savoir que le meilleur choix est celui de la configuration où n est égal au nombre des unités de calcul, c'est-à-dire quand il n'y a pas de communication.

Notons que le temps mentionné dans la table 7.6 est une estimation du temps d'exécution des étapes *Produits scalaires* et *Évaluation*. Cette estimation n'inclut donc pas l'étape *Générateur linéaire* parce que d'une part ce temps est difficile à estimer sans avoir au préalable terminé le calcul de *Produits scalaires* et d'autre part il est le plus souvent très petit devant les temps des deux autres étapes.

Taille de la matrice (mémoire requise)	Paramètres possibles	Temps prévu de la résolution	Ratio communications
3.6M (3.2 GB)	$\mathbf{(n = 4, m = 8)}$	**4.5 jours**	**16%**
	$(n = 2, m = 4)$	6 jours	37%
3M (2.7 GB)	$\mathbf{(n = 8, m = 16)}$	**2.5 jours**	**0%**
	$(n = 4, m = 8)$	3 jours	17%
	$(n = 2, m = 4)$	4.1 jours	38%

TABLE 7.6 – Données de résolution avec plusieurs paramètres (n, m) pour le record dans $\mathbb{F}_{2^{809}}$.

Pour la suite, nous allons utiliser la configuration $(n = 4, m = 8)$.

Organisation du calcul

La figure 7.2 représente comment est organisé le calcul de Wiedemann par blocs avec la configuration $(n = 4, m = 8)$ et exécuté sur 4 nœuds du cluster GPU.

Un nœud principal, ici le premier nœud, construit la matrice creuse A et les vecteurs blocs x et y. Il est indispensable que les vecteurs colonnes de y et les vecteurs lignes de x soient tous distincts. Le nœud principal copie la matrice et les différents vecteurs colonnes pour les différents nœuds.

Chaque nœud calcule sa sous-séquence de Krylov, qu'on désigne pour un nœud j par $S_j = \left({}^t xy^{(j)}, \ldots, {}^t xA^L y^{(j)}\right)$, où $y^{(j)}$ est la j^{e} colonne du bloc y. Le nombre d'itérations L est égal à $\left\lceil \frac{N}{4} \right\rceil + \left\lceil \frac{N}{8} \right\rceil + \varepsilon$. Toutes les sous-séquences sont collectées par le nœud 0 pour former la séquence $S = \left({}^t xy, \ldots, {}^t xA^L y\right)$.

Pendant la phase *Générateur linéaire*, un générateur linéaire matriciel de la séquence S est calculé en utilisant le logiciel `CADO-NFS` sur 16 cœurs du cluster CPU. Le lecteur intéressé par plus de détails sur l'algorithme de Berlekamp-Massey matriciel et son implémentation peut se référer à [Tho12b, chap. 12]. La sortie de cette phase est le générateur F.

La dernière étape, *Évaluation*, est effectuée en parallèle sur les 4 nœuds du cluster GPU. Un nœud d'indice j calcule le vecteur $w_j = F^{(j)}(A)y^{(j)}$. Quand tous les calculs ont abouti, le nœud principal collecte et somme les vecteurs w_j pour obtenir le vecteur w.

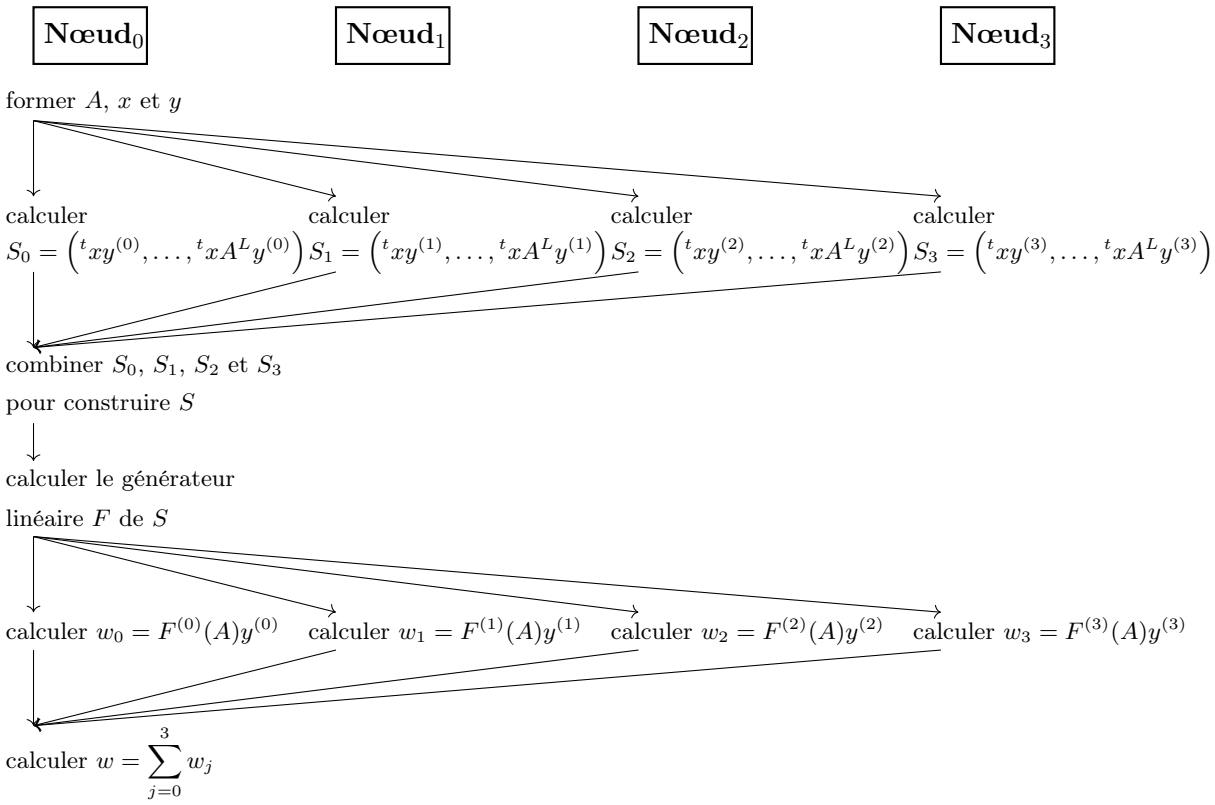


FIGURE 7.2 – Organisation du calcul de Wiedemann par blocs avec la configuration ($n = 4, m = 8$) et exécuté sur 4 nœuds.

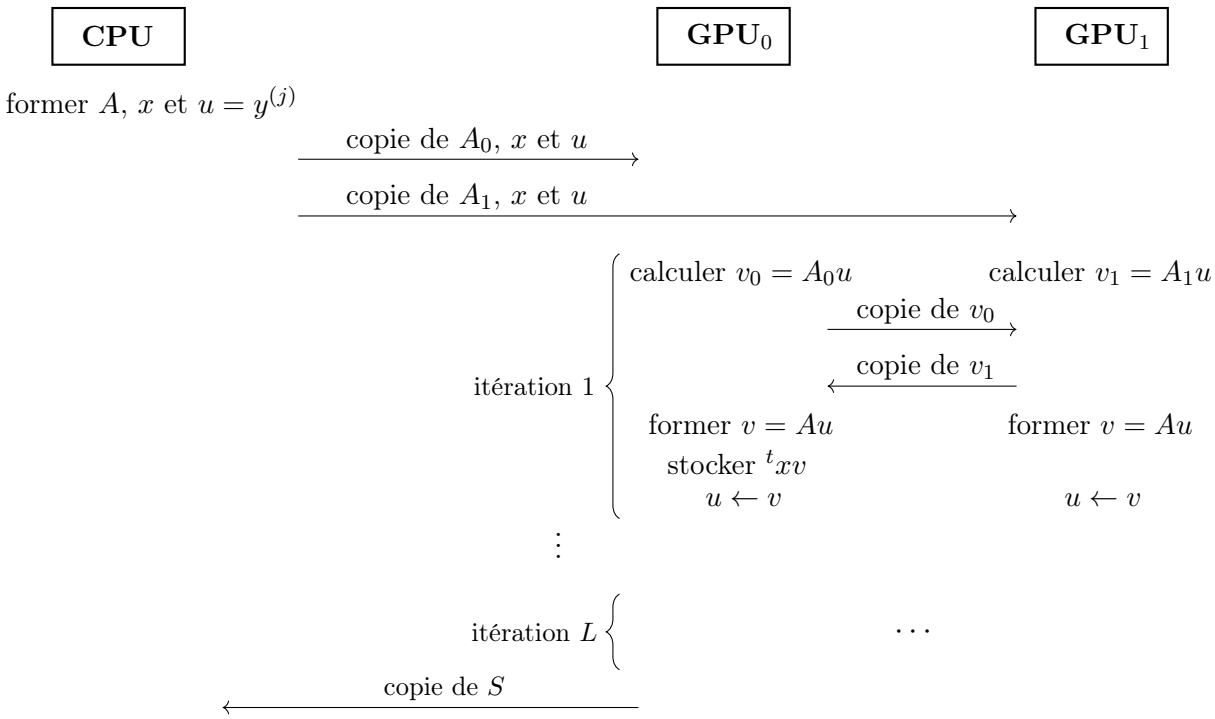


FIGURE 7.3 – Organisation du calcul de l'étape *Produits scalaires* sur un nœud d'indice j contenant 1 CPU et 2 GPU.

