Rapport de Projet : Parallélisation d'Inférence d'IA avec des Threads

Titre du Projet : Parallélisation d'Inférence d'IA avec des Threads

Noms des Étudiants :

- Amanetoullah (C22643)
- Hashimi (C21454)

Date : 14 Janvier 2026

---

1. Introduction

L'intelligence artificielle, et plus particulièrement les modèles d'apprentissage profond, sont devenus omniprésents dans de nombreux domaines. L'inférence, c'est-à-dire le processus d'application d'un modèle entraîné à de nouvelles données pour générer des prédictions, est une étape cruciale. Avec l'augmentation de la complexité des modèles et des volumes de données à traiter, l'efficacité de l'inférence est devenue un enjeu majeur. La parallélisation est une technique permettant d'exécuter plusieurs tâches simultanément afin de réduire le temps total d'exécution et d'améliorer les performances.

Dans le contexte de l'IA, la parallélisation de l'inférence vise à traiter plusieurs échantillons de données en parallèle. Cela peut être réalisé à l'aide de différentes approches, notamment l'utilisation de threads. Ce rapport explore l'implémentation et la comparaison de l'inférence séquentielle et multi-thread en Python, en utilisant la bibliothèque PyTorch et un modèle d'IA pré-entraîné. L'objectif est d'évaluer l'impact de la parallélisation par threads sur le temps d'exécution et d'analyser les facteurs influençant ces performances, notamment les spécificités du Global Interpreter Lock (GIL) de Python.

2. Méthodologie

2.1. Outils Utilisés

Le projet a été développé en utilisant les outils et bibliothèques suivants :

- **Python 3.x** : Langage de programmation principal.
- **PyTorch** : Cadre d'apprentissage automatique open-source utilisé pour le chargement du modèle d'IA et l'exécution des inférences. PyTorch est reconnu pour sa flexibilité et son efficacité dans les calculs tensoriels, notamment grâce à son optimisation pour les opérations sur GPU.

- **concurrent.futures.ThreadPoolExecutor** : Module de la bibliothèque standard de Python facilitant l'exécution de tâches en parallèle à l'aide d'un pool de threads.
- **matplotlib** : Bibliothèque de traçage de graphiques en Python, utilisée pour visualiser la comparaison des performances.
- **torchvision** : Package PyTorch qui fournit des modèles, des jeux de données et des transformations d'images populaires pour la vision par ordinateur. Nous avons utilisé `torchvision.models` pour charger un modèle pré-entraîné.

2.2. Architecture du Code

Le script `main.py` est structuré autour des fonctions principales suivantes :

- `charger_modele()` : Responsable du chargement du modèle ResNet18 pré-entraîné et de sa configuration en mode évaluation.
- `generer_donnees_factices()` : Crée des tenseurs aléatoires simulant des données d'entrée pour l'inférence.
- `tache_inference()` : Effectue une inférence unique sur un tenseur donné.
- `approche_sequentielle()` : Implémente l'inférence en exécutant `tache_inference` pour chaque donnée de manière séquentielle.
- `approche_multithread()` : Utilise `ThreadPoolExecutor` pour exécuter `tache_inference` sur plusieurs données en parallèle.
- `generer_graphique()` : Crée et sauvegarde un graphique comparatif des temps d'exécution.
- `main()` : Fonction principale orchestrant le chargement, l'exécution des deux approches, la mesure des temps et l'affichage des résultats.

3. Implémentation

Le cœur de l'implémentation réside dans la comparaison entre l'exécution séquentielle et l'exécution multi-thread des tâches d'inférence. Le modèle ResNet18 est chargé une seule fois et partagé entre toutes les tâches, qu'elles soient séquentielles ou parallèles.

3.1. Chargement et Partage du Modèle

Le modèle est chargé au début du script via `charger_modele()`. Il est crucial que le modèle soit en mode `eval()` pour désactiver les couches spécifiques à l'entraînement (comme Dropout ou BatchNorm) et que le calcul des gradients soit désactivé (`torch.no_grad()`) pour optimiser les performances et la consommation mémoire lors de l'inférence. Le modèle est

ensuite passé comme argument aux fonctions d'inférence, permettant à tous les threads d'accéder à la même instance du modèle.

