# Système de recommendations sur GPU

**Eléments logiciels pour traitement de données massives**

Dimitri Bouche, Remi de Torres

January 5, 2018

## 1 Introduction

Nous nous proposons pour ce projet de constuire un système de recommandation pour contenu multimédia sur GPU. De nombreux algorithmes différents sont présentés dans [1]. Nous avons choisi pour notre implémentation une version relativement simple et classique dans le domaine : la complétion de matrice avec features sous-jacent. Une version sur GPU de cette algorithme est proposée dans [3] sur lequel nous nous sommes appuyés.

## 2 Base de données

Nous utilisons les bases de données de notes utilisateurs sur des films ("movies database") du site GroupLens. Ces bases de données sont accessibles sur le site [2]. Plusieurs tailles sont proposés (de 100K notes à 20M notes). Nous avons développé notre algorithme sur la plus petite par simplicité, avant de le tester sur des bases plus étendues - la section résultats de ce document porte notamment sur ce sujet.

## 3 Rapide présentation du problème mathématique

Notons $n_u$ le nombre d'utilisateurs et $n_m$ le nombre de films. Notons $R \in \mathcal{M}_{n_m,n_u}(\mathbb{R})$ la matrix des notes. La majorité des cellules de cette matrice sont vides. L'objectif de l'agorithme est de les compléter.

Pour ce faire nous supposons que les utilisateurs et les films ont un certain nombre $n_f$ de caractéristiques sous-jacentes. L'objectif est d'apprendre ces caractéristiques sous-jacentes à partir des cellules non-vides de $R$. Le problème est ainsi paramétrisé.

Notons :

- $P \in \mathcal{M}_{n_u,n_f}(\mathbb{R})$ la matrice des caractéristiques utilisateurs
- $Q \in \mathcal{M}_{n_m,n_f}(\mathbb{R})$ la matrice des caractéristiques films

La matrice de notes prédites est alors $\hat{R} = QP^T$. En conséquence nous cherchons une solution au problème (avec $\Omega$ ensemble des indices des cellules non-vides de $R$ et $\lambda$ coefficient de régularisation):

$$\min_{Q,P} \sum_{(i,j)\in\Omega} \left((R_{ji} - Q_j P_i^T)^2 + \lambda(||P_i||^2 + ||Q_j||^2\right)$$

Plusieurs approches sont possible pour résoudre ce problème. Nous utilisons l'approche d'Alternated least squares (ALS) présentée dans [3].

Les matrices $P$ et $Q$ sont alternativement fixées et les sous-problèmes des moindre carrés correspondant aux utilisateurs et aux films sont résolus. Notons :

- $\Omega_i^u$ l'ensemble des films notés par l'utilisateur $i$ et $\Omega_j^m$ l'ensemble des utilisateurs ayant noté le film $j$
- $Q_{\Omega_i^u}$ (resp. $P_{\Omega_j^m}$) la restriction de $Q$ à ses lignes dont l'indice appartient à $\Omega_i^u$ (resp. $\Omega_j^m$)
- $R_{\Omega_i^u}$ le vecteur des notes données par l'utilisateur $i$ et $R_{\Omega_j^m}$ le vecteur des notes recçues par le film $j$.

La procédure à proprement parler est la suivante :

- A $Q$ fixée, nous résolvons :

$$\forall i \in [\![1, n_u]\!], \min_{P_i} \sum_{j \in \Omega_i^u} \left((R_{ji} - Q_j P_i^T)^2 + \lambda(||P_i||^2 + ||Q_j||^2)\right)$$

$$\Leftrightarrow \min_{P_i} ||R_{\Omega_i^u} - Q_{\Omega_i^u} P_i^T||^2 + \lambda||P_i||^2$$

- A $P$ fixée, nous résolvons :

$$\forall j \in [\![1, n_m]\!], \min_{Q_j} \sum_{i \in \Omega_j^m} \left((R_{ji} - Q_j P_i^T)^2 + \lambda(||P_i||^2 + ||Q_j||^2)\right)$$

$$\Leftrightarrow \min_{Q_j} ||R_{\Omega_j^m} - Q_j P_{\Omega_j^m}^T||^2 + \lambda||Q_j||^2$$

