Eléments logiciels pour le traitement des données massives

Édith Darin, Gauthier Schweitzer

5 février 2018

1 Introduction

Pour ce projet, nous avons choisi de mettre en oeuvre une Analyse en Composante Principale non linéaire en ayant le plus possible recours au GPU. Pour cela, nous nous appuyons sur l'algorithme "Gram-Schmidt orthogonalization" proposé par Andrecut, 2008 dans son papier "Parallel GPU Implementation of Iterative PCA Algorithms" [1].

Cet algorithme s'appuie initialement sur la libraire CUBLAS, développée par NVIDIA. Afin de travailler à une granularité plus fine, nous avons choisi de ne pas utiliser ces fonctions CUBLAS mais de coder nous-mêmes des fonctions en utilisant PyCuda et CUDA 9.1

Nous allons d'abord brièvement présenter le fonctionnement de l'algorithme, puis introduire les différentes fonctions que nous avons écrites en CUDA, avant de suggérer une implémentation.

2 Algorithme d'orthogonalisation de Gram-Schmidt

2.1 Avantages

Alors que l'algorithme NIPALS-PCA souffre de problèmes d'orthogonalité dès lors que l'on va au-delà des premières composantes principales, les algorithmes basés sur la réorthogonalisation de Gram-Schmidt permettent la stabilité de l'orthogonalité. Parmi ces algorithmes, certains comme le Classical Gram Schmidt (CGS) souffrent d'instabilité numérique, d'autres comme le MGS ne peuvent être exprimés avec des fonctions suffisamment simples pour bénéficier d'une parallélisation (opérations matrice-vecteurs). Le CGS, s'il est mis en place de

1. https://arxiv.org/pdf/0811.1081.pdf

manière itérative (ré-orthogonalisation des scores et des loadings à chaque itération, au prix toutefois d'un effort computationnel) combine les avantages d'être implémentable par des opérations matrice-vecteurs et de bénéficier de la stabilité numérique grâce à l'aspect itératif.

2.2 FORMALISATION

Nous allons présenter ici succinctement la formalisation de cet algorithme afin de pouvoir s'y référer lors du code. On a une matrice X en entrée que l'on peut décomposer ainsi :

$$X = T_{(K)} P_{(K)}^T + R \tag{2.1}$$

avec $T_{(K)}$ la matrice regroupant les composantes principales en colonnes, $P_{(K)}$ les *loads*, c'est à dire en quelque sorte les coordonnées des variables dans l'espace des composantes. R étant donc la matrice des résidus.
L'algorithme proposé par Andrecut est décrit dans le pseudo-code suivant
La logique de cet algorithme est d'aboutir à une PCA par une succession d'opérations simples sur les matrices et vecteurs, à même d'être parallélisées. Nous avons donc poursuivi cette logique en réécrivant des fonctions du plus bas niveau possible. Nous avons, dans chaque cas, cherché et réussi à écrire les opérations suivantes en PyCuda :

- Calcul du produit d'une matrice et d'un vecteur
- Addition d'un vecteur à une matrice, multiplié par un scalaire
- Multiplication d'un vecteur par un scalaire et ajoute un vecteur
- Calcul de la norme euclidienne d'un vecteur
- Transposition d'une matrice

