Parallélisation sur CPU d'une séquence d'opérations matricielles extraite d'un modèle Llama

Matthieu Bricaire - matthieu.bricaire@ensae.fr
Yseult Masson - yseult.masson@ensae.fr
Code associé au rapport

Introduction

Ce projet vise à étudier des pistes d'optimisation d'un modèle Llama. Après avoir visualisé le modèle, nous avons choisi de rassembler et paralléliser la séquence d'opérations représentée sur le schéma ci-dessous :

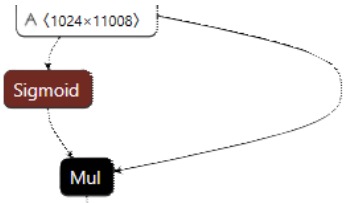

FIGURE 1 – Séquence d'opérations à paralléliser

Dans le modèle de base, ces opérations sont effectuées séquentiellement. On calcule d'abord la sigmoïde de chaque élément a d'une matrice A, ce qui nous donne une matrice B, puis on effectue l'opération $A * B$ terme à terme. Nous proposons donc de rassembler ces deux opérations et de calculer directement $\text{sigmoid}(A) * A$. Dans la suite du rapport, nous appelerons cette opération "sigmul".

Pour réduire le temps de calcul, nous avons parallélisé cette opération sur CPU avec Cython et C++. Nous avons utilisé les machines du SSP cloud pour mener nos expériences.

1 Différentes implémentations

Notre but est de comparer les temps d'exécution de différentes implémentations de `sigmul` :

1. Pytorch (implémentation de référence, qui consiste à effectuer d'abord la sigmoïde sur tous les éléments de A, et ensuite la multiplication terme à terme).
2. Implémentation naïve en Python, sans parallélisation (double boucle for).
3. C++ et Cython (on implémente `sigmul` en C++ dans le fichier `sigmul.cpp` du dépôt github, puis on utilise Cython pour faire le lien avec python dans `interface.pyx`).
4. Pur Cython (on implémente directement `sigmul` en Cython dans le fichier `interface.pyx`, qui compile ensuite en C++ et fait le lien avec Python).

2 Expériences menées et résultats obtenus

Les analyses sont effectuées dans le notebook `times_comparisons.ipynb`. On compare les temps d'exécution des différentes implémentations sur des matrices carrées de dimensions croissantes. Dans ce qui suit, on parle d'une matrice « de dimension d » pour désigner une matrice qui contient $d \times d$ coefficients. Pour chaque fonction, on mesure le temps d'exécution moyen sur 10 itérations. Les temps d'exécution peuvent varier légèrement selon le moment où le programme a été lancé, mais les tendances restent les mêmes. Les graphiques des temps d'exécution sont tous en échelle logarithmique pour faciliter la comparaison.

Pour une implémentation et une taille de matrice données, l'erreur est définie comme la moyenne de la différence en valeur absolue entre les résultats de notre implémentation et de celle en PyTorch.

2.1 Visualisation de l'effet de la parallélisation

On commence par appliquer nos quatre implémentations sur des matrices carrées de dimensions allant de 2^0 à 2^9 (512).