Les sous-problèmes ont les solutions explicites suivantes [4] :

- $P_i = (Q_{\Omega_i^u}^T Q_{\Omega_i^u} - \lambda I)^{-1} Q_{\Omega_i^u}^T R_{\Omega_i^u}$
- $Q_j = (P_{\Omega_j^m}^T P_{\Omega_j^m} - \lambda I)^{-1} P_{\Omega_j^m}^T R_{\Omega_j^m}$

***Remarque importante:*** *La matrice R n'est en fait jamais explicitement construite, la base de données de ratings correspond à une matrice dont chaque ligne est constituée d'un utilisateur, d'un film, d'un timestamp et d'une note.*

Notons enfin que la mesure d'erreur généralement utilisée comme benchmark pour les systèmes de recommandation - pour le Netflix prize notamment - est la "root mean square error". Nous définissons cette mesure puisque nous allons y faire référence par la suite :

$$RMSE(P, Q) = \sqrt{\frac{1}{Card(\Omega)} \sum_{(i,j) \in \Omega} (R_{ji} - Q_j P_i^T)^2}$$

## 4 Notes sur l'implémentation

Notre implémentation est en Python à l'aide de la librairie **pycuda**. Nous avons minimisé les transferts entre CPU et GPU dans notre implémentation du fait de la faible bande passante entre les deux.

La partie préparation de la base de données est faite sur le CPU à l'aide de pandas principalement.

Une fois cela fait, un unique transfert est effectué :

- Pour initialiser les matrices $P$ et $Q$ sur le GPU
- Pour passer la base de données préparée sur le GPU. Notons que non pas une version de la base mais deux sont d'ailleurs transférées plus une base de test.

Le choix d'avoir deux versions de la base est un compromis dans la mesure où pour chacune des étapes d'ALS, avoir à tour de rôle une base de données ordonnée par utilisateurs et une base de données ordonnée par films est très avantageux pour extraire les sous-matrices et sous-vecteurs $Q_{\Omega_i}$, $R_{\Omega_i}$, $P_{\Omega_j}$ et $R_{\Omega_j}$ sur le GPU. Ceci est un compromis - pas forcément optimal certes - que nous avons fait pour plusieurs raison :

- Trier sur le GPU n'est pas forcément une bonne idée - les algorithmes de tri ne sont pas du tout un point fort des GPUs - d'autant qu'il faudrait retrier avant chaque étape de résolution en utilisateurs et avant chaque étape de résolution en films.
- D'un autre côté effectuer à chaque sélection de sous-problème une opération de type "query" pour selectionner les films notés par un utilisateur ou les utilisateurs ayant noté un film semble coûteux
- Enfin stocker un objet de type dictionnaire associant à un utilisateur la liste des films qu'il a vus et à un film la liste des utilisateurs qui l'ont notés n'est pas du tout trivial puisque les dictionnaires n'existent pas sur le GPU et traduire cela en tableau ne semble pas une bonne idée puisque cela implique de stocker deux tableaux encore plus gros que la base de données, remplis en grande partie de zéros.

Les itérations d'ALS - un passage de tous les films et de tous les utilisateurs - sont effecutées uniquement sur le GPU. Pour l'opération d'inversion de matrice nous avons utilisé le module **scikit-cuda** qui permet d'appeler des routines de la librairie **cuSolver** de Nvidia. Celle-ci contient des implémentations de routines haut niveau d'algèbre linaire sur GPU - dont l'inversion de matrice et la décomposition de Cholesky que nous avons utilisées pour ce projet. Nous avons également utilisé l'implémentation du produit matriciel de cette librairie.