Algorithm 1 Algorithm Gram-Schmidt PCA

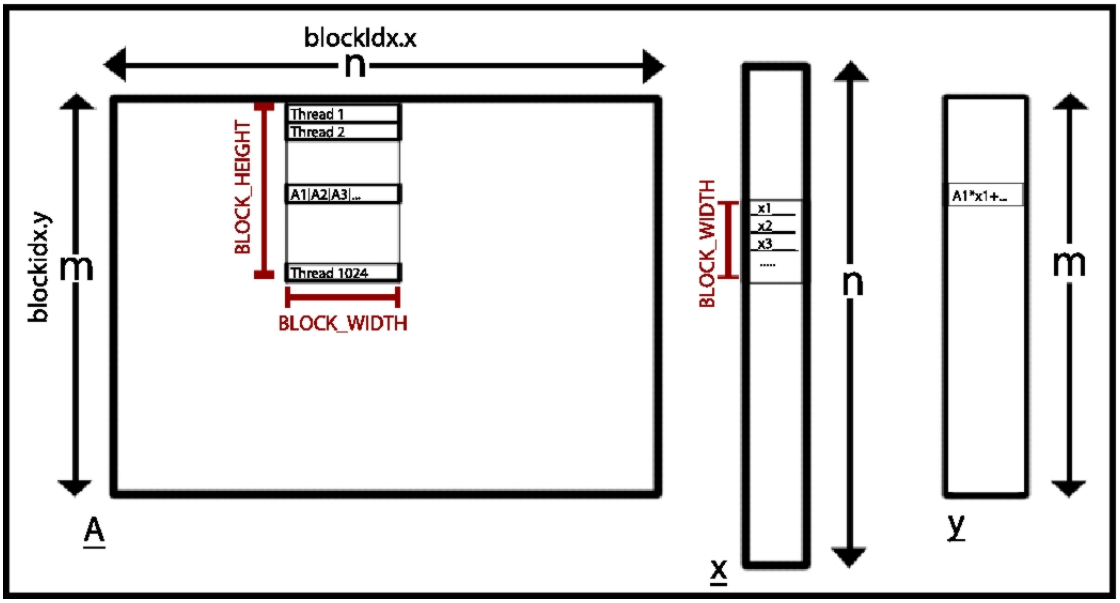
$R \leftarrow X$
for $(k = 0, \cdots, K-1)$ **do**
    {
    $\mu = 0$
    $V^{(k)} \leftarrow R^{(k)}$ **(1)**
    for $(j = 0, \cdots, J)$ **do**
        {
        $U^{(k)} \leftarrow R^T V^{(k)}$
        if $k > 0$ **then**
            {
            $A \leftarrow U_{(K)}^T U^{(k)}$ **(2)**
            $U^{(k)} \leftarrow U^{(k)} - U_{(k)} A$
            }
        $U^{(k)} \leftarrow U^{(k)} ||U^{(k)}||^{-1}$ **(3)**
        $V^{(k)} \leftarrow R U^{(k)}$
        if $k > 0$ **then**
            {
            $B \leftarrow V_{(K)}^T V^{(k)}$
            $V^{(k)} \leftarrow V^{(k)} - V_{(k)} B$
            }
        $\lambda_k \leftarrow ||V^{(K)}||$ **(4)**
        $V^{(K)} \leftarrow V^{(K)} / \lambda_k$
        if $|\lambda_k - \mu| \leq \epsilon$ **then**
            break
        $\mu \leftarrow \lambda_k$
        }
    $R \leftarrow R - \lambda_k V^{(K)} (U^{(K)})^T$
    }
$T \leftarrow V\Lambda$ **(5)**
$P \leftarrow U$
return T, P, R

3 Présentation de nos fonctions en PyCuda

3.1 MatrixVect : multiplication d'une matrice par un vecteur

3.1.1 Présentation de l'algorithme

Il s'agit de la fonction la plus complexe que nous ayons traitée, du fait des multiples choix possibles concernant la forme des blocs, le rôle des threads et la structure des grids. Nous avons choisi de mettre en place un algorithme du type "division en blocs puis multiplications" (par opposition aux algorithmes ayant recours à des transpositions). On considère ainsi en entrée une matrice a de taille $N\_ROW * N\_COL$ et un vecteur in de taille N_COL. Pour procéder à la multiplication, on découpe la matrice en blocs de hauteur $BLOCK\_HEIGHT$ et de largeur $BLOCK\_WIDTH$. L'output out sera donc un vecteur de taille N_ROW.



