Produit matrice-vecteur

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

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

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

Sommaire

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

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

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

5.1 Travaux et bibliothèques pour l'algèbre linéaire

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

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

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

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

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

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

5.2 Formats de représentation de la matrice creuse

Notations

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

5.2.1 Coordinate (COO)

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

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

(a) matrice creuse A

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

(b) sa représentation COO

5.2.2 Compressed Sparse Row (CSR)

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

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

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

(a) matrice creuse A

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

(b) sa représentation CSR

5.2.3 ELLPACK (ELL)

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

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

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

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

(a) matrice creuse A

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

(b) sa représentation ELL-R

5.2.4 Sliced Coordinate (SLCOO)

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

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

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

(a) matrice creuse A

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

(b) sa représentation SLCOO-2

5.2.5 Diagonal (DIA)

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

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

(a) matrice creuse A

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

(b) sa représentation DIA

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

5.3 Produits matrice-vecteur sur GPU

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

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

Notations

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

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

5.3.1 COO

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

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

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

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

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

5.3.2 CSR

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

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

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

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

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

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

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

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

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

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

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

5.3.3 ELL

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

5.3.4 SLCOO

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

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

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

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

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

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

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

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

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

5.3.5 Exécution des SpMV sur une matrice-jouet

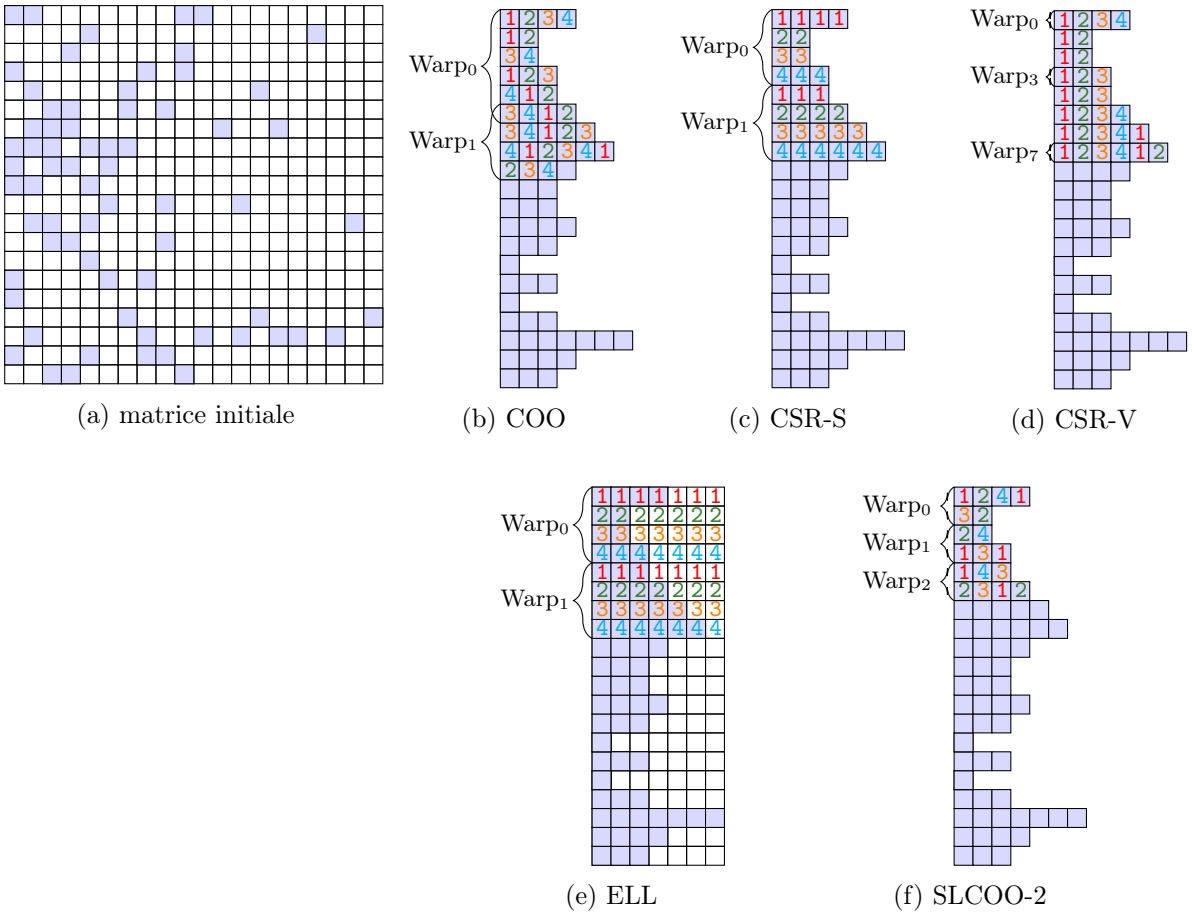
Dans la figure 5.6, nous prenons une petite matrice creuse composée de 20 lignes et colonnes et nous allons illustrer l'exécution des SpMV précédemment décrits sur cette matrice. Par souci de simplicité, on suppose qu'un *warp* contient 4 *threads* ; chaque *thread* est représenté par un nombre parmi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