Pour les étapes *Produits scalaires* et *Évaluation*, un nœud effectue les calculs sur ses deux cartes graphiques. La matrice est divisée verticalement en deux sous-matrices A_0 et A_1 (voir le chapitre 5 pour la parallélisation du SpMV). Au début de l'étape, chaque sous-matrice est transférée du CPU vers le GPU correspondant. Pendant une itération, chaque GPU traite sa sous-matrice pour calculer son résultat partiel ; par la suite, les fragments des vecteurs sont échangés entre les deux GPU pour former le vecteur d'entrée de l'itération suivante. La figure 7.3 schématise le calcul de l'étape *Produits scalaires* pour un nœud donné.

Temps de calcul

Dans la table 7.7, sont détaillés les temps d'exécution des différentes étapes. Pour l'étape *Produits scalaires*, une itération nécessite 169 ms sur chaque nœud, dont 27 ms pour les communications GPU. Cette étape a nécessité 2.6 jours sur les 4 nœuds. L'étape *Générateur linéaire* effectué en parallèle avec 16 *threads* exécutés sur 16 cœurs CPU a pris 2 heures. Pour l'étape *Évaluation*, une itération prend 173 ms sur chaque nœud, dont 27 ms pour les communications GPU. Cette étape a nécessité 1.8 jours sur les 4 nœuds. Le temps total de l'algèbre linéaire est de 846 h GPU et 32 h CPU.

Étape	Nombre d'itérations	Temps d'une itération	Débit en GOP/s	Temps réel écoulé	Temps total
Produits scalaires	1.35M	169 ms	51.1	2.6 jours	500 h GPU
Générateur linéaire	-	-	-	2 h	32 h CPU
Évaluation	0.9M	173 ms	49.9	1.8 jours	346 h GPU

TABLE 7.7 – Temps d'exécution des étapes de l'algèbre linéaire pour le record $\mathbb{F}_{2^{809}}$.

7.3.3 Conclusion

Au niveau de l'algèbre linéaire, ce record a permis d'utiliser en pratique notre implémentation GPU de l'algorithme de Wiedemann par blocs avec 4 séquences en parallèle et où le SpMV de chaque séquence a été parallélisé sur 2 cartes graphiques. Donc, ce calcul a permis de tester et de valider les approches utilisées dans les différents niveaux de parallélisme, même si le nombre limité de ressources (8 GPU) ne nous permet de pousser encore plus le parallélisme.

Équilibrer le crible et l'algèbre linéaire

D'une manière rétrospective, nous nous sommes posé la question : quand devons-nous arrêter le crible ? Devons-nous arrêter le crible juste après avoir obtenu un nombre suffisant de relations ou faut-il continuer de chercher encore des relations pour pouvoir relâcher la pression sur l'algèbre linéaire ? En effet, avoir plus de relations permet pendant le filtrage d'obtenir une matrice plus petite.

Répondre à la question avec une approche théorique n'était pas possible. C'est pourquoi nous traitons la question d'une façon empirique, en mesurant les coûts du crible et de l'algèbre linéaire pour différents nombres de relations. Afin d'avoir une même métrique pour les coûts des étapes du crible et de l'algèbre linéaire, nous considérons le coût de l'algèbre linéaire en heures CPU, plutôt qu'en heures GPU. Nous estimons le coût de l'algèbre linéaire en heures CPU en utilisant notre implémentation CPU. Les résultats sont reportés dans la figure 7.4.

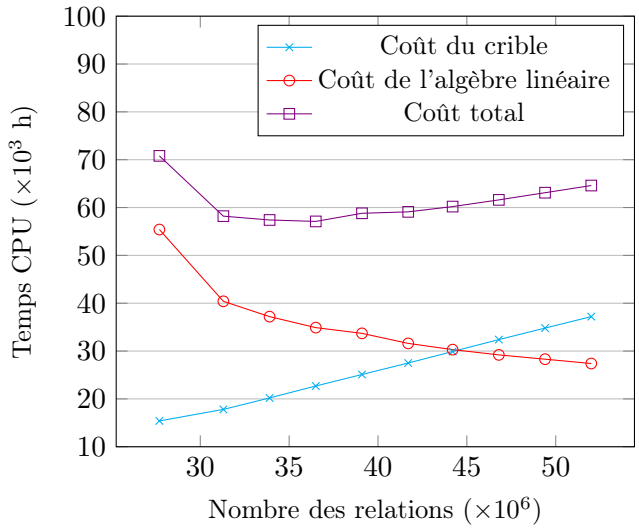


FIGURE 7.4 – Coûts du crible et de l'algèbre linéaire et coût total pour différents nombres de relations.

L'observation des courbes obtenues nous indique que la valeur optimale du nombre de relations pour minimiser le coût total des ces deux étapes est autour de 36M. Cette valeur a du sens si nous considérons l'implémentation CPU pour l'algèbre linéaire. Si nous utilisons les cartes graphiques pour la résolution, la contrainte de minimisation de la mémoire est aussi pertinente. En effet, nous avons vu dans la table 7.6 qu'une matrice plus petite permet de nous donner plus de liberté dans le choix des paramètres (n, m), ce qui peut encore diminuer le temps de calcul. De plus, on observe dans la figure 7.4 que cribler un peu plus ne rajoute pas un surcoût important au coût total. Par conséquent, avec du matériel avec des contraintes mémoire, il serait plus intéressant de continuer un peu plus le crible afin de faciliter l'algèbre linéaire.

Record $\mathbb{F}_{2^{1039}}$

Le record suivant à effectuer était de cibler un corps fini de taille un kilobit, plus spécifiquement le corps $\mathbb{F}_{2^{1039}}$. L'objectif était de résoudre le logarithme discret dans un sous-groupe de $\mathbb{F}_{2^{1039}}$, d'ordre premier ℓ de taille 265 bits.

Suffisamment de relations ont pu être collectées pendant la phase de crible. À la fin du filtrage, la matrice est formée de 60 millions lignes et colonnes, avec une densité moyenne de 100 coefficients non nuls par ligne. Une telle matrice nécessite environ 60 Go de RAM, alors que les cartes graphiques auxquelles nous avions accès possédaient chacune 4 Go de RAM. Dans ce cas, un schéma possible en supposant avoir accès à 16 cartes graphiques reliées avec de l'InfiniBand, est de diviser la matrice en 4×4 sous-matrices, chaque sous-matrice traitée par un GPU. Ce schéma est naturellement non-optimal, vu que le coût des communications entre 16 cartes graphiques sera très élevé. La résolution du système aurait nécessité avec ce schéma environ 76 mois.

Nous nous sommes alors orienté vers l'utilisation d'un cluster de processeurs multi-cœurs. Sur un cluster de 768 cœurs auquel nous avions accès (des détails sur ce cluster sont donnés dans la sous-section 7.4.3), notre implémentation CPU nécessiterait 15 mois en utilisant l'algorithme de Wiedemann par blocs avec la configuration $(n = 96, m = 192)$.

La résolution sur le cluster multi-cœurs est faisable ; mais nous ne l'avons pas lancée parce que

nous avions besoin de mieux explorer la faisabilité du calcul de générateur avec des paramètres (n, m) aussi grands. De plus, il est possible d'essayer d'autres choix de paramètres pour le crible qui pourraient donner une matrice plus petite.

Limite de faisabilité des calculs de logarithmes discrets dans $\mathbb{F}_{2^p}$ avec FFS

À partir des données du calcul de logarithme discret dans $\mathbb{F}_{2^{809}}$ et des estimations du calcul dans $\mathbb{F}_{2^{1039}}$, les limites d'utilisation de FFS pour attaquer les corps $\mathbb{F}_{2^p}$ sont plus claires. Nous pouvons conclure qu'atteindre le kilobit est faisable, mais nécessite des ressources conséquentes (en particulier pour l'algèbre linéaire).

Plus tard, Kleinjung a effectué un record de taille 1279 bits, calculé avec l'algorithme QPA avec des temps beaucoup plus petits que ceux de $\mathbb{F}_{2^{1039}}$ avec FFS [Kle14]. Ceci nous permet de conclure que l'algorithme QPA est plus efficace que FFS à partir du kilobit.

7.4 Record de logarithme discret dans $\mathbb{F}_{p_{180}}$

Ce record a été réalisé avec Bouvier, Gaudry, Imbert et Thomé et annoncé en juin 2014 [BGI+14]. Ce calcul a ciblé un corps fini premier en utilisant l'algorithme du crible algébrique NFS.

7.4.1 Record précédent

Le précédent record de calcul de logarithme discret sur les corps premiers est celui accompli par Kleinjung et al. pour un nombre premier p de taille 530 bit (160 chiffres décimaux) en utilisant l'algorithme NFS [Kle07].

