Factorisation de matrice en parallèle : implémentation de FPSGD sur GPU

Damien BABET, Julie DJIRIGUIAN,
Eléments logiciels pour le traitement des données massives,
compte-rendu de projet

5 février 2017

1 L'algorithme FPSGD

1.1 Factorisation de matrice

La factorisation de matrice est l'une des principales méthodes utilisées par les systèmes de recommendation. Elle vise à diminuer la taille des données en approximant les valeurs non-nulles d'une matrice $\mathbf{R}$ de très grande taille, mais *sparse*, c'est-à-dire presque vide, par un produit de deux matrices ($\mathbf{P}$ et $\mathbf{Q}$) de faible dimension, comprenant un petit nombre k de colonnes. Ces k colonnes peuvent alors être interprêtées comme des variables latentes. Les produit de ces deux matrices permet de faire des prédictions sur les valeurs manquantes de $\mathbf{R}$, et, donc, des recommandations. Par exemple, $\mathbf{R}$ peut être une matrice rassemblant les notes données à des films par des usagers. $\mathbf{Q}$ sera alors une matrice des films, notés selon chacune des k variables latentes, et $\mathbf{P}$ une matrice des usagers, avec leurs goûts dans chacune des k variables latentes. Le problème est donc de trouver $\mathbf{P}$ et $\mathbf{Q}$ qui minimisent une fonction de perte L, par exemple quadratique :

$$\begin{aligned}\mathbf{R} \approx \mathbf{P}\mathbf{Q}', \text{avec } \mathbf{R} \in \mathcal{M}_{m,n}(\mathbb{R}), \mathbf{P} \in \mathcal{M}_{m,k}(\mathbb{R}), \mathbf{Q} \in \mathcal{M}_{n,k}(\mathbb{R}) \\ L = \sum_{i,j \in Z} L(\mathbf{R}_{ij}, \mathbf{P}_{i*}, \mathbf{Q}'_{*j}) = \sum_{i,j,\in Z} (\mathbf{R}_{ij} - \mathbf{P}_{i*}\mathbf{Q}'_{*j}))^2 \\ \text{avec } Z = \{i, j,\ tq\ \mathbf{R}_{ij} \neq 0\}\end{aligned}$$

Plusieurs types de contraintes supplémentaires sont possibles sur $\mathbf{R}$, $\mathbf{P}$ et $\mathbf{Q}$, définissant des problèmes et des résultats différents, par exemple une contrainte de positivité. Le problème de la recherche des $\mathbf{P}$ et $\mathbf{Q}$ approximant au mieux R est diffile, car non convexe, les lignes de $\mathbf{P}$ (ou de $\mathbf{Q}$) étant interdépendantes. Cette recherche peut se faire par *gradient descent* mais le gradient est trop lourd à calculer pour de grandes matrices. On peut

alors recourir à une descente de gradient stochastique : à chaque itération, le gradient n'est calculé que par rapport à un seul élément de la matrice $\mathbf{R}$, et seules les lignes correspondantes de $\mathbf{P}$ et $\mathbf{Q}$ sont mises à jour [1]. En notant $\mathbf{p}_i$ et $\mathbf{q}_j$ les lignes i et j des matrices , et en ajoutant des facteurs de normalisation λ_p et λ_q à la fonction objectif pour éviter le surraprentissage, le problème d'optimisation devient :

$$\underset{\mathbf{P},\mathbf{Q}}{min} = \sum_{i,j \in Z} ((r_{ij} - \mathbf{p}_i' \mathbf{q}_j)^2 - \lambda_{\mathbf{P}} \|\mathbf{p}_i\|^2 - \lambda_{\mathbf{Q}} \|\mathbf{q}_j\|^2) \tag{1}$$

Pour un terme individuel de la somme, le gradient se calcule aisément et la mise-à-jour est :