Ainsi, pour chaque ligne d'un bloc considéré de la matrice a, on calcule à l'aide d'un unique thread son produit scalaire avec les éléments du vecteur à multiplier in correspondant et on ajoute le résultat à la coordonnée adaptée du vecteur de sortie out. Puisque pour chaque ligne d'un même bloc, le produit scalaire se fera avec les mêmes éléments du vecteur in, on stocke ces éléments (au nombre de $BLOCK\_HEIGHT$) dans la mémoire partagée du bloc. Pour chaque bloc, on crée également trois variables en mémoire partagée pour éviter d'avoir à reproduire $BLOCK\_HEIGHT$ fois les mêmes calculs pour chaque thread.

— *blockElt* : Cette variable correspond au nombre d'éléments sur lesquels le thread calcule le produit scalaire. Elle est généralement égale à $BLOCK\_WIDTH$ sauf pour les blocs les plus à droite de la matrice pour lesquels elle est égale à $N\_COL\%BLOCK\_WIDTH$ (avec % indiquant l'opération modulo) ;

— *blockxInd* (= $blockIdx.xBLOCK\_WIDTH$) et *blockyInd* (= $blockIdx.yBLOCK\_HEIGTH$) qui correspondent respectivement à la première et la deuxième coordonnée du premier élément du bloc dans la matrice.

Dans chaque bloc, deux threads consécutifs s'occupent de deux lignes consécutives. Puisque la matrice en entrée est ordonnée selon les colonnes, l'indexation de la matrice aplatie passe d'abord par la première colonne et la mémoire est utilisée en "coalescing".
On s'assure également que les sommes sur le vecteur d'output out sont bien atomiques afin d'éviter les problèmes d'écriture concurrente des blocs qui ont la même *blockIdx.y*. Après avoir parcouru ainsi l'ensemble des blocs, on a bien $a * in = out$.

La valeur de $BLOCK\_HEIGHT$ est hardware-dépendante. Puisque l'on va utiliser un thread par ligne dans l'algorithme et que l'on souhaite utiliser le nombre maximum de threads possibles dans un bloc, on fixe $BLOCK\_HEIGHT = MAX\_THREADS\_PER\_BLOCK$, valeur récupérée dans les device attributes et égale à 1024 sur notre GPU (Nvidia GTX 960 achetée pour l'occasion).
La valeur de $BLOCK\_WIDTH$ est également hardware-dépendante. Chaque thread va calculer le produit scalaire de deux vecteurs de taille $BLOCK\_WIDTH$ avant d'ajouter le résultat à la bonne coordonnée de out. Plus $BLOCK\_WIDTH$ est grand, plus le nombre de calculs effectués par un thread augmente. Ainsi, la valeur optimale de $BLOCK\_WIDTH$ est une balance entre la complexité computationnelle et l'optimisation de la mémoire. Augmenter $BLOCK\_WIDTH$ revient en effet à diminuer le nombre de blocs nécessaires, et donc le nombre d'aller-retour entre la mémoire globale GPU et la mémoire partagée. Empiriquement, la valeur de 64 apparaît être optimale [2].