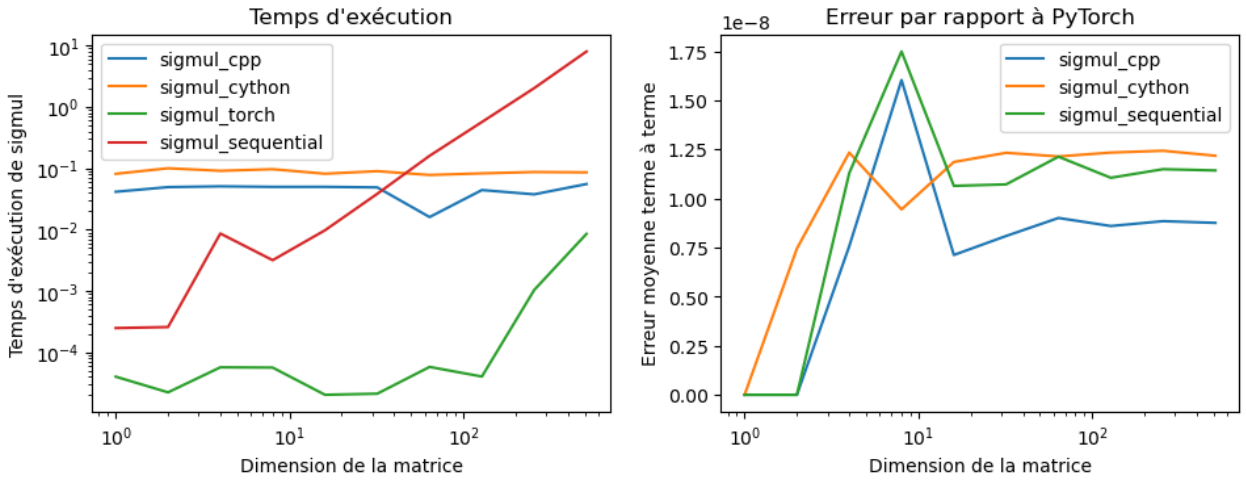

FIGURE 2 – Temps d'exécution et erreurs des quatres implémentations

L'implémentation de référence en PyTorch semble être optimale pour toutes les dimensions (Figure 2 gauche). L'approche naïve sans parallélisation est plus efficace que nos approches parallélisées sur des très petites dimensions, puis son temps d'exécution explose de façon exponentielle et dépasse ceux des approches Cython et C++ à partir de $d = 32$. Cette efficacité pour des petites matrices est certainement due au fait que le temps de transport d'information est plus faible que dans les approches parallèles et compense le temps de calcul plus élevé. Pour de plus grandes dimensions, cependant, le rapport entre ces deux coûts s'inverse.

Comme on peut le voir sur la figure 2 (droite), les erreurs commises sont négligeables, et ce même lorsque la taille des matrices augmente.

2.2 Comparaison des implémentations parallélisées de sigmul

Nous cherchons à présent à comparer nos implémentations en Cython et en C++ à l'implémentation de référence en PyTorch, sur des matrices de dimension 1 à 2^{14} (16384).

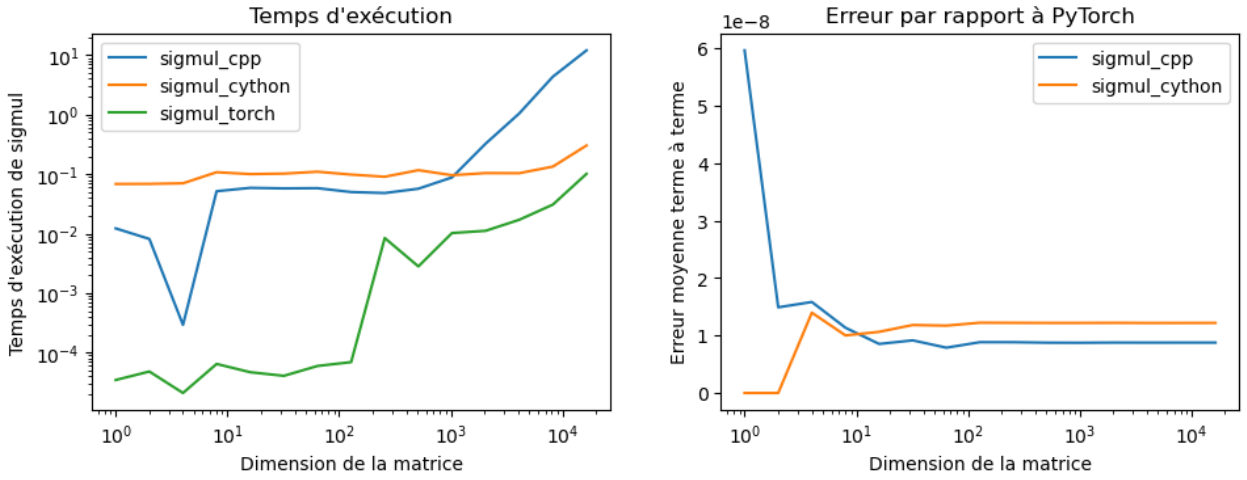

FIGURE 3 – Temps d'exécution et erreurs des implémentations C++, Cython et PyTorch

La version basée sur C++ fonctionne mieux pour de petites dimensions, puis son temps de calcul augmente de façon exponentielle à partir de 2^{10} (1024). Pour la suite des analyses, on retient donc uniquement la version Cython.