Les matrices de features et la base de notes étant sur le GPU nous avons également implémenté les fonctions de coûts et de "mean square error" sur le GPU. Néanmoins pour pouvoir garder une trace de l'évolution de la performance à chaque itération, une fois les calculs de coût et de RMSE faits sur le GPU, ceux-ci sont enregistrés dans une liste sur le CPU - 3 floats32 sont donc passés sur le GPU à chaque itération d'ALS, typiquement 10 itérations sont faites donc le coût de transfert est minime.

D'autre part, afin de pouvoir éxécuter la totalité de l'algorithme sur le GPU, nous écrit par nous-même certains kernels en **cuda C** en utilisant les fonctions de compilation de **pycuda** qui permettent ensuite d'exécuter ces kernels comme des fonctions Python. Effectivement, même si des ressources relativement importantes d'algèbre linéaire sont disponibles pour le GPU, beaucoup de fonctions dont nous avions besoin n'étaient pas réplicables de manière simple par combinaisons des librairies **pycuda** et **scikit-cuda**. Nous présentons à titre d'exemple détaillé un de ces noyaux :

- "SqrNormRowsKernel" : applique l'opération norme au carré à toutes les lignes d'une matrice. Retourne donc un vecteur dont la $i$ ème composante est la norme au carré de la ligne $i$ de la matrice. Ce kernel utilise des blocks en 2D (threads indicés sur 2D) qui reproduisent l'opération cible sur des sous-matrices de la matrice originale. Pour un block donné :
  1. chaque thread - indiçons les par tx et ty - calcule le carré de la cellule de coordonnées (tx, ty) dans la sous-matrice et stocke le résultat en mémoire partagée (__**shared** __) pour que les autres threads du block y aient accès.
  2. Avant de sommer le long des lignes, pour ne pas avoir un résultat erroné, on attend que chaque thread ait fini en utilisant __**syncthreads()**.

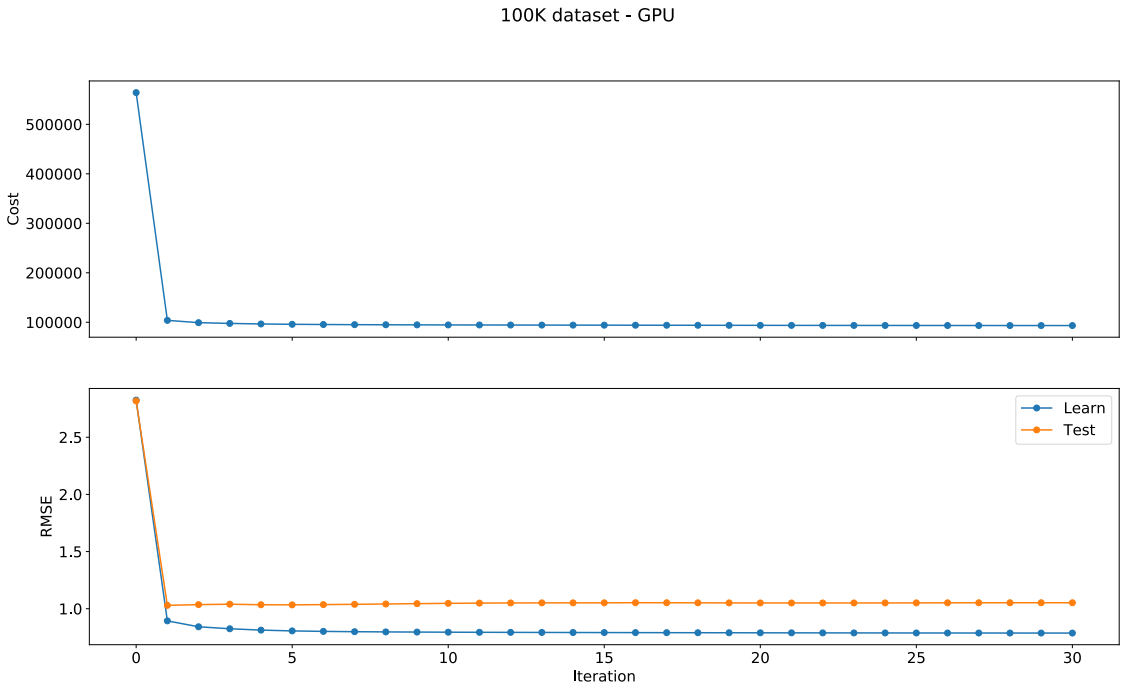
3. Le premier thread de chaque ligne somme les résultats de sa ligne auxquels il a accès grâce à la mémoire partagée et les écrit dans le vecteur devant contenir le résulat.