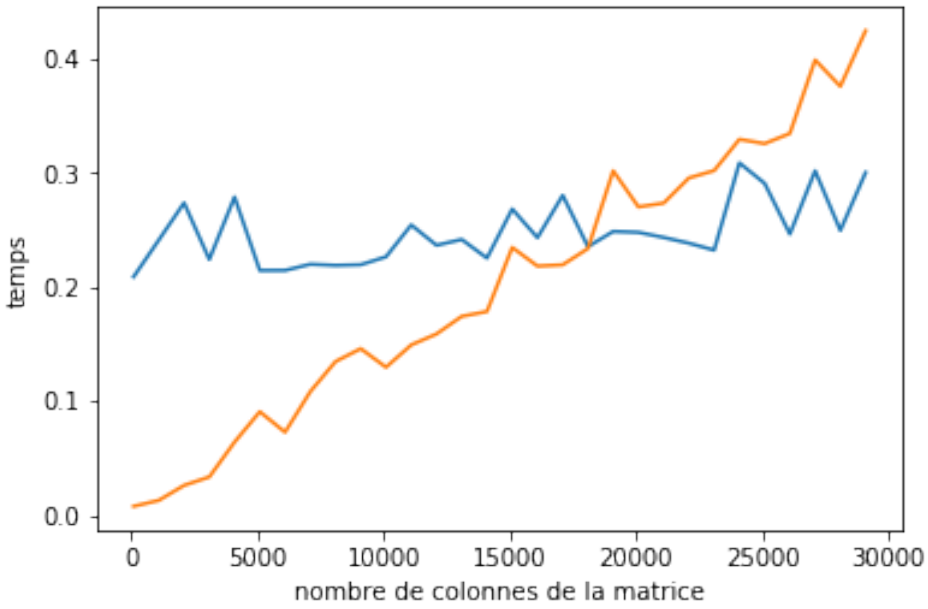
Concernant la dimension de la Grid, celle-ci sera de $\left(\lceil \frac{N\_COL}{BLOCK\_WIDTH} \rceil, \lceil \frac{N\_ROW}{BLOCK\_HEIGTH} \rceil, 1\right)$
Les threads sont quant à eux indicés de manière unidimensionnelle dans le bloc : chaque ligne du bloc correspondant à un thread (*threadIdx.x* parcourt les entiers de 0 à 1023).

3.1.2 Performance comparée CUDA/numpy

On teste la fonction pour observer ses performances, en comparaison de son équivalent numpy. On fixe d'abord $N\_ROW = 5000$ et on teste $N\_COL \in \{1000, ..., 30000\}$, en vérifiant naturellement que l'on obtient les bons résultats. On est limité par la taille de la mémoire pour la dimension des matrices que l'on veut tester.

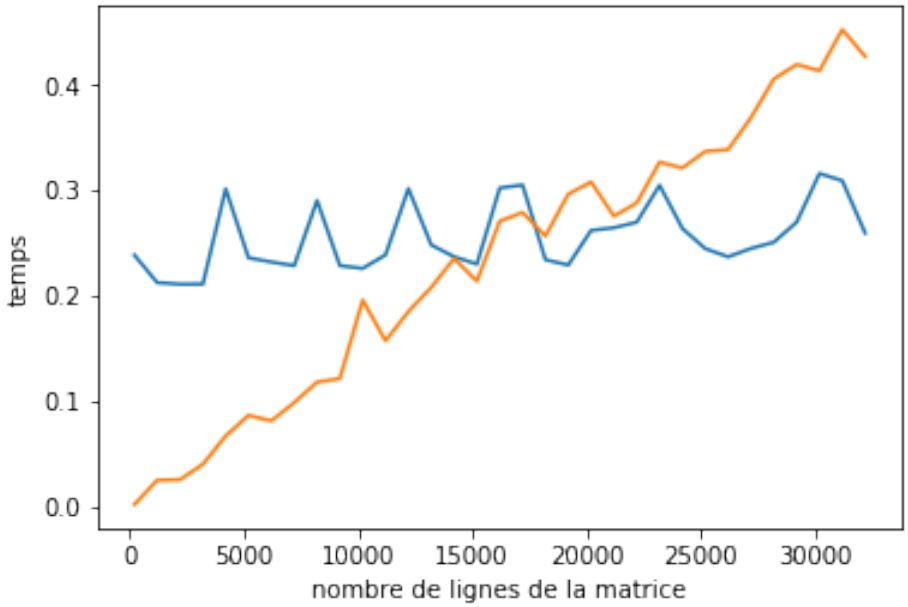
On voit ainsi que le temps d'exécution en CUDA augmente à peine avec la dimension, alors que celui de numpy augmente linéairement. On peut donc être satisfaits de la performance de notre fonction au-delà d'une certaine dimension.