7.4.2 Les données du calcul

Dans ce calcul, nous nous sommes intéressés au corps premier $\mathbb{F}_p$ avec p un nombre premier de taille 596 bits (180 chiffres décimaux) et qui est donné par

$$\begin{aligned} p &= \text{RSA-180} + 625942 \\ &= 191147927718986609689229466631454649812986246276667354864188503638 \setminus \\ &\quad 807260703436799058776201365135161278134258296128109200046702912984 \setminus \\ &\quad 568752800330221777752773957404540495707852046983. \end{aligned}$$

Le sous-groupe dans lequel le logarithme discret est calculé est d'ordre premier ℓ de taille 595 bits (179 chiffres décimaux) et qui est donné par

$$\begin{aligned} \ell &= (p-1)/2 \\ &= 955739638594933048446147333157273249064931231383336774320942518194 \setminus \\ &\quad 036303517183995293881006825675806390671291480640546000233514564922 \setminus \\ &\quad 843764001651108888763869787022702478539260234 91. \end{aligned}$$

Le calcul est effectué avec le logiciel `CADO-NFS` [CADO]. Il est composé de 5 étapes :

1. **Sélection polynomiale** :
   En utilisant l'algorithme de Kleinjung [Kle08], la sélection des polynômes a pris environ 2 mois CPU (sur Intel E5-2650 à 2 GHz). Les polynômes obtenus sont :

$$\begin{aligned} f(x) &= 17153280x^5 + 55645402596756x^4 + 2896424291003554669 45x^3 \\ &\quad -583903418367235648170825362 8x^2 - 348919545982234412735036794146466 0x \\ &\quad -247746689873713970845286181645074189 28. \end{aligned}$$

$$g(x) = 633287365084897327346023x - 256683250895227560765113615087202 91.$$

2. **Crible** :
   Les relations sont obtenues en criblant avec des *réseaux euclidiens*. Nous criblons sur des premiers algébriques et rationnels inférieurs à 800M et autorisons 2 nombres premiers inférieurs à 2^{29} du coté rationnel et 3 nombres premiers inférieurs à 2^{30} du coté algébrique. Nous criblons sur les spécial-$\mathfrak{q}$ entre 80M et 380M. Le crible a généré 253M relations et a pris 49.5 années CPU (sur Intel E5-2650 à 2 GHz).
3. **Filtrage** :
   Sur les 245M relations, il y a 175M relations uniques et 82M colonnes non vides. Au début du filtrage, une ligne de la matrice contient au plus une vingtaine de coefficients non nuls.

   Le filtrage a permis de produire une matrice finale qui contient 7.28M lignes et colonnes, avec 150 coefficients non nuls par ligne. Le procédé du filtrage proprement dit n'a nécessité que l'équivalent de 5 heures CPU sur un seul cœur (sur Intel E5-2650 à 2 GHz) ; mais, le calcul des *colonnes de caractères* a pris 0.9 années CPU (sur Intel E5-2650 à 2 GHz).
4. **Algèbre linéaire** :
   Le calcul d'un élément du noyau de la matrice est effectué avec l'algorithme de Wiedemann par blocs sur un cluster de processeurs multi-cœurs. Des détails sur cette étape sont donnés dans la sous-section suivante.

   Une fois l'algèbre linéaire terminée, nous obtenons les logarithmes discrets de plusieurs petits nombres premiers :

$$\begin{aligned}\log 2 \equiv\ & 14394742424980404689468652122583501155340452982569859698939499 5 \backslash \\ & 37509189519718986652049683275189725501776470006513329773475176 6 \backslash \\ & 54387676076061308411099885253085259407173106476434760 8 \pmod{\ell}.\end{aligned}$$

$$\begin{aligned}\log 3 \equiv\ & 12540255374709186945948836756152071692814462540757959805173613 9 \backslash \\ & 49252707487386035786690693592163692301618098936460400547559095 2 \backslash \\ & 63524577946074538124684456888597268322428333393912658 4 \pmod{\ell}.\end{aligned}$$

5. **Descente** : Le calcul du logarithme individuel est effectué avec une descente spécial-$\mathfrak{q}$ et a nécessité quelques heures. L'élément « arbitraire » pour lequel nous avons calculé le logarithme discret est RSA-1024 :

$$\begin{aligned}\log \text{RSA-1024} = \ & 13867056612682358487962586132633332631236394382562103922 0 \backslash \\ & 21558334615378333627255995552197035730130291204631078290 8 \backslash \\ & 65945075854910809291833135221575134605475521667300593993 3 \backslash \\ & 186397777 \pmod{\ell}.\end{aligned}$$

7.4.3 Les données de l'algèbre linéaire

Les entrées de l'algèbre linéaire

Les entrées de l'algèbre linéaire sont le nombre premier ℓ et une matrice A qui contient 7.28M lignes et colonnes, avec une densité moyenne d'environ 150 coefficients non nuls par ligne. En

plus des coefficients dont la taille ne dépasse pas un mot machine, la matrice contient 4 *colonnes de caractères* (voir la sous-section 1.4.4). Ces colonnes sont denses et leurs coefficients sont de la même taille que ℓ. Le système est à résoudre sur le corps de base $(\mathbb{Z}/\ell\mathbb{Z})$ (voir la figure 7.8).

Taille de ℓ	595 bits
Taille de la matrice (N)	7.28M
Nombre de coefficients non nuls (n_{NZ})	1.12G
Pourcentage de ± 1	87.6%
Pourcentage de coefficients inférieurs à 2^{10}	97.4%
Nombre de *colonnes de caractères*	4

TABLE 7.8 – Caractéristiques des entrées de l'algèbre linéaire pour le record dans $\mathbb{F}_{p_{180}}$.

Plate-forme de calcul

Les ressources auxquelles nous avons eu accès pour ce calcul sont deux clusters. Le premier est équipé de cartes graphiques. Le second, le cluster *catrel*, est un cluster de machines équipées de processeurs multi-cœurs.

Le cluster de cartes graphiques est composé de 4 nœuds ; chaque nœud contient deux cartes graphiques GeForce GTX 680 (voir la table 7.9). Les nœuds sont reliés par des connexions InfiniBand QDR à 40 Gbit/s (voir la sous-section 2.4.1).

Génération	Kepler (compute capability 3.0)
Nombre de cœurs CUDA	1536
Horloge processeur	1006 MHz
Mémoire globale	4096 Mo
Nombre de SM	8
Nombre maximal de *warps* par SM	64
Nombre maximal de *threads* par SM	2048

TABLE 7.9 – Spécifications GPU de la carte GeForce GTX 680.

Le cluster *catrel* contient 768 cœurs. Il est composé de 48 nœuds connectés avec des connexions InfiniBand FDR à 56 Gbit/s (voir la sous-section 2.4.1) ; chaque nœud possède 2 processeurs Intel Xeon E5-2650 (2 GHz) avec 8 cœurs dans chaque processeur.

Algorithme et paramètres optimaux

Dans cette algèbre linéaire, nous utilisons l'algorithme de Wiedemann par blocs. Nous avons accès à deux types de ressources. Au moment du calcul, nous n'avions pas envisagé de combiner les deux ressources, même si l'algorithme de Wiedemann par blocs offrait une indépendance des calculs qui aurait facilité l'utilisation simultanée de deux clusters. La difficulté venait du fait que les vitesses de calcul sont très différentes sur les deux clusters. En effet une séquence lancée sur le cluster de GPU aura une vitesse beaucoup plus grande qu'une séquence lancée sur un certain nombre de nœuds du cluster CPU. Il existe des solutions pour traiter des séquences déséquilibrées, en l'occurrence cette question a été étudiée dans [Tho12b, chap. 12]. Toutefois, au moment de lancer le calcul, par souci de simplicité, nous n'avions pas considéré la possibilité d'utiliser conjointement les deux ressources. Rétrospectivement, ce scénario aurait été envisageable.

Dans un premier temps, nous considérons le cluster GPU. La matrice représentée sous un format compressé, ainsi que les vecteurs d'entrée et de sortie, occupent un peu moins de 10 Go. Sachant que la mémoire disponible sur une carte graphique est de 4 Go, il faut qu'au moins 3 GPU collaborent ensemble pour effectuer un SpMV. Par conséquent, avec les 8 cartes graphiques dont nous disposons, la configuration qui est la plus adaptée est $(n = 2, m = 4)$, i.e., chaque groupe de 4 GPU calcule une séquence. Avec cette configuration, le temps prévu de l'algèbre linéaire serait de 65 jours, sans tenir compte du temps de calcul du générateur linéaire.

Taille de la matrice (mémoire requise)	Paramètres possibles	Temps prévu de la résolution	Ratio communications
7.3M (9.8 Go)	$(n = 2, m = 4)$	65 jours	32%

Maintenant, réfléchissons sur l'utilisation de notre implémentation CPU sur le cluster de processeurs multi-cœurs.

Un choix trivial des paramètres (n, m) serait de prendre $(n = 48, m = 96)$ de tel sorte qu'une séquence est calculée sur un seul nœud. Chaque nœud dispose de 16 cœurs, donc la matrice sera exactement divisée en 4×4 sous-matrices. Toutefois, cette configuration qui optimise les communications pose la difficulté que les paramètres (n, m) sont très grands, ce qui rendrait l'étape du calcul du générateur linéaire très dure, voire infaisable.

Par conséquent, il serait intéressant de considérer des configurations pour lesquelles n est plus petit. La configuration suivante est $(n = 24, m = 48)$, où une séquence est exécutée sur deux nœuds. Ainsi, 32 cœurs qui s'exécutent sur 2 nœuds collaborent ensemble pour calculer une séquence. Comme 32 n'est pas un carré, la division de la matrice en sous-matrices n'est pas optimale. On est amené à diviser la matrice en 5×5 et uniquement 25 cœurs sur les 32 seront utilisés. La même difficulté se pose pour la configuration $(n = 16, m = 32)$.

La configuration $(n = 12, m = 24)$ permet d'exécuter une séquence sur 4 nœuds. Par conséquent, la matrice est divisée en 8×8. Prendre un n plus petit implique certes qu'il y aurait beaucoup de communications, notamment des communications inter-nœuds. Cependant, cette configuration satisfait le compromis de ne pas trop augmenter le ratio de communications et de garder faisable le calcul du générateur linéaire.