2.3 Comparaison plus poussée des implémentations en Cython et en PyTorch

Dans cette analyse, nous utilisons des matrices de dimensions allant de 1 à 2^{15} (32768).

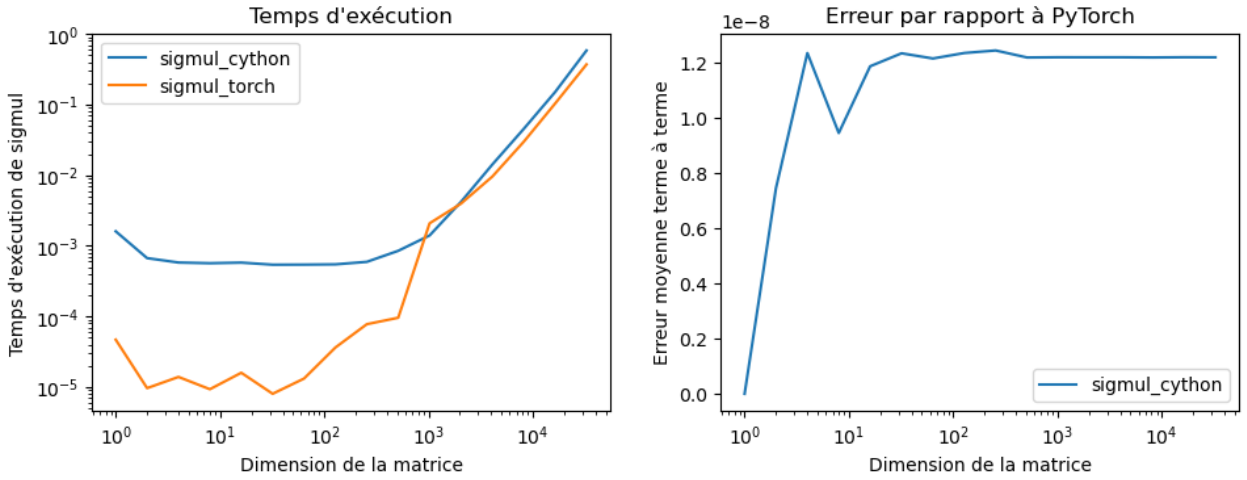

FIGURE 4 – Temps d'exécution et erreurs des implémentations Cython et PyTorch

Comme on peut le voir sur la figure 4, notre implémentation en Cython est bien moins bonne que PyTorch sur les petites dimensions, mais s'en rapproche pour les plus grandes dimensions. La figure 5 montre plus en détails le ratio entre les temps d'exécution de ces deux implémentations. Pour les plus grandes dimensions (2^{10} à 2^{15}), notre implémentation est entre 1.1 et 1.5 fois plus lente que celle de PyTorch, contre 40 à 70 fois pour les petites dimensions.