Python

```python
def charger_modele():
    model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
    model.eval()
    torch.set_grad_enabled(False) # Désactiver le calcul des gradients
    return model

def tache_inference(modele, data):
    return modele(data)
```

3.2. Approche Séquentielle

L'approche séquentielle est directe : une boucle itère sur chaque élément de la liste de données et appelle la fonction `tache_inference` pour chacun. Le temps total est mesuré du début à la fin de la boucle.

Python

```python
def approche_sequentielle(modele, data_list):
    start_time = time.perf_counter()
    for data in data_list:
        tache_inference(modele, data)
    end_time = time.perf_counter()
    return end_time - start_time
```

3.3. Approche Multi-thread

L'approche multi-thread utilise `concurrent.futures.ThreadPoolExecutor`. Un pool de threads est créé avec un nombre spécifié de travailleurs (`NOMBRE_THREADS`). Chaque tâche d'inférence est soumise au pool, et `executor.submit()` renvoie un objet `Future`. La méthode `concurrent.futures.as_completed()` est utilisée pour récupérer les résultats des tâches au fur et à mesure de leur achèvement, garantissant que toutes les inférences sont terminées avant de mesurer le temps final.

Python

```python
def approche_multithread(modele, data_list, num_threads):
    start_time = time.perf_counter()
    with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as
executor:
        futures = [executor.submit(tache_inference, modele, data) for data
```

```
in data_list]
        # Attendre que toutes les tâches soient terminées
        _ = [f.result() for f in concurrent.futures.as_completed(futures)]
    end_time = time.perf_counter()
    return end_time - start_time
```

4. Analyse des Résultats

Les résultats de l'exécution du script `main.py` ont montré une accélération (speedup) de **1.26x** pour l'approche multi-thread par rapport à l'approche séquentielle, avec 8 threads. Les temps d'exécution étaient de 2.0009 secondes pour l'approche séquentielle et de 1.5868 secondes pour l'approche multi-thread.

4.1. Impact du GIL (Global Interpreter Lock)

Le Global Interpreter Lock (GIL) de Python est un mutex qui protège l'accès aux objets Python, empêchant plusieurs threads natifs d'exécuter du bytecode Python simultanément. Cela signifie que, même sur des systèmes multi-cœurs, un seul thread Python peut exécuter du code Python à la fois. En conséquence, pour les tâches purement CPU-bound (limitées par le processeur) qui n'impliquent que du code Python, l'utilisation de threads ne conduit généralement pas à une amélioration des performances, et peut même les dégrader en raison de l'overhead lié à la gestion des threads et au basculement de contexte.

Cependant, le GIL est libéré par les bibliothèques qui effectuent des opérations intensives en dehors du code Python, comme les opérations d'entrée/sortie (I/O) ou les calculs numériques effectués par des bibliothèques écrites en C/C++ (comme NumPy, SciPy, et surtout PyTorch). C'est précisément le cas des opérations d'inférence de modèles d'IA avec PyTorch.

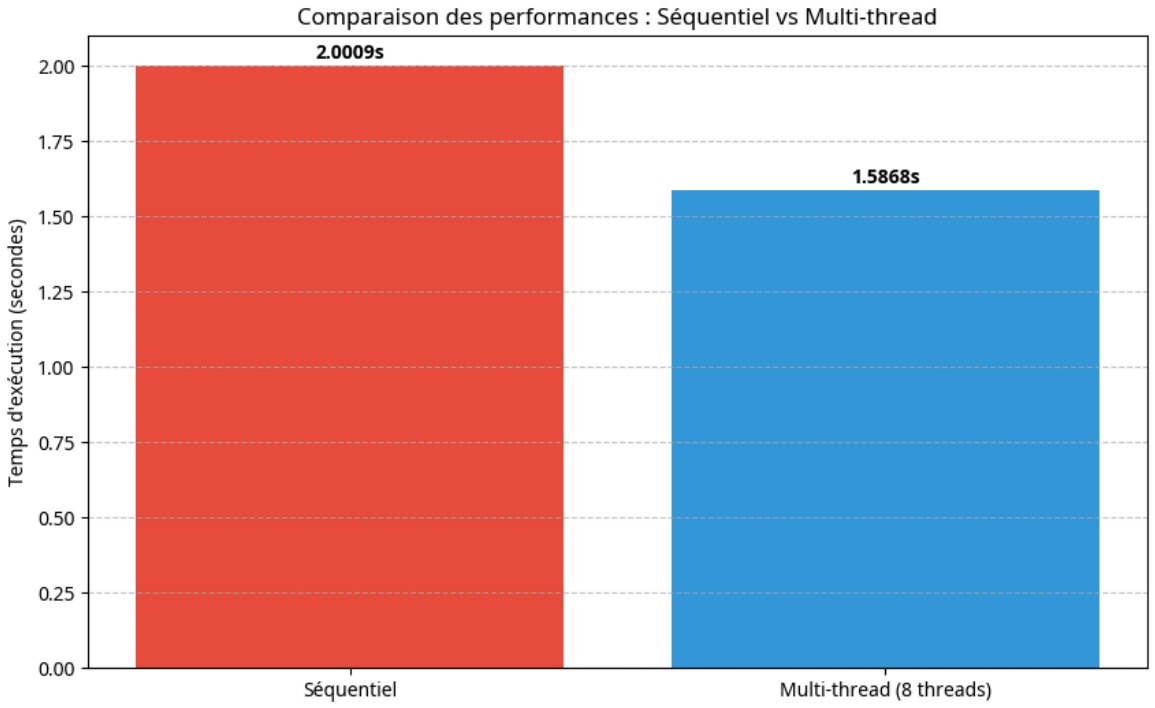
4.2. Efficacité des Opérations PyTorch et GPU