2. E. Opavsky E. Uysaler, 2012



Performance comparée de CUDA (bleu) et Numpy (jaune) pour $N\_ROW = 5000$ et différentes valeurs de N_COL

On teste ensuite $N\_COL = 5000$ et on teste $N\_ROW \in \{1000, ..., 30000\}$.



Performance comparée de CUDA (bleu) et Numpy (jaune) pour $N\_COL = 5000$ et différentes valeurs de N_ROW

La fonction ne pose aucun problème de scalabilité et fonctionne pour toutes les dimensions de matrice.

3.2 CenterMatrix : centrage d'une matrice

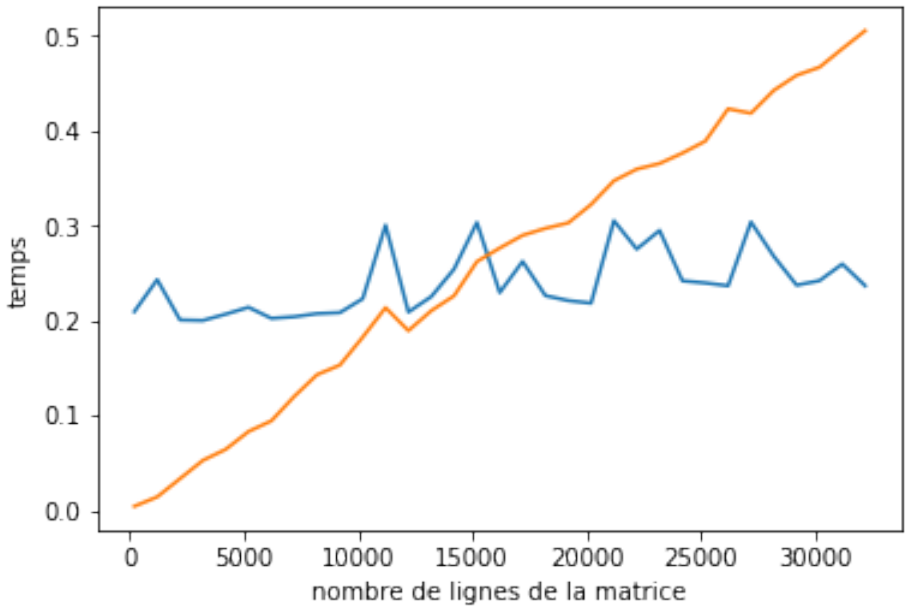
3.2.1 Présentation de l'algorithme

La fonction prend en entrée une matrice a_gpu à laquelle on veut ajouter, colonne par colonne le vecteur v multiplié par un scalaire $scalar$. Pour centrer une matrice w est le vecteur des sommes sur la colonne et $scalar$ est $\frac{-1}{N}$, avec N le nombre de lignes de la matrice. Cet algorithme est plus aisé que le précédent, nous en donnerons simplement les grandes lignes. Nous avons choisi de l'aborder en découpant la matrice a_gpu en blocs de hauteur $BLOCK\_HEIGHT = MAX\_THREADS\_PER\_BLOCK$ et de largeur $BLOCK\_WIDTH = 1$. Il est en effet primordial d'utiliser autant de threads que possible et il n'y a pas d'intérêt à ce qu'un thread fasse deux calculs qui n'ont rien à voir. Les threads sont donc considérés de manière unidimensionnelle : un par ligne, pour les 1024 lignes du bloc.

Dans notre cas, au sein d'un bloc, la valeur de la moyenne à soustraire élément par élément est la même, on stocke donc cette valeur (correspondant à *blockIdx.x*) dans la mémoire partagée pour un accès plus rapide.