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \gamma(e_{ij}\mathbf{q}_j - \lambda_{\mathbf{P}}\mathbf{p}_i) \tag{2}$$

$$\mathbf{q}_j \leftarrow \mathbf{q}_j + \gamma(e_{ij}\mathbf{p}_i - \lambda_{\mathbf{Q}}\mathbf{q}_j) \tag{3}$$

avec $e_{ij} = r_{ij} - \mathbf{p}_i' \mathbf{q}_j$

1.2 Parallélisation par bloc : de DSGD à FPSGD

La stratégie de parallélisation suivie ici consiste à découper la matrice $\mathbf{R}$ en s^2 blocs dont s seront traités en parallèle. Pour éviter que les mises-à-jour issues des différents blocs se chevauchent, les s blocs choisis sont indépendants : ils ne partagent aucune coordonnée, les parties de $\mathbf{P}$ et $\mathbf{Q}$ utilisées dans chaque fil de calcul parallèle sont donc distinctes. Cet algorithme est le DSGD [2] (« distributed stochastic gradient descent »).

L'apport de l'algorithme FPSGD [3] (« fast parallel stochastic gradient descent ») est d'augmenter le nombre de blocs ($(s+h)^2$) qui découpe $\mathbf{R}$. Comme la parallélisation continue d'opérer sur s fils de calcul, il reste toujours un certain nombre de blocs qui ne sont pas en cours de traitement mais qui sont indépendants de tous les blocs traités. L'intérêt est double. D'une part les fils de calcul qui terminent en premier peuvent se voir attribuer un nouveau bloc, indépendant de tous les autres en cours de calcul, et l'on évite ainsi d'avoir des fils inoccupés. D'autre part, l'attribution des blocs peut se faire de manière aléatoire tout en maintenant un nombre d'itération équivalent pour chaque bloc. Forts de ce niveau élevé d'aléa à l'échelle des blocs, les auteurs de FPSGD peuvent renoncer à utiliser l'aléa dans le traitement de chaque bloc, et effectuer les mises à jour dans l'ordre de lecture des observation, gagnant ainsi en vitesse d'accès mémoire.

1. [4]
2. [3]
3. [2]

En contrepartie, l'algorithme FPSGD implique une coordination plus fréquente entre les fils de calcul. L'implémentation proposée est conçue pour une parallélisation sur une machine unique, en exploitant les différents processeurs de la CPU. Elle reste donc intrinsèquement limitée. Notre projet consiste à paralléliser sur la GPU pour disposer de fils de calculs plus nombreux, tout en restant sur un machine unique.

2 Les difficultés spécifiques de la parallélisation sur GPU

Pour exploiter les capacités de calcul en parallèle de la GPU, nous avons utilisé le langage Cuda, accédé en python à travers le module pycuda [4].

Le calcul sur GPU (via Cuda) implique de transférer les données sous forme d'un certain nombre de vecteurs de dimension 1 à 3. Les processeurs de GPU disposent de mémoires locales de faible dimension, et d'un accès plus lent aux données de la mémoire globale. La question de la répartition optimale des données dans la mémoire locale est donc essentielle. Aujourd'hui, les algorithmes publiés de factorisation de matrice qui exploitent le calcul en parallèle sur GPU sont généralement moins exigeants que FPSGD en terme de synchronisation : il s'agit par exemple de HogWild (qui fait le choix de négliger le problème d'*overwriting*) ou de C++ (qui n'est pas un algorithme de SGD mais de ALS, « alternate least square », procédant alternativement à la mise-à-jour de $\mathbf{P}$ et $\mathbf{Q}$. Ces algorithmes peuvent également s'appuyer sur des briques élémentaires (OLS ou multplication de matrices) déjà codées et optimisées dans des librairies Cuda.

L'implémentation de FPSGD sur GPU n'offrait pas ces facilités. C'est peut-être d'ailleurs la raison pour laquelle elle ne semble pas avoir encore été réalisée (des test de performance [5] de différents algorithmes sur GPU mènent par exemple des comparaisons avec l'implémentation sur CPU de FPSGD, une comparaison qui souffre de la grande différence du nombre de fils de calcul parallèles dans un cas et dans l'autre).