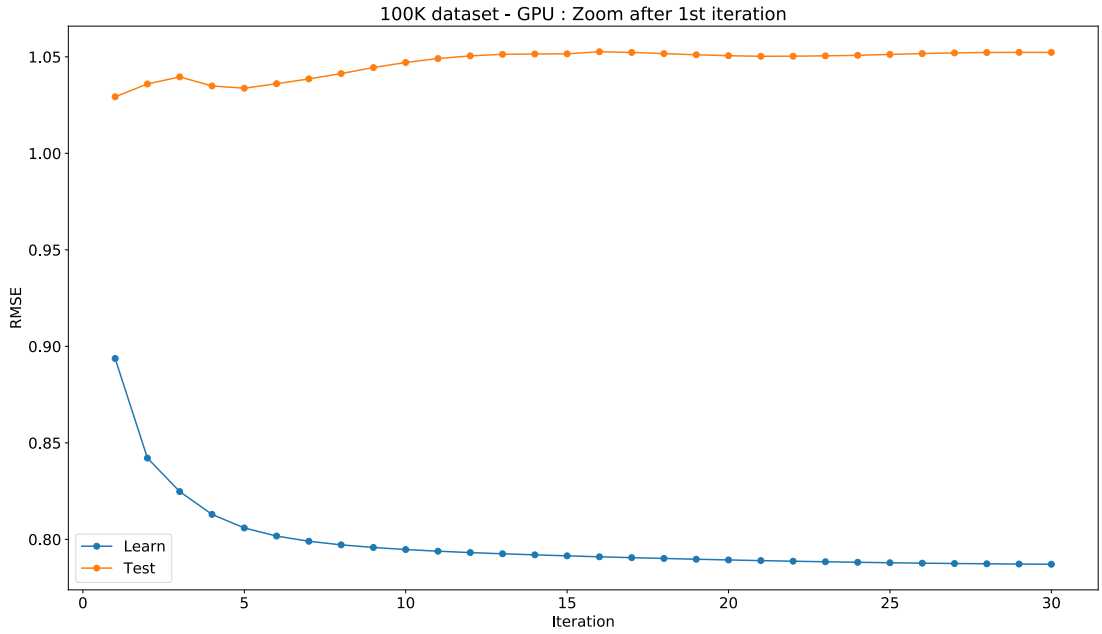
Enfin, nous avons systématiquement converti nos données en simple précision (float32 pour les réels et int32 ou uint32 selon les cas pour les entiers). Nous avons effectivement fait quelques expériences en 64 bits sur le GPU et le temps d'éxécution s'en voyait multiplié par bien plus de 2. Les cas dans nos tests personnels où la différence de précision est devenu un problème concernait l'inversion de grosses matrices mal conditionnées. Or les matrices que nous inversons ici sont petites (de taille $n_f$ qui sera typiquement petit $\leq 10$), bien conditionnées grâce au terme de pénalité qui booste la diagonale et symétriques - donc factorisables en Cholesky.

