Rapport de projet de programmation parallèle

Delhommais Romain, Ligonniere Louise, Mekki Lila
ENSAE Paris

23 mai 2025

1 Contexte et objectif

Dans ce projet, nous mettons en œuvre le mécanisme d'**attention**, défini par

$$\text{Attention}(Q, K, V) \;=\; \text{softmax}\Big(\frac{Q\,K^\top}{\sqrt{d_k}}\Big)\,V$$

avec

$$Q \in \mathbb{R}^{m\times d_q}, \qquad K \in \mathbb{R}^{m\times d_k}, \qquad V \in \mathbb{R}^{m\times d_v},$$

et où la fonction *softmax* s'écrit, pour tout vecteur $z = (z_1, \ldots, z_n)$:

$$\text{softmax}(z)_i \;=\; \frac{e^{z_i}}{\displaystyle\sum_{j=1}^{n} e^{z_j}}.$$

Pour ce calcul, on utilisera un algorithme parallélisé. Notre ojectif est de concevoir un benchmark adaptatif permettant de trouver la meilleure option de parallélisation. On choisira d'abord un nombre restreint de paramètres à optimiser, puis on proposera des méthodes permettant de choisir la configuration la plus efficace en termes de temps de calcul, avec moins d'essais qu'une grille qui testerait toutes les configurations possibles de ces paramètres.

2 Choix des paramètres à optimiser

Nous avons choisi trois paramètres à optimiser : le nombre de threads, la taille de bloc, et le type utilisé par python pour stocker les valeurs. Les valeurs possibles pour ces trois paramètres sont données dans le tableau suivant. On obtient une grille de 4x4x2=32 combinaisons.

Paramètre	**Nombre de valeurs**	**Valeurs possibles**
Nombre de threads (`threads`)	4	1, 2, 4, 8
Taille de blocs (`blocksize`)	4	8, 16, 32, 64
Type (`dtype`)	2	np.float32, np.float64

3 Algorithmes d'optimisation

3.1 Calcul de l'attention

La fonction de calcul de l'attention a été codée selon trois versions :
— Une version NumPy "naïve" servant de référence ;
— Une version Numba avec la compilation JIT, permettant d'accélérer les boucles notamment ;
— Une version combinant Cython et C++ incluant des instructions vectorisées AVX et la multiplication matricielle par blocs.

3.2 Méthodes de benchmark

L'objectif que l'on cherche à minimiser est le temps de calcul requis pour l'attention. Trois algorithmes ont été testés et implémentés pour choisir la combinaison de paramètres permettant de minimiser cet objectif :

- **Un algorithme d'optimisation bayésienne** : celui-ci repose sur un modèle probabiliste de l'objectif, entraîné à chaque itération et permettant de prédire la performance des différentes configurations.
- **Un algorithme de bandit multibras** : celui-ci teste les différentes configurations et calcule pour chacune un score de confiance UCB (upper-confidence bound), prenant en compte la valeur moyenne observée de l'objectif et un terme d'incertitude diminuant avec le nombre d'essais. À chaque nouvelle itération, l'algorithme choisit le bras ayant le score UCB le plus élevé.
- **Un algorithme génétique** : à partir d'une population initiale de configurations générées aléatoirement, évolue sur plusieurs générations afin de trouver rapidement la meilleure combinaison de paramètres. Le fonctionnement se base sur la sélection à chaque génération, des configurations les plus performantes, qui sont croisées et mutées pour explorer l'espace de recherche en profondeur.

Ces algorithmes ont été choisis car ils permettent d'optimiser les paramètres en limitant le nombre d'essais. Ils ont été testés pour différentes dimensions de matrices, notées `dim` (valeurs testées : 64, 128, 256, 400, 512, 768, 1024).

4 Résultats et analyse

Les résultats obtenus sont effectivement meilleurs que pour la grille : on trouve les meilleurs paramètres en 30 essais contre 32 pour la grille.

Le temps de calcul a été calculé pour chacun des algorithmes et chacune des trois fonctions (NumPy, Numba, Cython). Les résultats sont présentés dans le tableau 1. La méthode préférentielle dépend de la taille de la matrice.

Table 1 – Comparaison des temps moyens entre algorithmes

dim	Bandit (s)	Bayes (s)	Génétique (s)	Meilleure méthode
64	2.87e-05	2.87e-05	2.88e-05	bayes
128	2.35e-04	2.16e-04	2.61e-04	bayes
256	1.06e-02	9.08e-04	9.1e-04	bayes
400	2.73e-03	2.01e-03	2.10e-03	bayes
512	4.824e-03	6.17e-03	4.83e-03	bandit
768	1.06e-02	1.05e-02	1.08e-02	bayes
1024	2.58e-02	2.21e-02	2.40e-02	bayes

Le tableau 1 met en évidence :

- La supériorité générale de la recherche bayésienne pour la plupart des dimensions testées.
- Une performance ponctuelle de la stratégie « bandit » pour $d = 512$.
- Une méthode génétique compétitive, notamment pour les tailles intermédiaires, mais jamais la plus rapide.
- Une sensibilité non linéaire des temps de calcul à la dimension d.