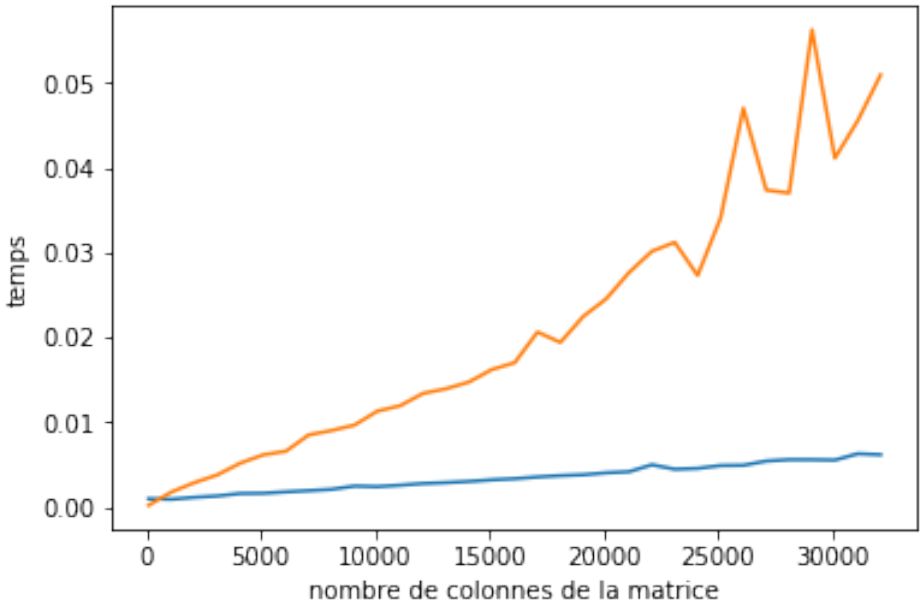
La dimension de la Grid est ainsi : $\left(p, \lceil \frac{N\_ROW}{BLOCK\_HEIGTH} \rceil, 1\right)$

3.2.2 Performance comparée CUDA/numpy



Performance comparée de CUDA (bleu) et Numpy (jaune) pour $P = 200$ et différentes valeurs de N

3.2.3 Performance comparée CUDA/numpy



Performance comparée de CUDA (bleu) et Numpy (jaune) pour $N = 200$ et différentes valeurs de P

On constate ainsi que, lorsque la matrice devient assez grande, la performance de l'algorithme distribué dépasse celle de la version numpy.

3.3 Transpose : transposition d'une matrice

Nous avions tout d'abord eu recours à une version naïve de l'algorithme, dans laquelle les threads écrivaient à de multiples endroits non contigus de la mémoire, générant des latences importantes. Nous avons ensuite eu recours à une version plus complexe, pour éviter les bank conflicts (conflits d'accès à la mémoire). La version que nous avons finalement mise en place utilise des tiles dont la dimension est fixée à 32.
Cette fonction présente toutefois des failles de fonctionnement. Elle tourne rapidement et est généralisable à des matrices de tailles qui ne soit pas des multiples de 32 (contrairement à la majorité des algorithmes présentés) mais ne donne de bons résultats que lorsque la matrice est carrée.

3.4 Fonction de calcul de la norme

Pour cette fonction, nous avons choisi d'utiliser les propriétés des produits scalaires, de telle sorte que $||x||^2 =< x, x >$. Grâce à ce procédé, le calcul de la norme est grandement simplifié. Les threads sont numérotés selon un seul indice, tout comme les blocs. Le bloc, caractérisé par $MAX\_THREADS\_PER\_BLOCK$ est donc de largeur $MAX\_THREADS\_PER\_BLOCK$ et le nombre de blocs, dans la longueur, est $n\%1024$.

4 IMPLÉMENTATION

FORMAT DES INPUTS On pourrait dans un premier temps remarquer que l'on considère tout au long du code les matrices comme des vecteurs apposés successivement des colonnes de la matrice originale. Si ce choix s'est imposé dans notre compréhension du fonctionnement de CUDA, il s'est avéré difficile à manier au fil de l'exercice. Ainsi par exemple `X\_gpu[k*N:(k+1)*N]` représente la colonne k de la matrice `X\_gpu (N,P)`.