Dans la table 7.10, nous résumons les données de la résolution avec les différentes configurations, sans tenir compte du temps de calcul du générateur linéaire. Si nous ne considérons pas le calcul du générateur linéaire, la configuration $(n = 48, m = 96)$ est la plus rapide. La configuration $(n = 12, m = 24)$ est meilleure que la configuration $(n = 24, m = 48)$, parce qu'elle utilise les 768 cœurs, alors que la dernière utilise uniquement 600 cœurs.

En définitif, nous garderons pour la suite le choix $(n = 12, m = 24)$, qui, même s'il n'était pas aussi rapide que le choix $(n = 48, m = 96)$, nécessitera un calcul de générateur linéaire moins difficile.

Paramètres possibles	Division de la matrice	Temps d'une itération	Nombre d'itérations	Temps prévu de la résolution
$(n = 48, m = 96)$	4×4	6.9 s	380k	31 jours
$(n = 24, m = 48)$	5×5	4.6 s	759k	41 jours
$\mathbf{(n = 12, m = 24)}$	**8×8**	**2.1 s**	**1.52M**	**37 jours**

TABLE 7.10 – Données de résolution avec plusieurs paramètres (n, m) pour le record dans $\mathbb{F}_{p_{180}}$ sur le cluster *catrel*.

En comparant les données de la résolution pour les ressources GPU et CPU, nous observons qu'avec un petit nombre de cartes graphiques, le cluster CPU serait plus adapté que le cluster GPU. Maintenant, supposons que nous ayons eu un cluster similaire au cluster *catrel*, mais contenant deux cartes NVIDIA GeForce GTX 680 dans chaque nœud. La résolution avec les 96 GPU en utilisant l'algorithme de Wiedemann par blocs avec la configuration $(n = 24, m = 48)$ aurait nécessité 5 jours et demi. Cette estimation n'est pas spéculative, puisque les données obtenues avec 8 cartes graphiques s'étendent parfaitement pour 96 cartes grâce au parallélisme de l'algorithme de Wiedemann par blocs.

Prendre en compte les *colonnes de caractères*

Nous revenons maintenant sur les contraintes rajoutées par la prise en compte des *colonnes de caractères* dans les matrices NFS et illustrons ces contraintes sur le record dans $\mathbb{F}_{p_{180}}$. Nous rappelons que les *colonnes de caractères* sont des colonnes denses qui contiennent des coefficients « grands ».

Nous avons déjà évoqué dans le chapitre 6 l'impact des coefficients « grands » sur l'arithmétique RNS. En effet, sans les *colonnes de caractères*, les opérations arithmétiques effectuées sont de la forme $x \leftarrow x + \lambda y$, où x et y sont des entiers dans $\mathbb{Z}/\ell\mathbb{Z}$ et λ un entier « petit ». Par conséquent, nous avons besoin en RNS d'un accumulateur dont la taille est plus grande que ℓ d'un mot machine, ce qui se traduit par un certain nombre de modules RNS. Pour traiter les *colonnes de caractères*, il faut aussi considérer des opérations arithmétiques de la même forme, mais avec un λ « grand ». Par conséquent, nous avons besoin d'agrandir la représentation RNS, i.e., le nombre des modules est quasiment doublé (voir sous-section 6.4.2). Si nous utilisons l'arithmétique RNS, le surcoût des *colonnes de caractères* sur le délai d'un SpMV est d'environ 30%. Si nous utilisons l'arithmétique MP, le surcoût est d'environ 10%. Au final, l'utilisation de l'arithmétique MP s'avère plus efficace que celle de l'arithmétique RNS, alors que sans les *colonnes de caractères*, l'arithmétique RNS aurait offert un net avantage.

La seconde contrainte concerne l'augmentation de la mémoire nécessaire pour représenter la matrice. La première matrice obtenue après le filtrage en fixant la densité moyenne à 100 coefficients non nuls par ligne avait 10.2M lignes et colonnes. La mémoire nécessaire pour représenter la matrice s'élevait à 7.9 Go, dont 3.3 Go pour les *colonnes de caractères*. C'est pourquoi nous avons choisi de pousser un peu plus le filtrage, autrement dit d'avoir une matrice plus petite et plus dense de sorte à diminuer l'impact des *colonnes de caractères*. Une seconde matrice obtenue avec une densité moyenne de 150 coefficients non nuls par ligne avait 7.3M lignes et colonnes. La mémoire nécessaire est alors de 7 Go, dont 2.3 Go pour les *colonnes de caractères*. S'il n'y avait pas de *colonnes de caractères*, la mémoire nécessaire aurait été 30% plus petite et on aurait pu avoir sur GPU des configurations de blocs plus intéressantes, typiquement $(n = 4, m = 8)$.

Organisation du calcul

Rappelons les données du calcul. Il s'agit de lancer un calcul de Wiedemann par blocs avec 12 séquences indépendantes sur les 48 nœuds du cluster *catrel*. Le schéma général est similaire à celui présenté dans la figure 7.2. Un groupe de 4 nœuds traite une séquence. La matrice est divisée en 64 sous-matrices ; chaque sous-matrice est traitée par un processus MPI qui s'exécute sur un cœur. Au début du calcul, un processus charge la sous-matrice et le fragment du vecteur d'entrée correspondants. Une fois le calcul partiel effectué, la combinaison des résultats partiels est réalisée avec des directives MPI selon le schéma décrit dans la section 4.3.

Comme le cluster est aussi utilisé par d'autres utilisateurs, nous n'avons pas un accès exclusif aux 48 nœuds pendant toute la durée du calcul et les ressources disponibles variaient au cours du temps. Nous divisons alors le calcul de chaque séquence en plusieurs tâches. Une tâche ou un *job* correspond à un nombre fixé d'itérations sur une séquence donnée. Une tâche correspond à environ 36 heures de calcul, ce qui correspondait à 1/16 du temps de calcul de la phase *Produits scalaires* et à 1/11 du temps de calcul de la phase *Évaluation* pour une séquence. Les tâches sont ordonnancées automatiquement. Quand une tâche est terminée ou quand un autre utilisateur libère des ressources, l'ordonnanceur lance une nouvelle tâche qui va calculer sur la séquence libre qui a le moins avancé. Tous les nœuds ont accès à un espace de partage commun, où les résultats des tâches sont stockés. Ce schéma nous permet de pouvoir utiliser des ressources variables et de garantir que les calculs sur les 12 séquences avancent quasiment à la même vitesse.

La figure 7.5 obtenue avec l'outil *Gantt Chart* du cluster *catrel*, montre le suivi de l'avancement de calcul pour l'étape *Évaluation*. En abscisse, le temps est représenté avec les dates des jours. En ordonnée, sont décrits les différents nœuds de la grille de calcul. Chaque bloc spécifié par un numéro et une couleur correspond à une tâche donnée.

Le mode de fonctionnement avec des tâches ordonnancées et des vérifications et des enregistrements périodiques nous a permis de gérer des comportements « curieux » de certains nœuds de calcul, liés à des problèmes avec les alimentations et les *c-states*.

Temps de calcul

La table 7.11 détaille les temps d'exécution des différentes étapes de l'algèbre linéaire. Pour l'étape *Produits scalaires*, une itération est effectuée en 2.1 s, dont 0.4 s pour les communications. Cette étape a nécessité au total 22 jours, ce qui est équivalent à 46 années sur un seul cœur. L'étape *Générateur linéaire* calculée en parallèle avec 144 *threads* exécutés sur 144 cœurs du cluster *catrel* a pris 15 heures. Pour l'étape *Évaluation*, une itération prend 2.1 s. Cette étape a requis 16 jours, i.e. 34 années CPU. Au total, le temps de la résolution du système linéaire a été de 80 années CPU.

Étape	Nombre d'itérations	Temps d'une itération	Débit en GOP/s	Temps réel écoulé	Temps total
Produits scalaires	913k	2.1 s	23.4	22 jours	46 années CPU
Générateur linéaire	-	-	-	15 h	3 mois CPU
Évaluation	610k	2.1 s	23.4	16 jours	34 années CPU

TABLE 7.11 – Temps d'exécution des étapes de l'algèbre linéaire pour le record dans $\mathbb{F}_{p_{180}}$, avec la configuration $(n = 12, m = 24)$.