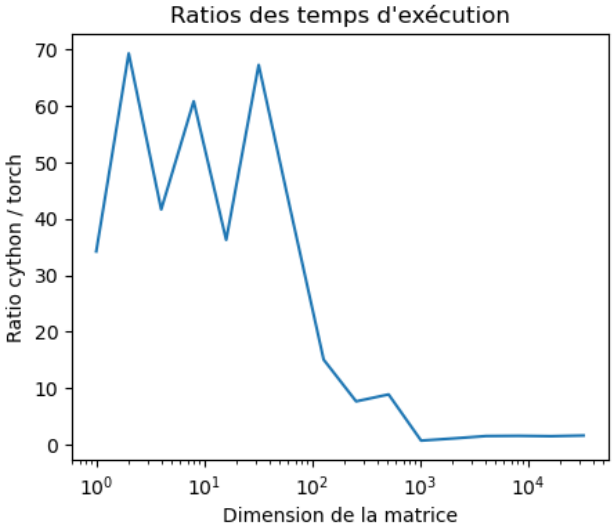

FIGURE 5 – Ratio des temps d'exécution entre Cython et PyTorch

Cependant, on a jusque là omis le coût de conversion des types entre C++ et PyTorch, qui est pourtant nécessaire si l'on veut réellement inclure la fonction dans le modèle Llama, codé en PyTorch.

2.4 Prise en compte des temps de conversion des types entre C++ et PyTorch

On ajoute à la fonction Cython les conversions nécessaires entre les types de données, afin de pouvoir prendre en entrée un `tensor` et renvoyer un `tensor`. Cette considération est nécessaire dans une optique d'insertion de cette fonction dans un modèle codé en PyTorch.

Comme on peut le voir sur la figure 6, ajouter les conversions augmente considérablement le temps d'exécution de notre fonction `sigmul` implémentée ave Cython. Celle-ci devient très inefficace par rapport à PyTorch.

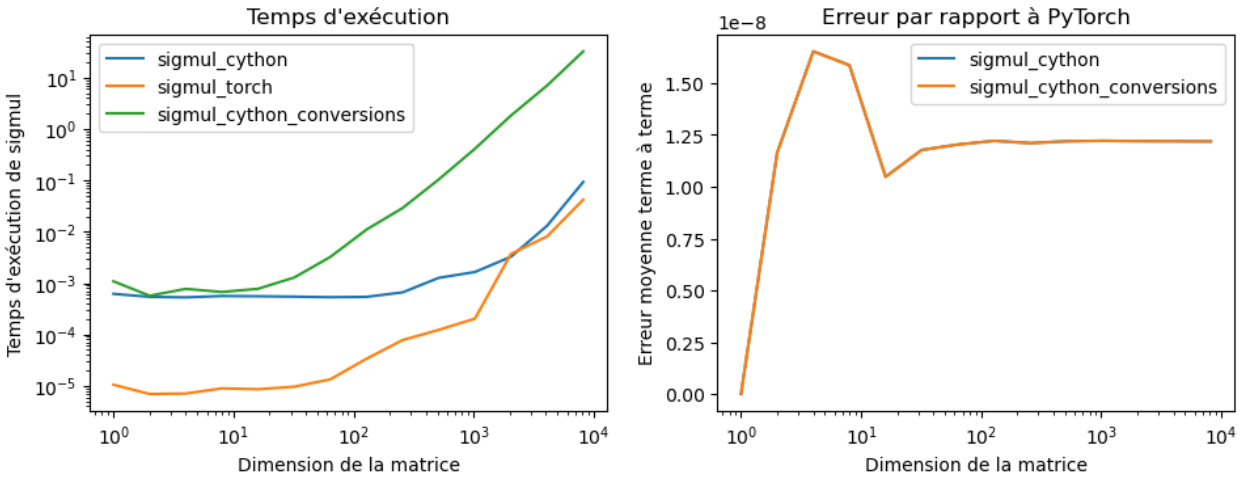

FIGURE 6 – Temps d'exécution et erreurs des implémentations Cython, Cython avec conversion des types, et PyTorch

Conclusion

En conclusion, PyTorch est toujours meilleur que nos implémentations, en particulier lorsque l'on adapte nos fonctions pour qu'elles puissent s'insérer dans un modèle Llama. Il n'est donc pas avantageux de remplacer cette opération dans le modèle initial. Les fonctions que nous avons choisies de rassembler (sigmoïde et multiplication terme à terme) sont très basiques et doivent donc être très bien optimisées en PyTorch, ce qui explique que nos performances ne soient pas aussi bonnes, même en effectuant les deux opérations simultanément et non pas à la suite l'une de l'autre.