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

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

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

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



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

(e) ELL (f) SLCOO-2

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

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

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

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

Notations

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

5.4.1 Schéma *séquentiel*

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

5.4.2 Schéma *parallèle*

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

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

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

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

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

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

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

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

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

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

5.5 Analyse comparative des produits matrice-vecteur sur GPU

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

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

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

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

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

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

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

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

5.5.1 Comparaison des schémas *séquentiel* et *parallèle*

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

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

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

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

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

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

5.5.2 Comparaison des SpMV CSR, COO et ELL

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

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

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

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

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

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

5.5.3 Comparaison des SpMV SLCOO et CSR-*vectoriel*

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

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

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

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

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

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

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

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

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

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

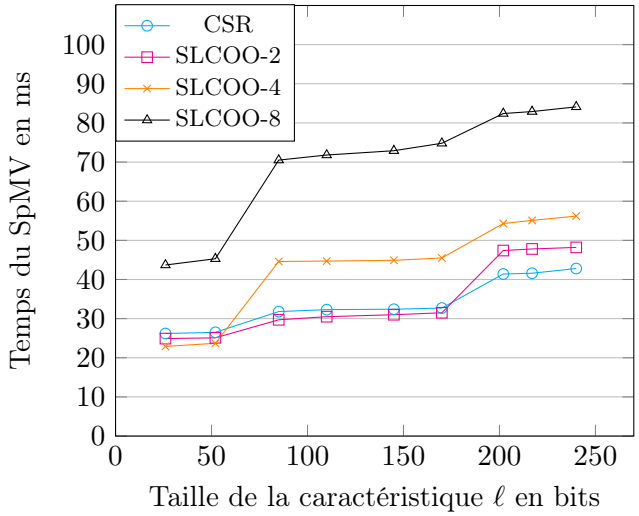


FIGURE 5.7 – Performances des SpMV CSR et SLCOO-σ en fonction de la taille de la caractéristique, avec la matrice FFS pour $\mathbb{F}_{2^{619}}$.

5.6 Améliorations pour le produit CSR-V

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

5.6.1 Cache texture

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

5.6.2 Réordonner les coefficients non nuls d'une ligne

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

5.6.3 Compresser le tableau de valeurs `data`

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

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

5.6.4 Améliorer l'équilibre des warps

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

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

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

5.7 Produits matrice-vecteur sur CPU

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

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

5.7.1 SpMV COO, CSR et ELL

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

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

Algorithm 5.7 SpMV COO

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

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

Algorithm 5.8 SpMV CSR

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

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

Algorithm 5.9 SpMV ELL-R

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

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

5.7.2 SpMV BCSR

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

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

(a) matrice creuse A

$$\texttt{data} = \begin{bmatrix} a_{00} & a_{01} & 0 & a_{11} & 0 & a_{03} & a_{12} & a_{13} & a_{20} & 0 & 0 & a_{31} & \dots \end{bmatrix}$$
$$\texttt{id} = \begin{bmatrix} 0 & 2 & 0 & \dots \end{bmatrix}$$
$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & \dots \end{bmatrix}$$

(b) sa représentation BCSR-2x2

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

Algorithm 5.10 SpMV BCSR-2x2

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

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

5.7.3 Comparaison des SpMV COO, CSR, ELL et BCSR

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

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

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

5.7.4 Optimisations pour le produit CSR

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

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

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

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

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

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

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

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

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

5.8 Comparaison avec des bibliothèques existantes

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

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

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

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

5.9 Conclusion

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