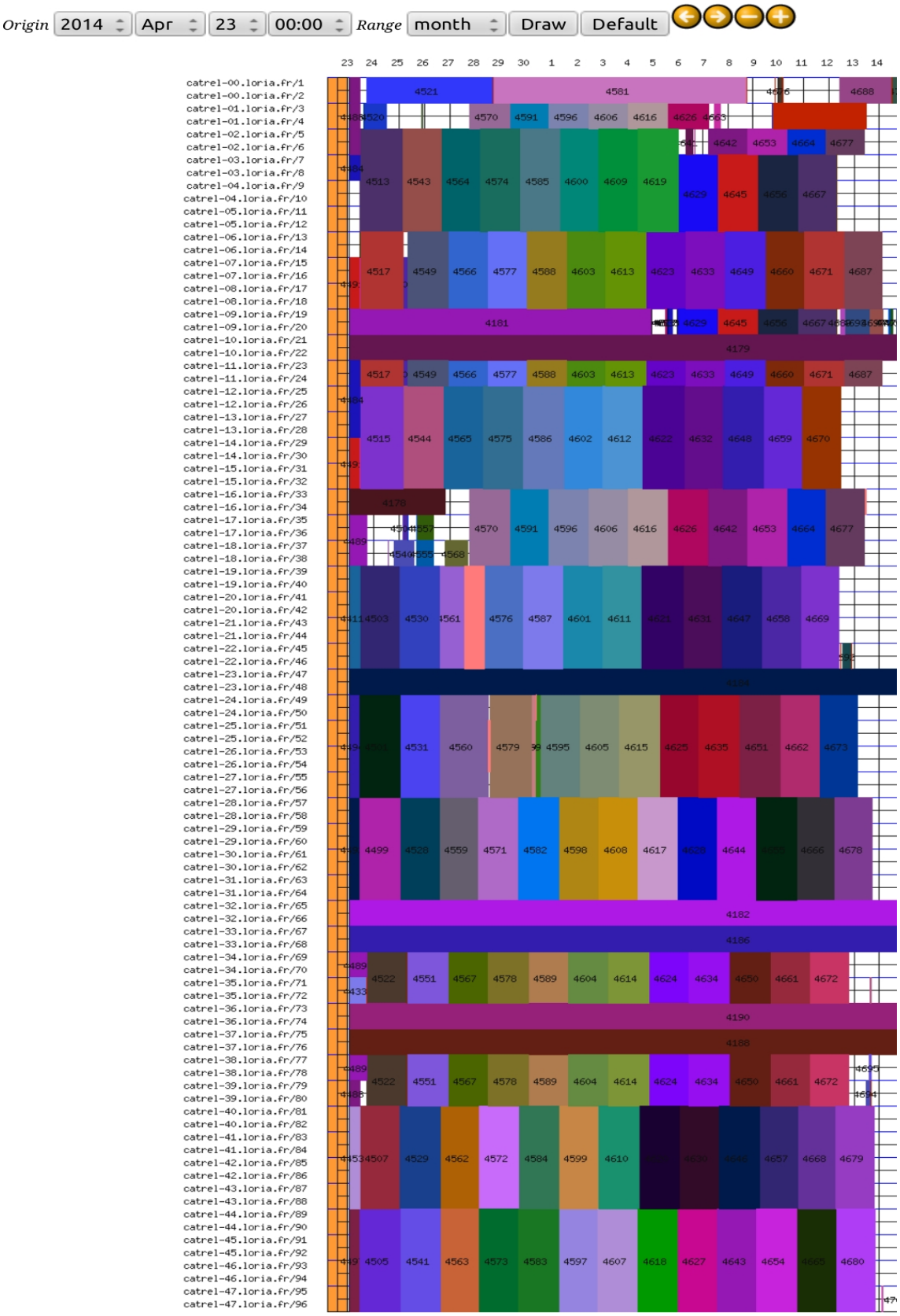


FIGURE 7.5 – Diagramme de Gantt de l'utilisation du cluster *catrel* pour l'algèbre linéaire du record dans $\mathbb{F}_{p_{180}}$.

7.4.4 Conclusion

Le calcul d'algèbre linéaire nous a amené à résoudre un système linéaire contenant 7.8M lignes et colonnes, défini modulo un nombre de 595 bits, en utilisant l'algorithme de Wiedemann par blocs avec les paramètres $(n = 12, m = 24)$.

Ce calcul a permis d'utiliser en pratique notre implémentation CPU de l'algorithme de Wiedemann par blocs. De plus, effectuer une résolution avec les paramètres $(n = 12, m = 24)$ a permis de mieux clarifier la faisabilité du calcul du générateur linéaire avec des *blocking factors* aussi grands. Plus précisément, augmenter ces paramètres peut amener à un calcul d'une part long, et d'autre part qui nécessite beaucoup de mémoire. C'est pourquoi nous avons privilégié la configuration $(n = 12, m = 24)$ par rapport à la configuration $(n = 48, m = 96)$. Pour ce calcul de générateur linéaire, effectué en parallèle sur 144 processeurs, la RAM nécessaire pour chaque processeur était de 10 Go.

Le cluster CPU a été plus adapté pour ce calcul que le cluster GPU. Ceci est dû au nombre limité de cartes graphiques dans le cluster GPU. Nous avons vu dans la sous-section 7.4.3 qu'avec un cluster équivalent au cluster *catrel* et pourvu de GPU, le temps de la résolution est réduit d'au moins un facteur 7. Le coût de rajouter 96 cartes graphiques au cluster *catrel* est estimé à environ 80k€, alors que le coût du cluster s'élève à 200k€. Ainsi, on voit qu'avec un coût moins élevé, l'accélération fournie par le GPU l'emporte sur celle fournie par le multi-cœur. Toutefois, ce propos mérite d'être nuancé. En effet, nous avons observé notamment avec le calcul dans $\mathbb{F}_{2^{1039}}$ que pour des grandes tailles de la matrice, la mémoire limitée des GPU fait que la résolution n'est plus possible et que dans ce cas, seules des machines multi-cœurs pourvues de beaucoup de mémoire sont adaptées à l'algèbre linéaire.

Conclusion générale

Contributions

Dans le cadre de ce travail, nous nous sommes intéressé aux différents éléments qui permettent d'accélérer le calcul d'algèbre linéaire issue des calculs de logarithme discret sur les corps finis.

Nous avons exploré et comparé dans le chapitre 2 les architectures parallèles appropriées à ce type de calcul.

Dans le chapitre 4, nous avons étudié la parallélisation sur plusieurs nœuds de l'opération centrale des calculs d'algèbre linéaire, qui est le produit matrice-vecteur. Nous avons en particulier réfléchi sur comment équilibrer les charges de travail et répartir les processus de sorte à minimiser les coûts des communications.

Nous avons étudié dans le chapitre 5 comment effectuer le produit partiel matrice-vecteur. Nous avons considéré deux architectures : les GPU et les CPU. Pour chaque architecture, nous avons exploré les formats classiques de représentation des matrices creuses et nous avons implémenté les algorithmes du produit matrice-vecteur (SpMV) en les adaptant à notre contexte d'algèbre linéaire sur des corps finis de grande caractéristique. Après avoir comparé les performances des différents formats, nous avons déduit que le format CSR est le plus efficace en terme de temps du SpMV. Par la suite, nous avons ajouté des optimisations qui exploitent certaines spécificités des matrices et qui tiennent compte des caractéristiques matérielles de chaque architecture.

Dans le chapitre 6, nous nous sommes intéressé à la parallélisation de l'arithmétique dans le corps fini de grande caractéristique sur des unités vectorielles, pour les *threads* du GPU ou avec les instructions SIMD du CPU. Nous avons montré que l'utilisation de la représentation RNS, qui élimine la propagation de retenue entre les unités de calcul, présente un intérêt particulier sur ce type d'architectures. Nous avons implémenté et comparé différentes versions d'arithmétiques RNS sur les GPU et les CPU et nous avons observé une accélération en utilisant cette arithmétique plutôt que l'arithmétique MP, en particulier pour l'addition.

Le chapitre 7 a servi à illustrer en pratique l'ensemble des niveaux que nous avons étudiés dans les chapitres précédents sur des calculs concrets de logarithme discret.

Nous avons réalisé deux implémentations de résolution de systèmes linéaires basées sur l'algorithme de Wiedemann par blocs

— une implémentation adaptée à un cluster de GPU NVIDIA, développée en CUDA et en PTX, avec l'arithmétique RNS.
— une implémentation adaptée à un cluster de CPU multi-cœurs (avec extensions SIMD), développée en C, avec l'arithmétique multi-précision et l'arithmétique RNS.

Ces implémentations ont contribué à la réalisation de records de calcul de logarithme discret dans les corps $\mathbb{F}_{2^{619}}$, $\mathbb{F}_{2^{809}}$ et $\mathbb{F}_{p_{180}}$. Notre implémentation GPU a été utilisée par Barbulescu et al. pour attaquer un corps fini de la forme $\mathbb{F}_{p^2}$ dont la taille est 529 bits [BGA+14].

Pistes de recherche futures

Utilisation des architectures *many-cœurs* (MIC). Dans ce travail, nous avons montré que les GPU sont adaptés au problème d'algèbre linéaire et que l'accélération fournie par cette architecture a permis de résoudre efficacement des résolutions de systèmes linéaires. Nous avons cependant vu que pour des tailles de systèmes très grandes, la taille de mémoire du GPU limite l'utilisation de ces architectures. Les CPU multi-cœurs peuvent disposer d'une mémoire plus grande, mais ont un nombre de cœurs très petit devant celui des GPU. Les coprocesseurs *many-cœurs* sont pourvus d'un nombre de cœurs plus grand que celui des CPU multi-cœurs et plus petit que celui des GPU et disposent d'une mémoire plus grande que celle des GPU. Ces architectures peuvent être des alternatives intéressantes, surtout lorsque la taille de la mémoire nécessaire devient problématique sur les GPU.

Réduction modulaire en RNS. Dans le chapitre 6, nous avons présenté l'algorithme de Bernstein qui permet de réduire en RNS un nombre x représenté en RNS modulo un nombre ℓ. La complexité de cet algorithme est de $n \times (n+1)$ multiplications RNS, avec n le nombre des composantes RNS pour représenter x. Cette complexité quadratique en n fait que l'opération de réduction est très coûteuse comparée aux autres opérations arithmétiques comme l'addition et la multiplication. Dans ce travail, nous amortissons le coût de cette opération en réduisant la fréquence des réductions modulaires. Dans un contexte plus général, il serait intéressant de diminuer la complexité de cette opération en cherchant un algorithme de complexité sous-quadratique dans l'intervalle de taille que nous considérons dans le contexte du logarithme discret.