Les valeurs détaillées des paramètres choisis et des temps d'exécution pour les algorithmes sont présentés dans les tableaux 3, 4 et 2 :

TABLE 2 – Valeurs optimisées des paramètres et temps moyen d'exécution pour l'algorithme Genetique

dim	dtype	blocksize	threads	NumPy (s)	Numba (s)	Cython (s)
64	float32	64	1	3.11e-05	2.09e-07	2.88e-05
128	float32	8	2	1.61e-04	2.73e-04	2.61e-04
256	float32	64	4	7.14e-04	5.58e-04	9.16e-04
400	float32	16	2	2.31e-03	7.07e-04	2.10e-03
512	float32	32	4	5.38e-03	1.05e-03	4.83e-03
768	float32	32	4	9.54e-03	2.47e-03	1.08e-02
1024	float32	16	2	2.35e-02	5.70e-03	2.41e-02

TABLE 3 – Valeurs optimisées des paramètres et temps moyen d'exécution pour l'algorithme Bandit

dim	dtype	blocksize	threads	NumPy (s)	Numba (s)	Cython (s)
64	float32	24	8	5.52e-05	5.63e-05	3.12e-05
128	float32	16	8	3.11e-04	6.10e-04	1.02e-02
256	float32	32	2	2.84e-03	2.35e-02	5.92e-04
400	float32	16	1	4.19e-03	2.47e-02	2.16e-02
512	float32	32	4	8.44e-03	1.52e-02	1.74e-02
768	float32	64	1	2.08e-02	1.97e-02	1.99e-02
1024	float32	8	4	3.72e-02	3.60e-02	2.84e-02

TABLE 4 – Valeurs optimisées des paramètres et temps moyen d'exécution pour l'algorithme Bayes

dim	dtype	blocksize	threads	NumPy (s)	Numba (s)	Cython (s)
64	float32	47	6	1.29e-03	2.14e+00	9.09e-04
128	float32	41	4	2.11e-03	7.40e-01	2.02e-03
256	float32	63	8	1.02e-02	9.29e-01	1.05e-02
400	float32	11	5	2.01e-02	8.26e-01	2.21e-02
512	float32	26	2	7.50e-03	1.24e+00	6.17e-03
768	float32	63	8	1.02e-02	9.29e-01	1.05e-02
1024	float32	11	5	2.01e-02	8.26e-01	2.21e-02

Le choix du backend le plus performant varie selon l'algorithme et la dimension d :

— **Génétique** : Numba s'impose dès $d \geq 256$, grâce au JIT et à la vectorisation. NumPy reste compétitive uniquement pour $d = 128$, et Cython améliore légèrement NumPy sans rattraper Numba.

— **Bandit** : le meilleur alterne entre Cython ($d = 64, 256, 1024$) et NumPy ($d = 128, 400, 512$), tandis que Numba ne devient avantageux qu'à $d \geq 768$.

— **Bayésien** : Cython est quasi systématiquement le plus rapide, NumPy occupe la deuxième place, et Numba ne prend l'avantage que sur des dimensions intermédiaires où son coût initial de compilation est amorti.

Systématiquement, le choix de float32 s'est imposé par rapport à float64.

5 Conslusion et perspectives

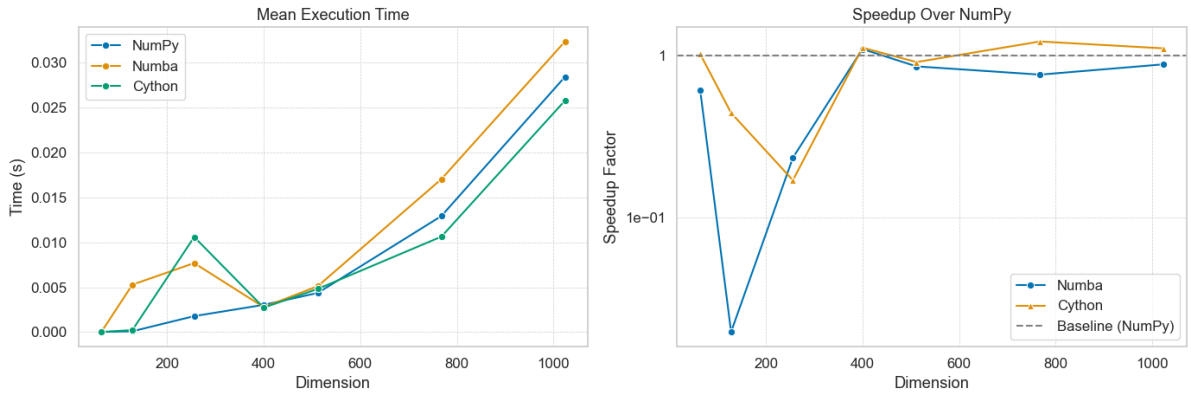

FIGURE 1 – Comparaison des temps d'exécution moyens (gauche) et des speedups sur NumPy (droite) pour l'algorithme Bandit.