## 5 Résultats

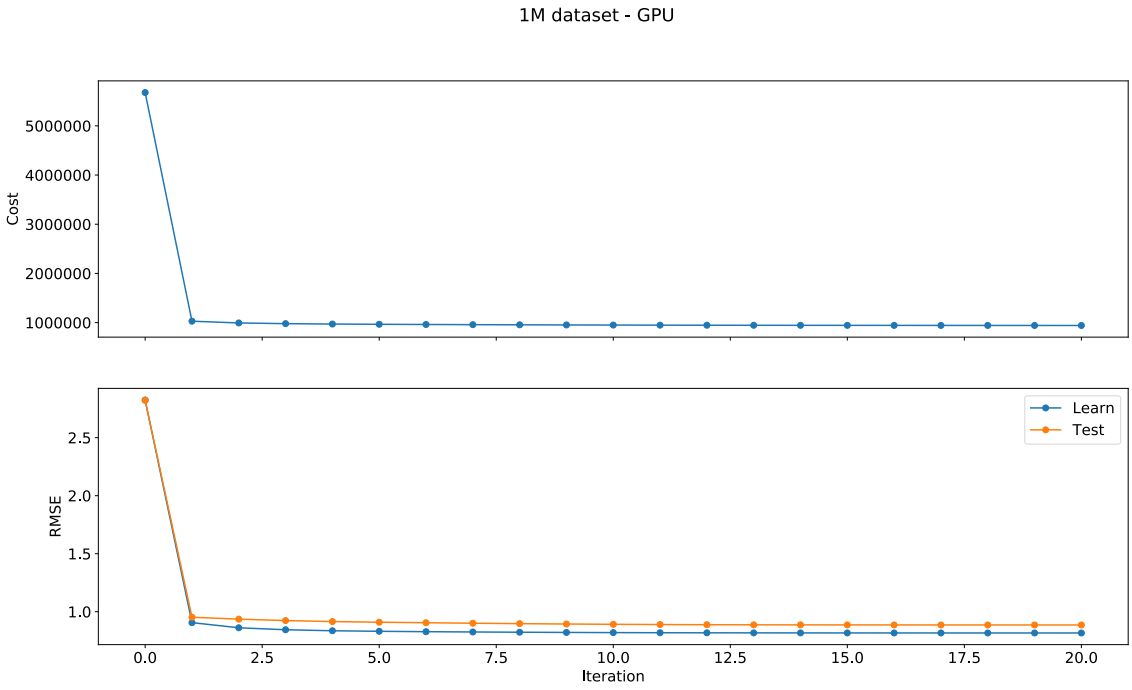
Les résultats sont cohérents sur le plan mathématique. Nous parvenons bien à la réduction de la fonction d'objectif attendue. Nous avons écrit le même algorithme sur le CPU pour vérifier/comparer les performances et nous trouvons exactement les mêmes résultats à quelques erreurs de $10^{-7}$ près. Nous prenons un nombre de features $n_f = 4$ d'après les recommandations de la littérature [1].Ces recommandations paraissent rationnelles pour éviter d'avoir un espace des paramètres trop grand par rapport aux nombres de données d'apprentissage.

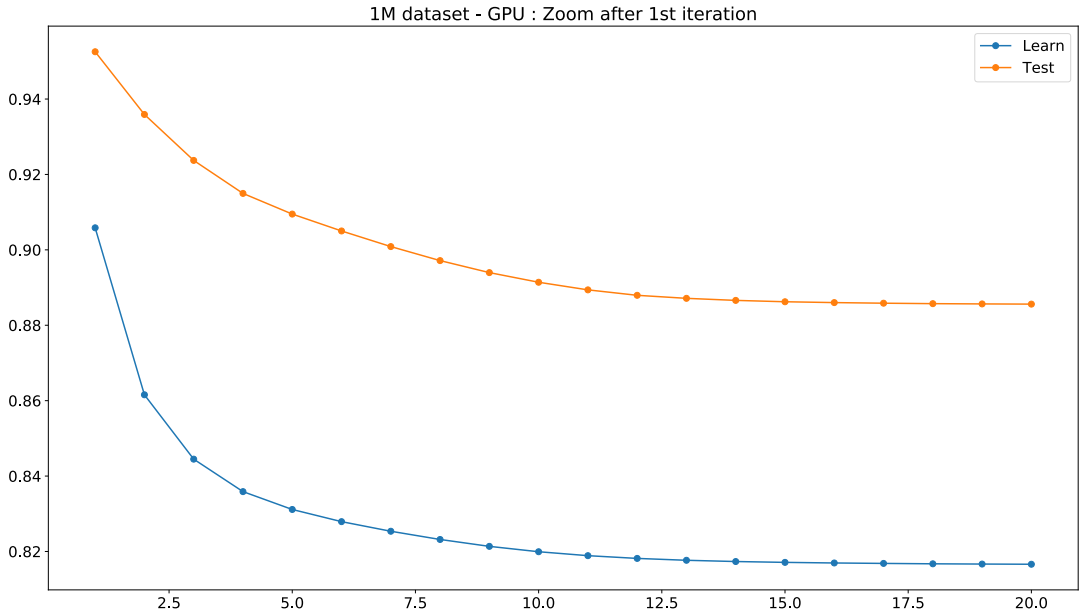
Nous avons implémenté notre algorithme sur un petite base de données pour commencer (100K notes pour 1000 utilisateurs et 1700 films). Pour ce premier dataset, la convergence est rapide. Effectivement au bout de la 5ème itération, nous constatons que le RMSE sur la base de test commence à remonter ce qui signale que l'algorithme est en train d'overfitter les données. Pour ce dataset la version GPU est toutefois moins rapide que la version CPU qui prend à peu près 0.63X le temps de calcul GPU, ce qui n'est pas vraiment le résultat escompté.