Logarithme discret sur les corps finis de petite caractéristique. Nous avons vu dans le chapitre 1 que pour le cas des corps finis de petite caractéristique, il existe des algorithmes nouveaux dont la complexité est meilleure que celle des algorithmes en $L(\frac{1}{3})$. Pour ces algorithmes, l'algèbre linéaire ne consiste plus à résoudre un très grand système linéaire, mais plutôt plusieurs systèmes dont la taille est plus petite. Ainsi, dans le cas des corps finis de petite caractéristique, on n'aurait pas à résoudre des tailles aussi grandes que celles rencontrées avec des calculs faits avec l'algorithme FFS. Cette évolution change les caractéristiques du problème d'algèbre linéaire et il est pertinent de voir plus en détails comment s'adapter à ce nouveau contexte.

Bibliographie

[AD93] L. M. Adleman and J. DeMarrais. A Subexponential Algorithm for Discrete Logarithms over All Finite Fields. *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO '93*, pages 147–158, 1994. 14

[AH99] L. M. Adleman and M. A. Huang. Function Field Sieve Method for Discrete Logarithms over Finite Fields. *Inf. Comput.*, pages 5–16, May. 1999. 17

[AMO+14] G. Adj, A. Menezes, T. Oliveira et F. Rodríguez-Henríquez. Computing Discrete Logarithms in $F_{3^{6*137}}$ and $F_{3^{6*163}}$ Using Magma. WAIFI 2014 : 3-22. 22

[Adl79] L. M. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. *20th Annual Symposium on Foundations of Computer Science (FOCS '79)*, pages 55–60, 1979. 14

[Adl94] L. M. Adleman. The function field sieve. *Algorithmic Number Theory*, pages 108–121, 1994. 17

[BBB+12] R. Barbulescu, J. W. Bos, C. Bouvier, T. Kleinjung, and P. L. Montgomery. Finding ECM-friendly curves through a study of Galois properties. *ANTS-X 10th Algorithmic Number Theory Symposium - 2012*, 2012. 15

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods, 2nd Edition. SIAM, 1994, Philadelphia, PA. 68

[BBD+12] R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. The relationship between some guy and cryptography. *ECC2012 rump session talk (humoristic)*. http://ecc.2012.rump.cr.yp.to/, 2012. 116, 119

[BBD+13] R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. Discrete logarithm in GF(2^{809}) with FFS. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, April 2013. 22, 119

[BBD+14] R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. Discrete logarithm in GF(2^{809}) with FFS. *PKC 2014*, pages 221–238. Springer Berlin Heidelberg, 2014. 113, 119

[BDG10] B. Boyer, J-G. Dumas, and P. Giorgi. Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures. http://arxiv.org/abs/1004.3719, 2010. 77

[BDK98] J-C. Bajard, L-S. Didier, and P. Kornerup. An RNS Montgomery Modular Multiplication Algorithm. *IEEE TRANSACTIONS ON COMPUTERS*, pages 766–776. 1998. 96

[BDM98] J-C. Bajard, L-S. Didier, and J-M. Muller. A New Euclidean Division Algorithm for Residue Number Systems. *Journal of VLSI signal processing systems for signal, image and video technology*, pages 167–178. 1998. 92

[BF01] D. Boneh and M. Franklin. Identity-Based Encryption from the Weil Pairing. *Advances in Cryptology – CRYPTO 2001*. Lecture Notes in Computer Science 2139. Springer Berlin / Heidelberg, 2001, pages 213–229. 11

[BG08] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. 2008, NVIDIA Corporation, Technical Report, NVR-2008-004. 68, 73, 77

[BGA+14] R. Barbulescu, P. Gaudry, A. Guillevic, and F. Morain. Discrete logarithms in GF(p^2) — 160 digits. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, June 2014. 136

[BGG+14] R. Barbulescu, P. Gaudry, A. Guillevic, and F. Morain. Discrete logarithms in GF(p^2) — 160 digits. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, June 2014. 22

[BGG+15] R. Barbulescu, P. Gaudry, A. Guillevic, and F. Morain. Improving NFS for the discrete logarithm problem in non-prime finite fields. *Eurocrypt 2015, 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Lecture Notes in Computer Sciences, pp. 27, 2015. 21

[BGI+14] C. Bouvier, P. Gaudry, L. Imbert, H. Jeljeli, and E. Thomé. Discrete logarithms in GF(p) — 180 digits. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, June 2014. 22, 127

[BGJ+14] R. Barbulescu, P. Gaudry, A. Joux, E. Thomé. A Heuristic Quasi-Polynomial Algorithm for Discrete Logarithm in Finite Fields of Small Characteristic. *Advances in Cryptology – EUROCRYPT 2014*, pages 1-16, 2014. 17, 20

[BGY80] R. P. Brent, F. G. Gustavson, and D. Y. Yun. Fast solution of toeplitz systems of equations and computation of Padé approximants. *Journal of Algorithms*, pages 259 - 295, 1980. 55

[BHZ93] G.E. Blelloch, M.A. Heroux, and M. Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. School of Computer Science, Carnegie Mellon University, 1993. 73

[BL94] B. Beckerman and G. Labahn. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM. J. Matrix Anal. & Appl.*, 15(3), pages 804–823, 1994. 56

[BLS01] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security : Advances in Cryptology*, pages 514–532, 2001. 11

[BP04] J-C. Bajard and T. Plantard. RNS bases and conversions. *SPIE Annual Meeting 2004, Advenced Signal Processing Algorithms, Architectures, and Implementation XIV*, pages 60–69 2-6 August 2004. Denver, Colorado, USA. 95, 99

[BP14] R. Barbulescu and C. Pierrot. The multiple number field sieve for medium and high characteristic finite fields. *IACR Cryptology ePrint Archive*, 2014 :147, 2014. 18, 21

[Bar13] R. Barbulescu. Algorithms of discrete logarithm in finite fields. Ph.D Thesis from Université de Lorraine. https://tel.archives-ouvertes.fr/tel-00925228, Dec 2013. 20, 116, 120

[Bar13] R. Barbulescu. Selecting polynomials for the Function Field Sieve. *CoRR*, http://arxiv.org/abs/1303.1998, 2013. 20, 116, 120

[Ber68] E. R. Berlekamp. *Algebraic coding theory*, vol.15, no.1, McGraw-Hill, 1968. 55

[Ber95] D. J. Bernstein. Multidigit modular multiplication with the explicit chinese remainder theorem. Technical report. 1995. 97

[Bou13] C. Bouvier. The Filtering Step of Discrete Logarithm and Integer Factorization Algorithms. Preprint, http://hal.inria.fr/hal-00734654, 2013. 20, 48, 117, 120

[CADO] S. Bai, C. Bouvier, A. Filbois, P. Gaudry, L. Imbert, A. Kruppa, F. Morain, E. Thomé, and P. Zimmermann. CADO-NFS : Crible Algébrique : Distribution, Optimisation - Number Field Sieve. http://cado-nfs.gforge.inria.fr/. 55, 58, 69, 71, 127

[CUDAa] NVIDIA Corporation. CUDA Programming Guide Version 6.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide, August 2014. 27, 36, 37, 103

[CUDAb] NVIDIA Corporation. CUDA C BEST PRACTICES GUIDE. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, August 2014. 36

[CUSP] N. Bell and M. Garland. Cusp : Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://code.google.com/p/cusp-library/, 2012. 69, 73

[Can69] G. Cantor. Über einfache Zahlensysteme. *Zeitschrift für Math. und Physik*, 121–128, 1869. 94

[Cav02] S. Cavallar. On the number field sieve integer factorization algorithm. Ph.D. thesis, Universiteit Leiden, 2002. 48

[Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *Information Theory, IEEE Transactions on*, pages 587–594, Jul. 1984. 17

[Cop93] D. Coppersmith. Solving Linear Equations over GF(2) : Block Lanczos Algorithm. *Linear Algebra and its Applications*, Volume 192, Pages 33-60, ISSN 0024-3795, http://www.sciencedirect.com/science/article/pii/002437959390235G, October 1993. 55

[Cop94] D. Coppersmith. Solving Homogeneous Linear Equations over GF(2) via Block Wiedemann Algorithm. *Mathematics of Computation*, Volume 62, Pages 333–350, ISSN 0025-5718, http://dx.doi.org/10.2307/2153413, 1994. 56

[DBS04] R. Dutta, R. Barua, and P. Sarkar. Pairing-based cryptography : A survey. *Cryptology ePrint Archive*, Report 2004/064, 2004 http://eprint.iacr.org/. 11

[DGV13] J. Detrey, P. Gaudry, and M. Videau. Relation Collection for the Function Field Sieve. *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*, pages 201–210, April 2013. 19, 116, 120

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 1976, pp. 644-654. 8

[DMT74] B. Dickinson, M. Morf, and T. Kailath. A minimal realization algorithm for matrix sequences. *Automatic Control, IEEE Transactions on*, vol.19, no.1, pp.31–38, Feb 1974. 56

[Dek71] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, pages 224–242. 1971. 103

[Die11] C. Diem. On the discrete logarithm problem in elliptic curves. *Compositio Mathematica*, 147, pp 75-104, 2011. 22

[EK97] W. Eberly and E. Kaltofen. On Randomized Lanczos Algorithms. *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation - ISSAC '97*, pages 176–183, Kihei, Maui, Hawaii, USA, 1997. 54

[EKÇ13] E. Saule, K. Kaya, and Ü. V. Çatalyürek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. 2013, http://arxiv.org/abs/1302.1078. 68

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT–31(4) :469–472, July 1985. 10

