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).

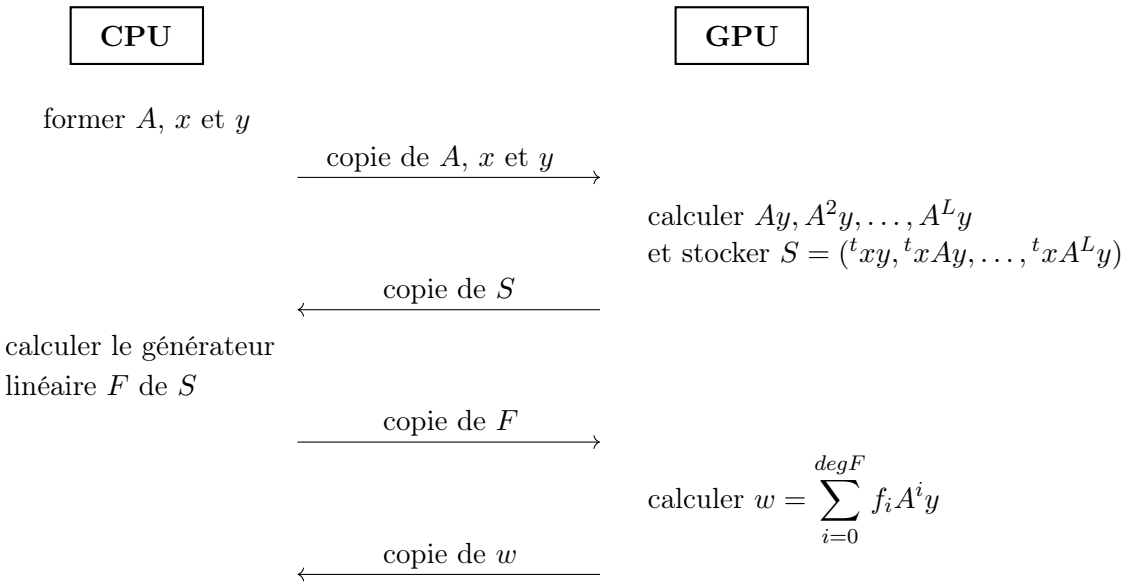

FIGURE 7.1 – Organisation du calcul de Wiedemann simple qui est exécuté sur un GPU et un CPU.

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 :
   $$f(x,t) = x^6 + \texttt{0x7}x^5 + \texttt{0x6b}x^3 + \texttt{0x1ab}x^2 + \texttt{0x326}x + \texttt{0x19b3}$$
   $$g(x,t) = x + \texttt{0x800000000000000000000000000001e7eaa}$$
   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.