Nous avons ensuite testé l'algorithme sur une base de données un peu plus conséquente (1M notes pour 6000 utilisateurs et 4000 films). La convergence est moins rapide que dans le cas précédent et demande une vingtaine d'itération. Nous n'observons effectivement pas le même phénomène de remontée du RMSE sur la base de test donc l'algorithme ne semble pas overfitter, même après 20 itérations. Nous aurions donc pu prolonger l'apprentissage un peu plus même si les gains à espérer semblent assez minimes. Pour cette base de données plus conséquente, l'implémentation sur GPU est toutefois plus rapide que celle sur CPU qui prend à peu près 1.23X le temps de calcul GPU, soit une économie de $\approx 18$ % de temps de calcul.






Nous avons ensuite fait une tentative sur une base de données encore plus conséquentes (10M et 5M notes). Néanmoins l'éxécution est rapidement stoppée par la taille de la mémoire de la carte graphique (2Go), puisque les vecteurs et matrices ne rentrent plus dedans - pycuda._driver.MemoryError: CuMemAlloc failed : out of memory. Nous nous retrouvons ici confrontés aussi aux possibles limites dues à notre choix de stocker 2 versions de la base de données ordonnées différemment.

## 6 Conclusions

En conclusion, nous avons développé un système de recommandations de films optimisé par moindre carrés alternés sur le GPU. Pour une petite base de données (100K), la version GPU est significativement plus lente que la version CPU. Toutefois en augmentant la taille de la base de données (1M) nous arrivons à avoir un gain de vitesse par rapport au CPU même si celui-ci est assez faible. Ce gain décevant est probablement à mettre en relation ...

- avec notre implémentation de nombreux kernels GPU "à la main" là où la version CPU utilise **numpy**.
- avec le design global de notre algorithme qui peut très certainement être amélioré et optimisé.
- enfin, avec des facteurs matériels : le laptop sur lequel nous éxécutons a un gros processeur (IntelCore i7 7700 HQ @ 2.80Ghz) et une carte graphique relativement plus modeste (Nvidia GeForce GTX1050, 2Go).

Enfin pour des bases de données encore plus grosses (5M, 10M), la taille de la mémoire de la carte graphique devient un problème puisque'elle ne permet plus de stocker les matrices et vecteurs nécéssaires.

## References


[1] Ines Dutra, Alipio Jorge, and André Valente Rodrigues. "Accelerating Recommender Systems using GPUs". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (2015).

[2] *GroupLens' webpage*. https://grouplens.org/datasets/movielens/.

[3] Fransesco Ricci et al. *Recommender Systems Handbook*. Ed. by Springer. 2011.

[4] *Tikhonov regularization Wikipedia's page*. https://en.wikipedia.org/wiki/Tikhonov_regularization.