[FFLAS] The FFLAS-FFPACK group. FFLAS-FFPACK : Finite Field Linear Algebra Subroutines / Package. v2.0.0, http://linalg.org/projects/fflas-ffpack, 2014. 69

[FM67] G. E. Forsythe and C. B. Moler. Computer Solution of Linear Algebraic Systems, 1967. 53

[FR94] G. Frey and H-G. Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of computation*, April 1994, pp. 865–874. 11

[Fermi] NVIDIA Corporation. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture : Fermi. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2009. 27

[Flo67] R. W. Floyd. Nondeterministic Algorithms. *Journal ACM*, Vol. 14 pages 636–644. Oct. 1967. 12

[Fly72] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, vol.C-21, no.9, pages 948–960 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5009071&isnumber=5009065, September 1972. 24

[GGM+13] F. Göloğlu, R. Granger, G. McGuire, and J. Zumbrägel. On the Function Field Sieve and the Impact of Higher Splitting Probabilities. *Advances in Cryptology – CRYPTO 2013*, pages 109–128, 2013. 17

[GKA+09] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing*, 2009, 36-77. 68, 86

[GKZ14] R. Granger, T. Kleinjung, and J. Zumbrägel. Discrete Logarithms in GF(2^{9234}). E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, January 2014. 22

[GM93] D. M. Gordon and K. S. McCurley. Massively parallel computation of discrete logarithms. *Advances in Cryptology — CRYPTO' 92*, pp 312–323, 1993. 19

[GMP] T. Granlund and the GMP development team. GNU MP : The GNU Multiple Precision Arithmetic Library. http://gmplib.org/. 87

[GPUDirect] GE Intelligent Platforms. GPUDirect™ RDMA. http://defense.ge-ip.com/library/detail/13493. 44

[GV14] P. Giorgi and B. Vialla. Generating Optimized Sparse Matrix Vector Product over Finite Fields. *Proc. ICMS 2014 : Fourth International Congress on Mathematical Software*, Seoul, Korea. LNCS Volume 8592, pp. 685–690, August 2014. 69, 87

[Gam91] D. Gamberger. New approach to integer division in residue number systems. *IEEE Symposium on Computer Arithmetic*, pages 84–91, IEEE, 1991. 92

[Gar59] H. L. Garner. The Residue Number System. *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, 146–153, 1959. 94

[Gor93] D. M. Gordon. Discrete logarithms in gf(p) using the number field sieve. *SIAM J. Discrete Math*, pages 124–138, 1993. 18

[HK95] M. A. Hitz and E. Kaltofen. Integer Division in Residue Number Systems. *IEEE Transactions on Computers*, 44, pages 983–989, 1995. 92

[HS52] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 409–436, 1952. 53

[Har13a] M. Harris. Using Shared Memory in CUDA C/C++. Post in *NVIDIA Developer Zone*, http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/, January 2013. 32

[Har13b] M. Harris. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. Post in *NVIDIA Developer Zone*, http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/, January 2013. 33

[JL02] A. Joux and R. Lercier. The Function Field Sieve Is Quite Special. *Algorithmic Number Theory*, pages 431–445, 2002. 17, 20

[JL03] A. Joux and R. Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the gaussian integer method. *Math. Comp.*, 72(242) (2003), pp 953–967. 18

[JL05] A. Joux and R. Lercier. Discrete logarithms in GF(2^{607}) and GF(2^{613}). E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, September 2005. 116

[JP13] A. Joux and C. Pierrot. The Special Number Field Sieve in $\mathbb{F}_{p^n}$, Application to Pairing-Friendly Constructions. *Cryptology ePrint Archive*, http://eprint.iacr.org/eprint-bin/cite.pl?entry=2013/582, 2013. 18, 21

[JP14] A. Joux and C. Pierrot. Discrete logarithm record in characteristic 3, GF($3^{5\times479}$) a 3796-bit field. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, September 2014. 22

[Jel14a] H. Jeljeli. Accelerating Iterative SpMV for Discrete Logarithm Problem Using GPUs. WAIFI 2014 : 25-44, 2014. 67, 89, 113

[Jel14b] H. Jeljeli. Resolution of Linear Algebra for the Discrete Logarithm Problem Using GPU and Multi-core Architectures. *Euro-Par 2014 Parallel Processing*, 764-775,http://dx.doi.org/10.1007/978-3-319-09873-9_64, 2014. 47, 59, 113

[Jou00] A. Joux. A one round protocol for tripartite Diffie-Hellman. *Proceedings of the 4th International Symposium on Algorithmic Number Theory*, pages 385–394, Springer-Verlag, 2000. 11

[Jou04] A. Joux. A One Round Protocol for Tripartite Diffie-Hellman. *Journal of Cryptology*, pages 263–276, 2004. 11

[Jou12] A. Joux. Discrete Logarithms in a 1175-bit Finite Field. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, December 2012. 22

[Jou13a] A. Joux. Faster Index Calculus for the Medium Prime Case Application to 1175-bit and 1425-bit Finite Fields. *Advances in Cryptology – EUROCRYPT 2013*, pages 177–193, 2013. 17, 120

[Jou13b] A. Joux. A New Index Calculus Algorithm with Complexity L(1/4+o(1)) in Small Characteristic. *Selected Areas in Cryptography – SAC 2013*, pages 355–379, 2014. 17

[Jou13c] A. Joux. Discrete logarithms in GF(2^{6168}) [=GF($(2^{257})^{24}$)]. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, May 2013. 22

[Jou13d] A. Joux. Discrete Logarithms in a 1425-bit Finite Field. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, January 2013. 22

[KPH04] H. Kim, H-J. Park, and S-H. Hwang. Parallel Modular Multiplication Algorithm in Residue Number System. *Parallel Processing and Applied Mathematics*, pages 1028–1033. 2004. 96

[Kepler] NVIDIA Corporation. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture : Fermi. http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf, 2012. 27

[Kle07] T. Kleinjung. Discrete logarithms in GF(p) — 160 digits. E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, February 2007. 22, 127

[Kle08] T. Kleinjung. Polynomial Selection. *CADO workshop on integer factorization*, Nancy, France. http://cado.gforge.inria.fr/workshop/abstracts.html, 2008. 127

[Kle14] T. Kleinjung. Discrete Logarithms in GF(2^{1279}). E-mail to the NMBRTHRY mailing list ; http://listserv.nodak.edu/archives/nmbrthry.html, October 2014. 22, 127

[Knu70] D. E. Knuth. The analysis of algorithms. *Proc. International Congress of Mathematicians*, pages 269-274, 1970. 55

[Kra13a] J. Kraus. An Introduction to CUDA-Aware MPI. Post in *NVIDIA Developer Zone*, http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/, March 2013. 42

[Kra13b] J. Kraus. Benchmarking CUDA-Aware MPI. Post in *NVIDIA Developer Zone*, http://devblogs.nvidia.com/parallelforall/benchmarking-cuda-aware-mpi/, March 2013. 42

[LC92] M. Lu and J. S. Chiang. A Novel Division Algorithm for the Residue Number System. *IEEE Trans. Comput.*, pages 1026–1032, August 1992. 92

[LHK+79] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, Vol. 5, No. 3, pages 308–323, September 1979. 68

[LO90] B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. *Advances in Cryptology - CRYPTO '90*, volume 537, pages 109–133, Santa Barbara, CA, USA, August 11-15, 1990. 48

[LR11] J. Luitjens and S. Rennich. CUDA Warps and Occupancy. *GPU Computing Webinar*, http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf, 2011. 37

[LSC+13] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing - ICS '13*, 2013, 273–282. 68

[Lan52] C. Lanczos. Solution of Systems of Linear Equations by Minimized Iterations. *Journal of Research of the National Bureau of Standards*, 33–53, 1952. 53, 54

[Len87] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, pages 649–673, 1987. 15

[LinBox] Project LinBox : Exact computational linear algebra. http://www.linalg.org/. 69

[MOV93] A. J. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, September 1993, pp. 163–1646. 11

[MPFQ] The MPFQ development team. MPFQ - A finite field library. http://mpfq.gforge.inria.fr/. 114

[MSK02] S. Mitsunari, R. Sakai, and M. Kasahara. A new traitor tracing. *IEICE Trans. Fundamentals*. Vol. E58-A, No. 2, pp. 481-484, 2002.. 11

[MUMPSa] P. R. Amestoy and I. S. Duff and J. Koster and J.-Y. L'Excellent. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, Vol. 23, No. 1, pages 15–41, 2001. 69

[MUMPSb] P. R. Amestoy and A. Guermouche and J.-Y. L'Excellent and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, Vol. 32, No. 2, pages 136–156, 2006. 69

[MW96] U. M. Maurer and S. Wolf. Diffie-Hellman Oracles. *Advances in Cryptology - CRYPTO 96*, 1996, pp. 268–282. 9

[Mas69] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, vol.15, no.1, pp.122–127, http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1054260&isnumber=22643, January 1969. 55

[Mau94] U. M. Maurer. Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms. *Advances in Cryptology - CRYPTO 94*, 1994, pp. 271-281. http://dx.doi.org/10.1007/3-540-48658-5_26 9

[Mer64] R. D. Merrill. Improving digital computer performance using residue number theory. *IEEE Transactions on Electronic Computers*, EC-13(2) :93–101, April 1964. 100

[Mon95] P. L. Montgomery. A Block Lanczos Algorithm for Finding Dependencies over GF(2). *Advances in Cryptology — EUROCRYPT '95*, Pages 106-120, http://dx.doi.org/10.1007/3-540-49264-X_9, 1995. 55

[NBG+08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. Queue, 40–53, 2008. 73