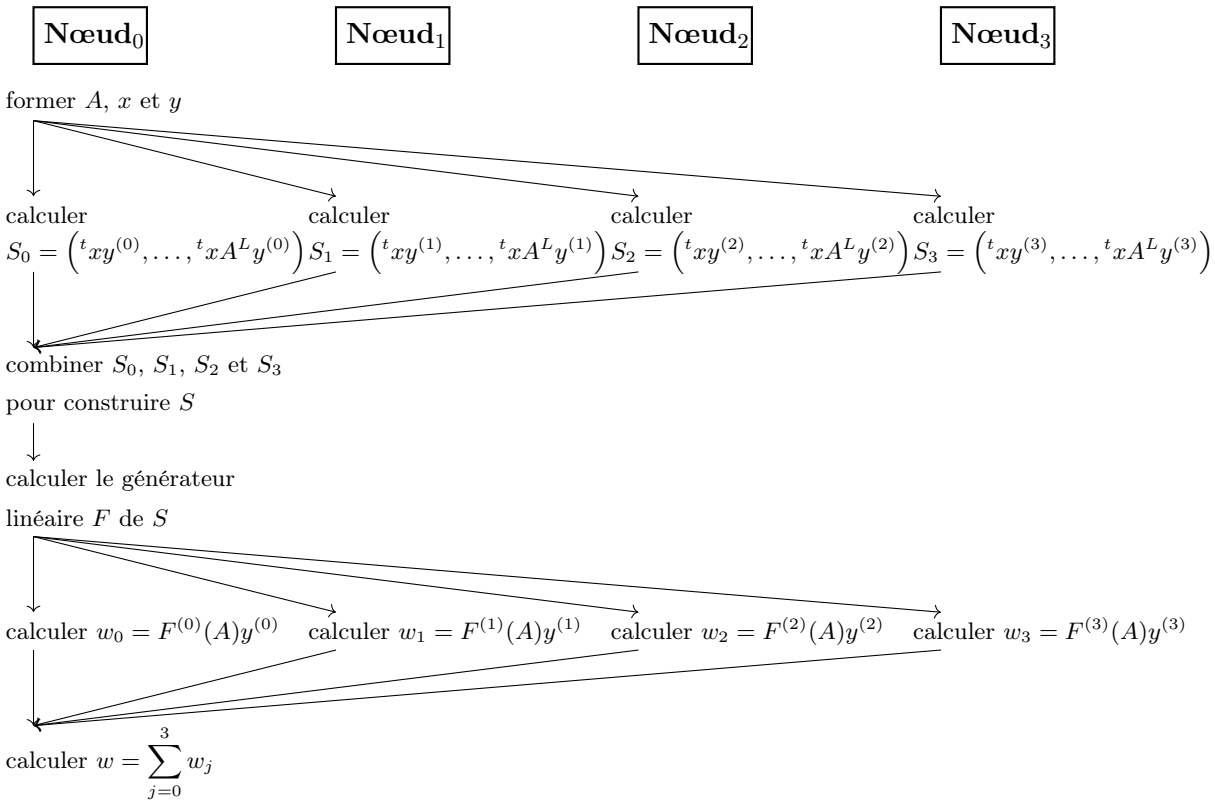

FIGURE 7.2 – Organisation du calcul de Wiedemann par blocs avec la configuration ($n = 4, m = 8$) et exécuté sur 4 nœuds.

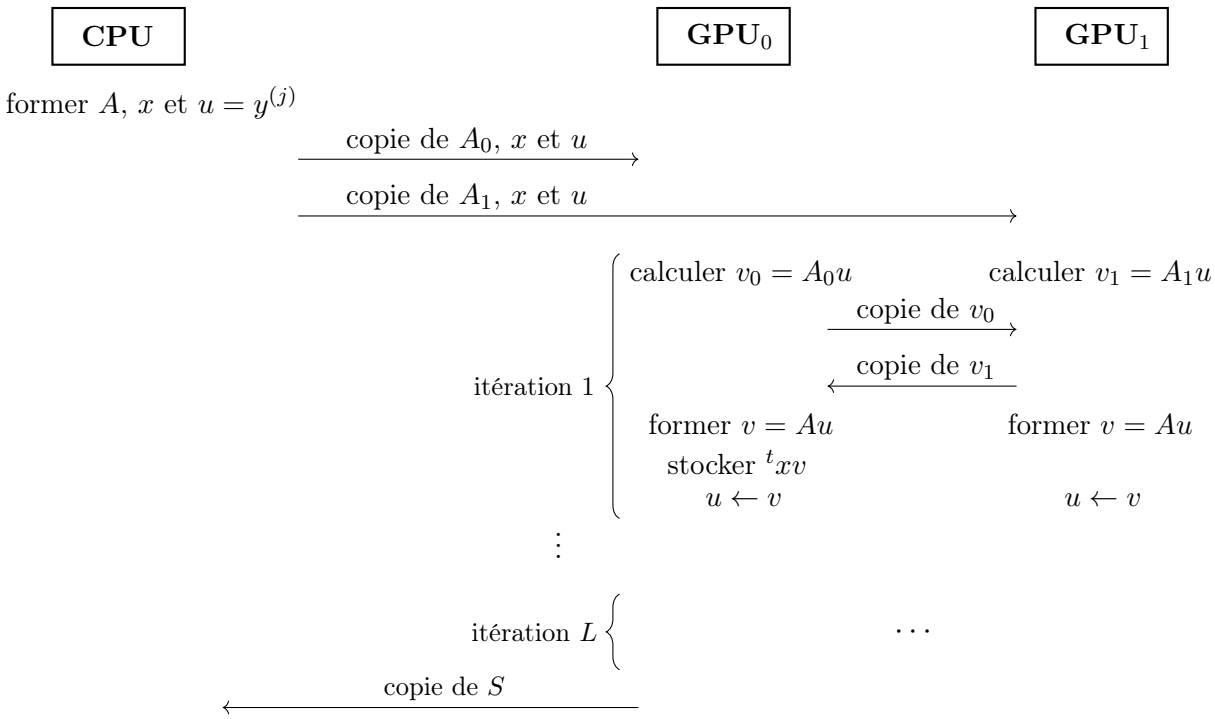

FIGURE 7.3 – Organisation du calcul de l'étape *Produits scalaires* sur un nœud d'indice j contenant 1 CPU et 2 GPU.

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.

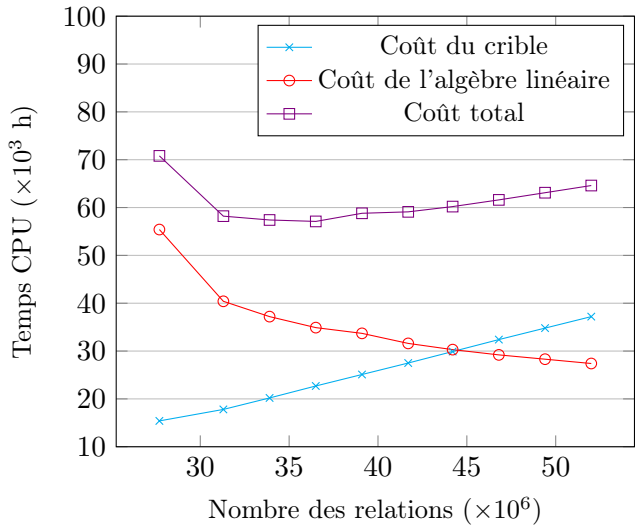

FIGURE 7.4 – Coûts du crible et de l'algèbre linéaire et coût total pour différents nombres de relations.