Il serait cependant intéressant de connaître le comportement de FPSGD lorsqu'on augmente le nombre de fils de calcul (et donc le paramètre s). A un extrême, lorsque s est égal à 1, l'algorithme est une descente de gradient (même pas stochastique). A l'autre extrême, lorsque s^2 est de la taille de $\mathbf{R}$, il se rapproche d'une version très alourdie de HogWild (avec le mince avantage

4. les difficultés d'installation découlant de ce choix d'implémentation sont détaillées sur notre GitHub : https ://github.com/DamienBabet/LIBMF-with-Pycuda

5. [5]

d'éviter les problèmes d'*overwiting*). Il existe sans doute une parallélisation optimale (qui dépend de la matrice à factoriser), et il serait intéressant de chercher ce s optimal et de mener les comparaisons à partir de cette version de l'algorithme.

3 Implémentation

En l'état actuel, notre travail n'implémente pas l'algorithme FPSGD, mais seulement sa version plus simple, DSGD. Nous avons donc un *scheduler* simplifié, qui génère une permutation de $\{1, ..., s\}$ à chaque itération pour sur la GPU les s blocs de $\mathbf{P}$ dans l'ordre, les s blocs de $\mathbf{Q}$ dans l'ordre de la permutation, et les blocs de $\mathbf{R}$ correspondant. Nous avons également choisi la factorisation de matrice en facteurs positifs : notre algorithme n'offre pas de choix quant au type de factorisation possible.

Lors de nos essais sur des données artificielles de petite taille, notre *scheduler* fonctionne dans une version itérative de l'algorithme (sans utiliser Cuda et sans parallélisation). Notre *kernel* Cuda (le code executé s fois sur la GPU) est également fonctionnel, lorsqu'on alimente avec des données construites « à la main ». En revanche, nous ne sommes pas encore parvenus à faire fonctionner ensemble le *scheduler* et le *kernel*, manifestement pour des question de « contexte » dans la gestion de la mémoire que nous ne parvenons pas encore à comprendre.

Nous avons également implémenté le code permettant de charger des données réelles de grande taille et de les mettre en forme pour l'algorithme FPSGD (ou DSGD), ce qui en soit peut demander de recourir à des stratégies de parallélisation selon la taille des données. Ces données sont issues du site « MovieLens ». L'implémentation principale est disponible dans le *notebook* « Notebook_MatrixFactorization ».

4 Conclusion et suites possibles

Il est possible que l'implémentation de FPSGD sur GPU soit une mauvaise idée, l'algorithme étant particulièrement adapté pour une paralélisation modeste sur CPU. Notre état d'avancement ne nous permet pas encore de conclure.

Les étapes suivantes pour mener à bien notre projet sont claires, outre la résolution du problème de code pour faire fonctionner ensemble *scheduler* et *kernel*. Il s'agirait d'abord de tester le code sur des données réelles, et de mesurer le temps de calcul (pour un nombre donné d'itération puis pour un niveau donné de convergence), et sa variation en fonction de s. Il faut

évidemment compléter le *scheduler* pour implémenter effectivement FPSGD plutôt que DSGD. Par ailleurs, le code est très largement optimisable, notamment dans la gestion de la mémoire locale de la GPU (et la division du travail non seulement en *threads*, mais aussi en *blocks* de dimension 1 à 3). Enfin, il faudrait idéalement implémenter un certain nombre des fonctionnalités de LIBMF [6], le code des auteurs de FPSGD : plusieurs types de factorisation, plusieurs choix de fonction de perte, une gestion dynamique du taux d'apprentissage, en particulier.

Références

[1] Wei-Sheng Chin, Bo-Wen Yuan, Meng-Yuan Yang, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. Libmf : A library for parallel matrix factorization in shared-memory systems. *Journal of Machine Learning Research*, 17(86) :1–5, 2016.

[2] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(1) :2, 2015.

[3] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[4] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[5] Wei Tan, Liangliang Cao, and Liana Fong. Faster and cheaper : Parallelizing large-scale matrix factorization on gpus. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230. ACM, 2016.

6. [1]