LOGIQUE DE L'IMPLÉMENTATION On utilise en tant que données uniquement la matrice à réduire qu'on appelle X dans le CPU et Xraw_gpu une fois dans le GPU. A cela s'ajoutent quelques hyperparamètres : ϵ qui contrôle la marge d'erreur acceptable dans la convergence, N_ITER qui indique le nombre d'itérations afin de trouver la convergence, N_COMPONENTS le nombre de dimensions auxquelles réduire la matrice X.
A la fin de l'algorithme sont produits deux matrices :
- d'une part T qui regroupe les coordonnées de X selon les nouveaux axes `\lambda_k`
- d'autre part P qui regroupe les coordonnées des variables dans les nouvelles composantes.
On peut ainsi chercher à comprendre quelles sont les variables d'origine qui ont eu le plus influencé le résultat des composantes.

CONTENU Nous avons choisi d'utiliser la forme du script Python pour le rendu, organisé comme suit : les fonctions CUDA ont chacune leur script qui permet de les comprendre mais aussi de les appréhender avec des exemples et des tests intégrés. Ainsi le script `OpMatrix1D` est consacré aux transformations linéaires de matrices, celui `OpMatrix2D` aux multplications de matrices par vecteur, `OpMatrixTranspose` est dédié à la transposition de matrices tandis que `OpVectorNorm` se charge du produit eculidien d'un vecteur. Ces différentes briques servent à rédiger le script `Main` qui contient l'ensemble du processus nécessaire à l'obtention d'une PCA.
Si la plupart des scripts sont complètement flexibles quant à la taille de matrices/vecteurs concernés (dans la limite de la mémoire globale disponible i.e inférieur à 10GB dans notre cas, c'est-à-dire 28 millions d'entiers), nous n'avons pas réussi à nous départir de la contrainte du format carré pour la transposition de matrices. En découle donc l'impossibilité d'avoir une implémentation parfaitement fonctionnelle de la PCA. `Main` est donc plus un script-vitrine que réellement opérationnel.

CE QU'ON L'ON AURAIT AIMÉ FAIRE Idéalement, nous aurions souhaité pouvoir approfondir plusieurs éléments qui nous ont interpelés au long de notre travail :

— étudier la vitesse de convergence de l'algorithme. En effet, le recours à l'orthogonalisation des loadings et des scores à chaque itération s'avère certainement coûteux au plan computationnel ;

— observer l'impact du choix float32/float64 sur le temps d'exécution. Sur les grandes matrices, il apparaît que lorsque l'on fait un produit scalaire, des divergences apparaissent entre notre fonction CUDA et sa symétrique numpy, du fait notamment d'erreurs d'arrondis qui s'accumulent.

5 Conclusion

Notre parti pris dans ce travail a été de descendre au plus bas échelon possible, où l'on choisit finement quel thread manie quel nombre. Si ce travail nous a permis de bien mieux comprendre les difficultés logicielles derrière le traitement de données en grande dimension (accès mémoire, parallélisation, erreur d'arrondi qui s'accumule, etc.), la tâche s'est avérée si complexe pour nous que nous n'avons pas été capables de mettre sur pieds un algorithme n'utilisant que les fonctions que nous avions redessinées pour CUDA. Nous avons toutefois privilégié ce chemin à celui d'une mise en application à plus haut niveau : même si les chances de réussite auraient été bien supérieures, nous ne regrettons rien de la démarche. Pour le CUDA'près, nous aurons en tout cas déjà acquis une connaissance pratique du fonctionnement des algorithmes.

Nous avons du réinstaller à plusieurs reprises l'ensemble Linux-Python-Cuda-PyCuda suite à des problèmes de pilotes graphiques ou de choc électrique. En plus de la connaissance, ce projet nous aura donc également appris la résilience.