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 :

$$\begin{aligned}\log 2 \equiv\ & 14394742424980404689468652122583501155340452982569859689893949 95 \backslash \\ & 37509189519718986652049683275189725501776470006513329773475176 6 \backslash \\ & 54387676076061308411099885253085259407173106476434760 8 \pmod{\ell}.\end{aligned}$$

$$\begin{aligned}\log 3 \equiv\ & 12540255374709186945948836756152071692814462540757959805173613 9 \backslash \\ & 49252707487386035786690693592163692301618098936460400547559095 2 \backslash \\ & 63524577946074538124684456888597268322428333393912658 4 \pmod{\ell}.\end{aligned}$$

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 $(n = 12, m = 24)$.

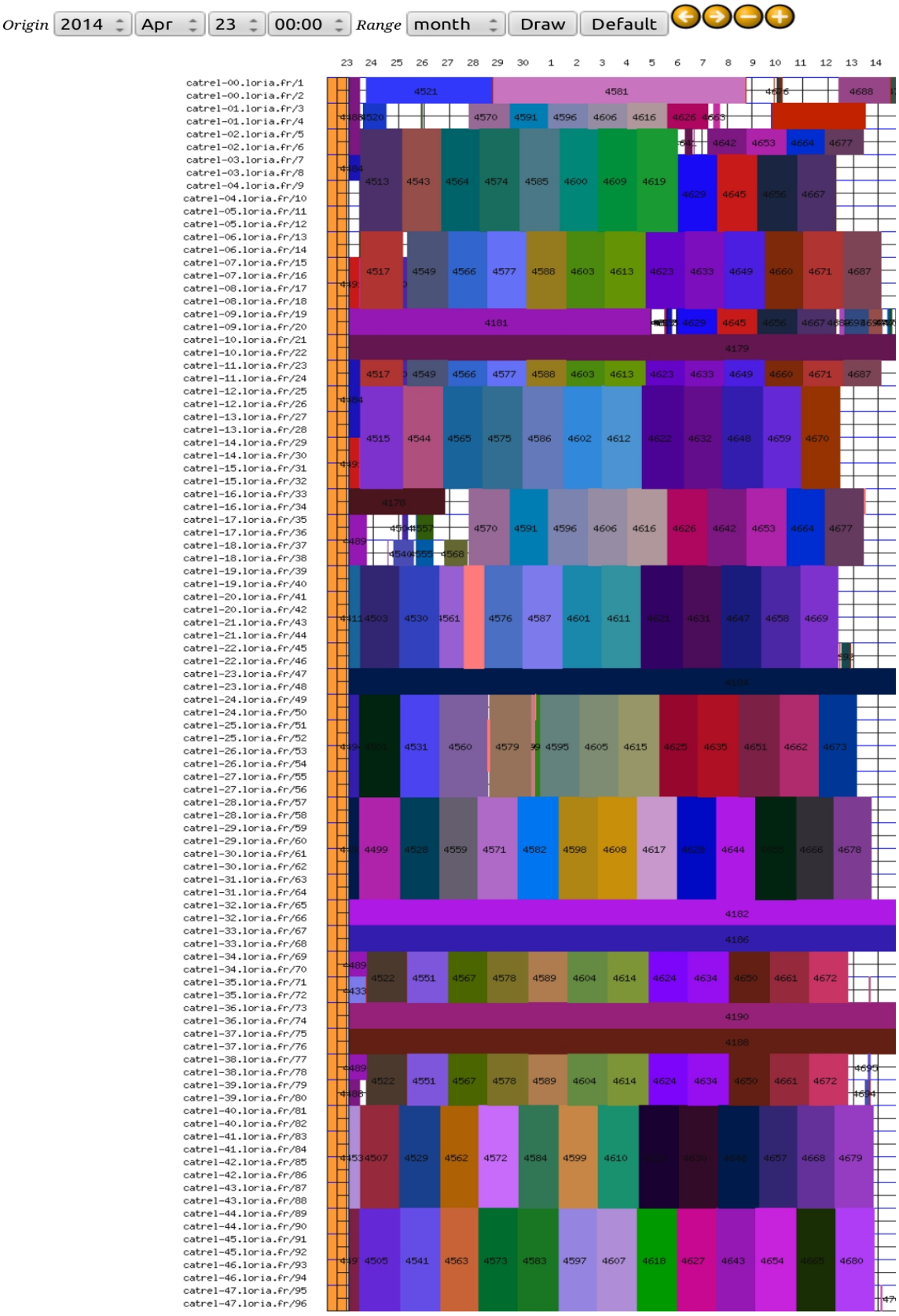

FIGURE 7.5 – Diagramme de Gantt de l'utilisation du cluster *catrel* pour l'algèbre linéaire du record dans $\mathbb{F}_{p_{180}}$.

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.