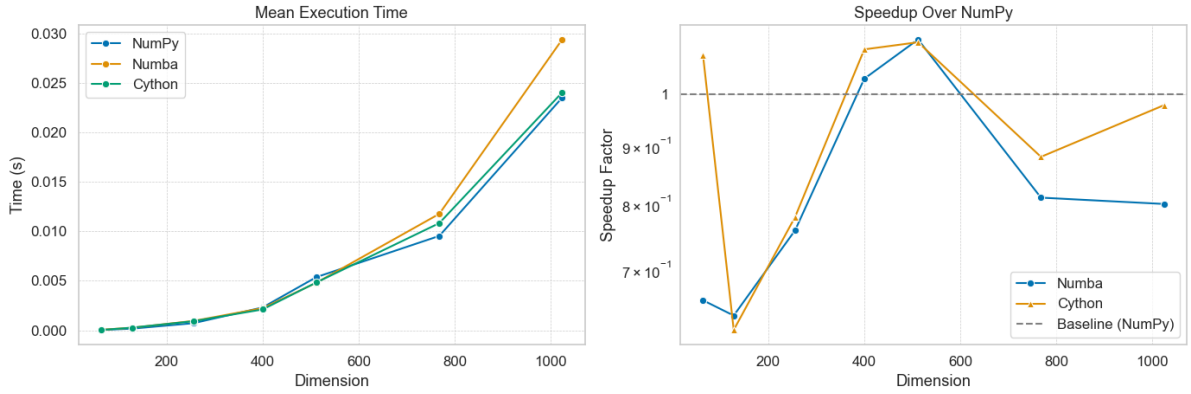

FIGURE 2 – Comparaison des temps d'exécution moyens et des speedups sur NumPy pour l'algorithme Génétique.

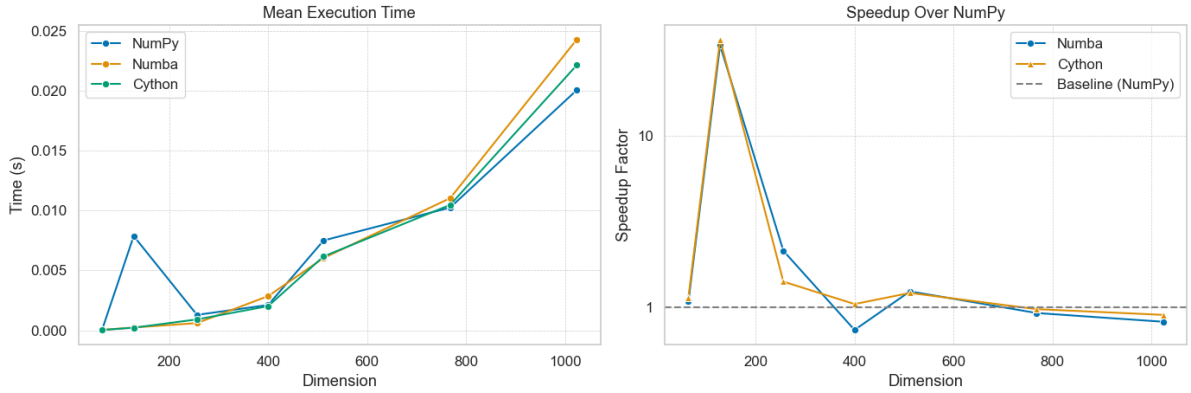

FIGURE 3 – Comparaison des temps d'exécution moyens et des speedups sur NumPy pour l'algorithme Bayésien.

Les figures 1, 2 et 3 montrent clairement que :

— **Bandit :** Cython domine en très basse dimension ($d \leq 256$), NumPy reste compétitif pour $d = 128, 400, 512$ et Numba s'impose dès $d \geq 768$ lorsque le JIT amortit son overhead.
— **Génétique :** Numba prend l'avantage dès $d \geq 256$ via la vectorisation JIT, NumPy l'emporte ponctuellement pour $d = 128$, et Cython améliore systématiquement NumPy mais ne rattrape pas Numba.
— **Bayésien :** En très petite dimension ($d \leq 128$), speedups spectaculaires ($> 10\times$) pour Cython et Numba ; pour $256 \leq d \leq 1024$, Cython reste légèrement devant, Numba est pénalisé par son overhead initial.

Dans ce benchmark adaptatif, la recherche bayésienne s'avère la plus robuste sur un large éventail de tailles, grâce à sa capacité d'exploration efficace. Cython offre un excellent compromis statique, tandis que Numba excelle dès que la complexité et la taille des boucles justifient son coût de compilation. L'algorithme bandit, plus simple à mettre en place, conserve un léger avantage en très basse dimension, et l'approche génétique reste compétitive sur les dimensions intermédiaires.

Pour aller plus loin nous avons imaginé tester d'autres stratégies (grille, évolution différentielle, optimisateurs gradient-based) ou bien étendre l'étude à des fonctions plus coûteuses (convolutions, FFT, réseaux de neurones). On pourrait également intégrer des critères multi-objectifs (temps vs mémoire vs précision) et affiner l'exploration de l'espace de paramètres (recherche hiérarchique).