Benchmark Adaptatif du Mécanisme d'Attention

Arnaud Cournil, Marc Deroo, Laurent Vong

1 Introduction

Ce projet développe une stratégie de benchmarking adaptatif pour optimiser le mécanisme d'attention utilisé dans les modèles de Transformers. Nos travaux poursuivent trois objectifs principaux. Premièrement, nous cherchons à surpasser les performances de NumPy grâce à des implémentations optimisées utilisant les instructions vectorielles AVX et la parallélisation. Deuxièmement, nous visons à développer une méthode de recherche adaptative capable de déterminer les meilleurs hyperparamètres avec un minimum d'essais. Enfin, nous nous efforçons d'assurer la reproductibilité des résultats en garantissant leur stabilité et leur cohérence d'une exécution à l'autre.

2 Méthodologie

2.1 Implémentations

Notre étude compare trois implémentations distinctes du mécanisme d'attention. La première utilise NumPy comme référence, s'appuyant sur ses opérations matricielles natives optimisées. La seconde exploite Numba pour la compilation JIT, permettant d'accélérer les boucles et les calculs sans modification majeure du code Python. La troisième, combine Cython et C++ avec des optimisations avancées incluant les instructions vectorielles AVX, la parallélisation via OpenMP et des algorithmes de multiplication matricielle par blocs.

2.2 Stratégie de benchmarking adaptatif

Nous adoptons une stratégie de recherche basée sur un algorithme génétique pour identifier rapidement la meilleure combinaison d'hyperparamètres. Une population initiale de configurations est générée aléatoirement, puis évolue au fil de plusieurs générations. À chaque étape, les configurations les plus performantes sont sélectionnées, croisées et légèrement mutées pour explorer efficacement l'espace de recherche. Nous avons initialement utilisé l'algorithme Successive Halving, mais celui-ci s'est révélé trop coûteux en temps de calcul pour notre cas. L'approche génétique permet de limiter le nombre d'évaluations à environ 18. Les hyperparamètres considérés sont la taille de bloc (8, 16, 32 ou 64), le nombre de threads (1, 2, 4 ou 8) et le type de données (float32 ou float64).

3 Résultats

3.1 Performance comparative

Dim	Type	Block	Threads	NumPy (s)	Numba (s)	Cython (s)	Speedup Numba	Speedup Cython
64	f32	8	2	1.05e-04	3.36e-05	1.23e-04	3.13	0.85
128	f32	16	2	1.00e-03	3.36e-04	2.87e-04	2.99	3.50
256	f32	32	4	2.43e-03	1.84e-03	1.57e-03	1.32	1.55
400	f32	64	4	6.98e-03	3.73e-03	4.25e-03	1.87	1.64
512	f32	16	1	1.51e-02	7.96e-03	1.25e-02	1.89	1.21
768	f32	32	4	4.41e-02	1.66e-02	3.22e-02	2.66	1.37
1024	f32	32	1	1.31e-01	4.16e-02	1.19e-01	3.16	1.10

TABLE 1 – Comparaison des performances des différentes implémentations

L'analyse des performances révèle des tendances significatives. Numba offre globalement les meilleures performances sur l'ensemble des dimensions testées. L'implémentation Cython/C++ montre des performances variables selon la dimension : particulièrement efficace pour dim=128, elle reste néanmoins supérieure à NumPy pour presque toutes les dimensions testées, à l'exception

de dim=64. Notre algorithme adaptatif a invariablement privilégié le type float32, confirmant sa supériorité en termes de performance par rapport au float64 pour cette application.

3.2 Analyse et conclusion

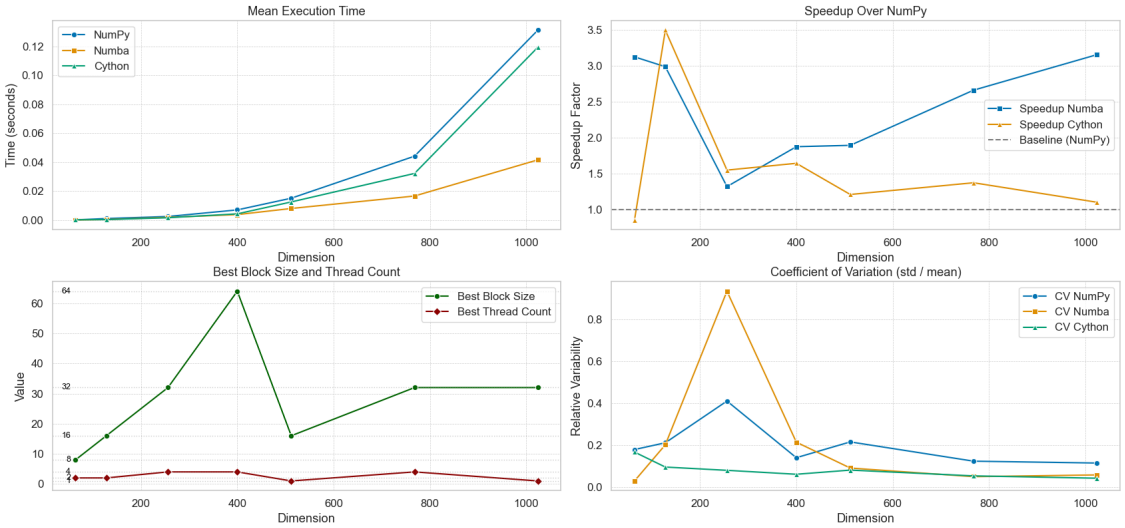

FIGURE 1 – Analyse comparative des performances

Le premier graphique montre une évolution pour les trois implémentations avec l'augmentation de la dimension des matrices. Pour les dimensions inférieures à 400, les performances sont relativement proches, bien que Numba conserve déjà un avantage. Au-delà de ce seuil, on constate qu'il y a un écart qui se fait entre les différentes implémentations.

Le graphique des speedups montre que la performance relative de Numba par rapport à NumPy reste constamment supérieure à 1. Cython présente une courbe plus irrégulière, avec un pic à dim=128 , suivi d'une diminution puis d'une stabilisation autour de 1.

L'examen des hyperparamètres ne révèle aucune tendance réelle.

Le graphique du coefficient de variation révèle que Numba présente la plus forte variabilité , tandis que Cython maintient la plus grande stabilité sur toutes les dimensions. Pour les grandes matrices les trois implémentations se stabilisent avec des CV très bas, suggérant qu'un nombre réduit de répétitions suffirait à ces dimensions.

Les résultats obtenus révèlent que Numba constitue le meilleur choix, malgré une instabilité dans les faibles dimensions. L'implémentation Cython se montre également intéressante, surpassant NumPy dans presque tous les cas testés, avec un pic à la dimension 128.