[Nec94] V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2) :165–172, 1994. 11

[OpenCL] Khronos OpenCL Working Group. The OpenCL Specification Version 2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, October 2014. 27

[PH78] S. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, IT–24 :106–110, 1978. 13

[PHV+13] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPU. *42nd International Conference on Parallel Processing (ICPP)*, pages 80–89, http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6687341, 2013. 44

[PP92] K. C. Posch and R. Posch. Modulo Reduction in Residue Number Systems. *IIG-Report-Series : Institute für Informationsverarbeitung*. 1992. 97

[PS92] C. Pomerance and J. W. Smith. Reduction of Huge, Sparse Matrices over Finite Fields via Created Catastrophes. *Experimental Mathematics*, 89–94, 1992. 48

[PTV+92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes in C : The Art of Scientific Computing, 1992. 53

[PTXa] NVIDIA Corporation. PTX : Parallel Thread Execution ISA Version 4.1. http://docs.nvidia.com/cuda/pdf/ptx_isa_4.1.pdf. August 2014. 36

[PTXb] NVIDIA Corporation. Inline PTX Assembly in CUDA. http://docs.nvidia.com/cuda/pdf/Inline_PTX_Assembly.pdf. August 2014. 36

[Pol75] J. M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, pages 331–334. 1975. 12

[Pol78] J. M. Pollard. Monte Carlo methods for Index Computation (mod p). *Mathematics of Computation*, Vol. 32 pages 918–924. 1978. 12

[Pol93] J. M. Pollard. The lattice sieve. *The development of the number field sieve*, pages 43–49, 1993. 19

[Pom82] C. Pomerance. Analysis and comparison of some integer factoring algorithms. *Mathematisch Centrum Computational Methods in Number Theory*, Part 1, pp. 89-139, 1982, http://www.math.dartmouth.edu/~carlp/PDF/analysiscomparison.pdf. 14

[Pom84] C. Pomerance. The quadratic sieve factoring algorithm. *Advances in Cryptology, Proceedings of Eurocrypt 84*, pages 169–182, Paris, 1984. 15

[RHJ13] R. Rabenseifner, G. Hager, and G. Jost. Hybrid Parallel Programming Hybrid MPI and OpenMP Parallel Programming. Tutorial at *SC13*, http://openmp.org/sc13/HybridPP_Slides.pdf, November 2013. 39

[SAD11] B. Schmidt, H. Aribowo, and H-V. Dang. Iterative Sparse Matrix-Vector Multiplication for Integer Factorization on GPUs. *Euro-Par 2011 Parallel Processing*, 413-424, 2011. 71, 74, 75, 77, 80

[SHZ+07] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan Primitives for GPU Computing. Graphics Hardware 2007, 2007. 73

[SK10] J. Sanders and E. Kandrot. CUDA by Example : An Introduction to General-Purpose GPU Programming. 2010. 27

[SK89] P. P. Shenoy and R. Kumaresan. Fast Base Extension Using a Redundant Modulus in RNS. *IEEE Transactions on Computers*, pages 292–297. February 1989. 96, 97, 98

[SOK00] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems based on pairings. *SICS. Symposium on cryptography and information security*. 2000. 11

[SWD96] O. Schirokauer, D. Weber, and T. Denny. Discrete logarithms : The effectiveness of the index calculus method. *Algorithmic Number Theory*, Springer Berlin Heidelberg, 1996, 1122, pp 337–361. 18

[Sch00] O. Schirokauer. Using number fields to compute logarithms in finite fields. *Math. Comp*, pages 1267–1283, 2000. 18

[Sch11] T. C. Schroeder. Peer-to-Peer & Unified Virtual Addressing. *CUDA Webinar*, http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GPUDirect_uva.pdf, 2011. 41

[Sch93] O. Schirokauer. Discrete logarithms and local units. *Philosophical Transactions of the Royal Society of London A : Mathematical, Physical and Engineering Sciences*, pp 409–423, 1993. 19

[Sem02] I. A. Semaev. Special Prime Numbers and Discrete Logs in Finite Prime Fields. *Math. Comput.*, pages 363–377, January 2002. 18, 21

[Sha71] D. Shanks. Class number, a theory of factorization, and genera. *In 1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y., 1969)*, pages 415–440. Providence, R.I., 1971. 11

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. *Advances in Cryptology – EUROCRYPT '97*, volume 1233, pages 256–266, May 1997. 11

[Sou07] L. Sousa. Efficient method for magnitude comparison in rns based on two pairs of conjugate moduli. *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, 2007. 92

[Str69] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 354–356, 1969. 53

[Tay84] F. J. Taylor. Residue Arithmetic A Tutorial with Examples. *Computer*, 50–62, IEEE Computer Society Press, Los Alamitos, CA, USA, May 1984. 91

[Tho02] E. Thomé. Subquadratic Computation of Vector Generating Polynomials and Improvement of the Block Wiedemann Algorithm. *Journal of Symbolic Computation*, 33(5) :757–775, 2002. 55, 56

[Tho03] E. Thomé. Algorithmes de Calcul de Logarithmes Discrets dans les Corps Finis. Ph.D. Thesis, http://tel.archives-ouvertes.fr/tel-00007532, May 2003. 56

[Tho12a] E. Thomé. Block Wiedemann implementation in CADO-NFS. Exposé dans le séminaire *Efficient Linear Algebra for Gröbner Basis Computations*, http://wiki.lmona.de/events/elagb/schedule?action=AttachFile&do=view&target=thome.pdf, June 2012. 63

[Tho12b] E. Thomé. Théorie algorithmique des nombres et applications à la cryptanalyse de primitives cryptographiques. Habilitation Thesis from Université de Lorraine. https://tel.archives-ouvertes.fr/tel-00765982, Dec 2012. 63, 123, 129

[VGM+09] F. Vázquez, E. M. Garzón, J. A. Martinez, and J. J. Fernández. The sparse matrix vector product on GPU. 2009, Technical report. 68, 71, 77

[Vil97] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. LMC-IMAG, Grenoble, France, Apr. 1997. 56

[Vol10] V. Volkov. Better Performance at Lower Occupancy. *GPU Technology Conference 2010 (GTC 2010)*, http://on-demand.gputechconf.com/gtc/2010/presentations/S12238-Better-Performance-at-Lower-Occupancy.pdf, 2010. 36, 37

[WOL+07] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing - SC '07*, 2007, 38 :1–38 :12. 68, 86

[Wes68] J. R. Westlake. A Handbook of Numerical Matrix Inversion and Solution of Linear Equations, 1968. 53

[Wie86] D. H. Wiedemann. Solving Sparse Linear Equations over Finite Fields. *IEEE Transactions on Information Theory*, 54–62, January 1986. 53, 55

Résumé

Les primitives de la cryptographie à clé publique reposent sur la difficulté supposée de résoudre certains problèmes mathématiques. Dans ce travail, on s'intéresse à la cryptanalyse du problème du logarithme discret dans les sous-groupes multiplicatifs des corps finis. Les algorithmes de calcul d'index, utilisés dans ce contexte, nécessitent de résoudre de grands systèmes linéaires creux définis sur des corps finis de grande caractéristique. Cette algèbre linéaire représente dans beaucoup de cas le goulot d'étranglement qui empêche de cibler des tailles de corps plus grandes.

L'objectif de cette thèse est d'explorer les éléments qui permettent d'accélérer cette algèbre linéaire sur des architectures pensées pour le calcul parallèle. On est amené à exploiter le parallélisme qui intervient dans différents niveaux algorithmiques et arithmétiques et à adapter les algorithmes classiques aux caractéristiques des architectures utilisées et aux spécificités du problème. Dans la première partie du manuscrit, on présente un rappel sur le contexte du logarithme discret et des architectures logicielles et matérielles utilisées. La seconde partie du manuscrit est consacrée à l'accélération de l'algèbre linéaire.

Ce travail a donné lieu à deux implémentations de résolution de systèmes linéaires basées sur l'algorithme de Wiedemann par blocs : une implémentation adaptée à un cluster de GPU NVIDIA et une implémentation adaptée à un cluster de CPU multi-cœurs. Ces implémentations ont contribué à la réalisation de records de calcul de logarithme discret dans les corps binaires $\mathbb{F}_{2^{619}}$ et $\mathbb{F}_{2^{809}}$ et dans le corps premier $\mathbb{F}_{p_{180}}$.

Abstract

The security of public-key cryptographic primitives relies on the computational difficulty of solving some mathematical problems. In this work, we are interested in the cryptanalysis of the discrete logarithm problem over the multiplicative subgroups of finite fields. The index calculus algorithms, which are used in this context, require solving large sparse systems of linear equations over finite fields. This linear algebra represents a serious limiting factor when targeting larger fields.

The object of this thesis is to explore all the elements that accelerate this linear algebra over parallel architectures. We need to exploit the different levels of parallelism provided by these computations and to adapt the state-of-the-art algorithms to the characteristics of the considered architectures and to the specificities of the problem. In the first part of the manuscript, we present an overview of the discrete logarithm context and an overview of the considered software and hardware architectures. The second part deals with accelerating the linear algebra.

We developed two implementations of linear system solvers based on the block Wiedemann algorithm : an NVIDIA-GPU-based implementation and an implementation adapted to a cluster of multi-core CPU. These implementations contributed to solving the discrete logarithm problem in binary fields $\mathbb{F}_{2^{619}}$ et $\mathbb{F}_{2^{809}}$ and in the prime field $\mathbb{F}_{p_{180}}$.