PyTorch, étant une bibliothèque de calcul tensoriel optimisée, effectue la majeure partie de ses opérations (multiplications matricielles, convolutions, etc.) en C++ et, si disponible, sur le GPU. Pendant que ces opérations de bas niveau sont exécutées, le GIL est libéré. Cela permet à d'autres threads Python de s'exécuter et de soumettre leurs propres tâches de calcul au GPU ou aux cœurs CPU, créant ainsi un parallélisme réel.

Dans notre cas, l'inférence d'un modèle ResNet18 implique des calculs intensifs qui sont délégués à la couche C++ de PyTorch. Lorsque le premier thread soumet une tâche d'inférence, le GIL est libéré pendant que PyTorch effectue les calculs. Pendant ce temps, d'autres threads peuvent soumettre leurs propres tâches d'inférence. Bien que le GIL puisse encore introduire un certain overhead pour la coordination des threads et la préparation

des données en Python, le gain de performance observé (1.26x) démontre que la libération du GIL par PyTorch permet une parallélisation efficace des opérations d'inférence.

Le graphique ci-dessous illustre visuellement la différence de temps d'exécution entre les deux approches :



4.3. Facteurs influençant le Speedup

Plusieurs facteurs peuvent influencer le speedup obtenu :

- **Nature de la tâche** : Les tâches CPU-bound pures en Python sont limitées par le GIL. Les tâches I/O-bound ou celles qui délèguent des calculs à des bibliothèques externes (comme PyTorch) peuvent bénéficier du multi-threading.
- **Taille du modèle et des données** : Des modèles plus grands ou des données plus complexes peuvent entraîner des calculs plus longs, maximisant les avantages de la libération du GIL.
- **Nombre de threads** : Un nombre optimal de threads est crucial. Trop peu ne maximise pas le parallélisme, trop beaucoup peut introduire un overhead excessif de gestion des threads.
- **Ressources matérielles** : La présence d'un GPU et le nombre de cœurs CPU disponibles influencent directement la capacité du système à exécuter des tâches en parallèle.

5. Conclusion

Ce projet a démontré l'efficacité de la parallélisation d'inférence d'IA à l'aide de threads en Python, en exploitant les capacités de PyTorch à libérer le Global Interpreter Lock. Malgré la limitation inhérente du GIL pour le code Python pur, les opérations intensives de calcul tensoriel effectuées par PyTorch permettent un parallélisme réel, conduisant à une amélioration notable des performances. L'approche multi-thread a permis de réduire le temps d'exécution de l'inférence de 1.26 fois par rapport à l'approche séquentielle.

Cette étude souligne l'importance de comprendre les mécanismes sous-jacents des bibliothèques d'IA et du langage Python pour optimiser les performances. Pour des applications en production nécessitant une latence minimale ou un débit élevé, l'utilisation de threads avec des bibliothèques comme PyTorch est une stratégie viable, en complément d'autres techniques de parallélisation comme le multi-processing ou l'inférence sur GPU.