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 $\mathbb{F}_{p_{155}}$.

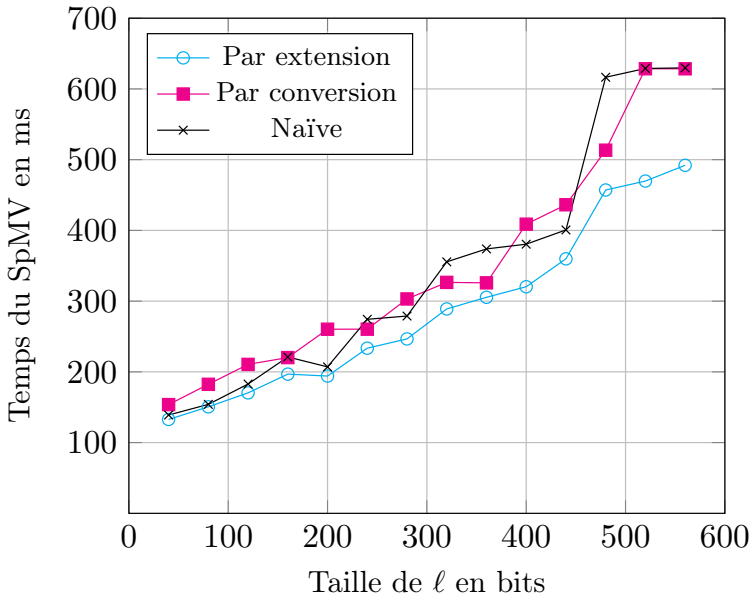

FIGURE 6.1 – Temps du SpMV avec les méthodes naïve, par conversion et par extension en fonction de la taille de ℓ.

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.