Université Mohammed V - Rabat
École Nationale Supérieure d'Informatique et d'Analyse des Systèmes

Rapport du Projet du calcul parallèle

Filière

Ingénierie de l'Intelligence Artificielle (2IA)

Sujet :

Accélération GPU avec CUDA et Analyse des Gains de Performance

---

Réalisé par :

ZIYAD ABIDATE

Encadré par :

M.NOUREDDINE KERZAZI

Année Universitaire 2024-2025

Table des matières

Résumé

Ce rapport présente une étude approfondie sur l'accélération des modèles d'apprentissage profond à l'aide des technologies GPU. Nous explorons différentes techniques, notamment l'utilisation de CUDA, la parallélisation avancée et la précision mixte. À travers l'analyse de six codes sélectionnés depuis Kaggle, nous démontrons comment ces méthodes permettent d'améliorer significativement la vitesse d'exécution et la performance globale des modèles.

0.1 Introduction et Contexte Général

L'augmentation de la complexité des modèles d'apprentissage automatique et la taille croissante des jeux de données nécessitent des ressources informatiques de plus en plus puissantes. Les processeurs graphiques (GPU) se sont imposés comme des outils essentiels pour l'entraînement et l'inférence de ces modèles en raison de leur capacité à effectuer des opérations matricielles massivement parallèles.

Dans ce contexte, l'utilisation de techniques d'accélération GPU devient cruciale pour réduire les temps d'entraînement et rendre les expérimentations plus efficaces. Le but de ce rapport est d'explorer différentes stratégies pour optimiser l'exécution de modèles d'apprentissage profond sur GPU, à travers six codes sélectionnés depuis Kaggle.

0.2 Exécution sur GPU et Importance

L'exécution sur GPU repose principalement sur le traitement parallèle de grandes quantités de données. Contrairement aux CPU, conçus pour des tâches séquentielles, les GPU permettent l'exécution simultanée de milliers de threads, rendant leur utilisation particulièrement adaptée aux calculs massivement parallèles rencontrés dans l'apprentissage profond.

Les avantages principaux de l'exécution sur GPU sont :

— **Accélération des calculs** : Réduction significative des temps d'entraînement.
— **Gestion de grands volumes de données** : Traitement efficace de jeux de données volumineux.
— **Optimisation de l'utilisation des ressources** : Utilisation efficace de CUDA et des bibliothèques comme cuDNN.
— **Précision Mixte et XLA** : Réduction de la charge mémoire et amélioration des performances.

Les sections suivantes présenteront, code par code, l'approche adoptée pour accélérer les modèles sélectionnés.

Référencement des projets Kaggle sélectionnés

Dans le cadre de cette étude, nous avons sélectionné six projets issus de la plateforme Kaggle, une référence majeure dans la communauté de la data science et du machine learning. Ces projets ont été choisis en raison de leur pertinence dans l'application de techniques d'accélération de l'entraînement de modèles sur GPU, notamment via CUDA, la parallélisation multi-GPU et l'utilisation de bibliothèques optimisées telles que TensorFlow et PyTorch.

Les projets sélectionnés sont les suivants :

— **Projet 1 :** Twitter sentiment Extaction-Analysis,EDA and Model .
— **Projet 2 :** Intro to Recurrent Neural Networks LSTM — GRU.
— **Projet 3 :** Time Series Modelling Google Stocks: Python LSTM.
— **Projet 4 :** Diabetes:ANN tuning .
— **Projet 5 :** Guava Disease Classification Using CNN.
— **Projet 6 :** Quantum-Enhanced Convolutional Neural Networks.

Chaque projet a été rigoureusement testé et modifié dans le cadre de cette étude afin d'optimiser les performances d'entraînement sur GPU. Les résultats et analyses associés

sont présentés dans les sections suivantes.

0.3 Présentation des projets et optimisation GPU

Dans cette section, chaque projet sélectionné est détaillé individuellement. Pour chaque projet, nous présentons le contexte initial, les défis de performance rencontrés, les solutions d'accélération GPU proposées et les résultats obtenus après optimisation.

0.4 Optimisation de l'entraînement des modèles NER pour l'extraction de sentiments

Le projet consiste en l'entraînement de modèles de reconnaissance d'entités nommées (NER) pour l'extraction de sentiments à partir de tweets. L'objectif est d'identifier automatiquement les portions d'un texte reflétant le sentiment exprimé (positif, négatif ou neutre).

Problématiques identifiées

Les principales limitations du code de base sont les suivantes :

- **Complexité temporelle élevée :** L'entraînement du modèle SpaCy est réalisé de manière séquentielle, ce qui allonge significativement le temps de traitement, surtout pour des jeux de données volumineux.
- **Exploitation limitée des ressources :** Le code d'origine n'exploite pas les capacités de calcul parallèle ni les accélérateurs matériels comme le GPU.
- **Taille des mini-lots :** La taille des mini-batches est fixée de manière sous-optimale, ce qui peut ralentir la convergence du modèle.
- **Gestion de la complexité des pipelines SpaCy :** Le pipeline NER est utilisé seul, mais sans désactivation explicite des autres composants inutiles.

Approches proposées pour l'accélération

Pour remédier aux problématiques identifiées, plusieurs techniques d'accélération ont été mises en place :

- **Utilisation du GPU (CUDA) :** Le modèle est transféré vers le GPU (`torch.cuda`) pour accélérer les calculs matriciels massifs.
- **Entraînement parallèle :** Utilisation de `multiprocessing.Pool` pour entraîner simultanément plusieurs modèles sur différents ensembles de données (sentiments positifs et négatifs).
- **Optimisation des mini-lots :** Adoption d'une taille de mini-batch croissante (compounding) pour un compromis entre performance et précision.
- **Réduction du dropout :** Ajustement du taux de régularisation (dropout) pour accélérer la convergence sans perte de performance.
- **Désactivation des composants non essentiels :** Désactivation explicite des autres pipelines SpaCy non utilisés.

Résultats et performances

Les résultats de performance temporelle obtenus avant et après l'optimisation sont les suivants :

- **Temps d'exécution avant optimisation :** Environ 138.47 secondes pour un entraînement séquentiel.
- **Temps d'exécution après optimisation (parallèle et GPU) :** 3.32 secondes, soit une réduction de plus de 40x.

Potentiel d'amélioration supplémentaire

D'autres pistes d'amélioration ont été identifiées pour optimiser davantage l'entraînement du modèle :

- **Quantization des modèles :** Réduction de la précision des poids pour un modèle plus léger.
- **Distillation de modèle :** Entraînement d'un modèle plus léger basé sur le modèle initial.
- **Optimisation des bibliothèques :** Migration vers des frameworks comme `PyTorch Lightning` pour une gestion avancée des ressources.

0.5 Optimisation de l'entraînement d'un modèle "Recurrent Neural Networks" (LSTM) pour la prédiction des prix boursiers IBM

Le projet consiste à entraîner un réseau de neurones récurrent de type LSTM (Long Short-Term Memory) pour prédire les prix des actions de la société IBM en utilisant des données historiques boursières. L'objectif est de modéliser la dynamique temporelle des prix et de minimiser l'erreur de prédiction.

Problématiques identifiées

Les limitations du code de base initial sont les suivantes :

— **Temps d'exécution long :** L'entraînement de l'architecture LSTM avec 50 époques prend environ 847.93 secondes.
— **Non-utilisation du GPU :** Le code initial n'exploitait pas les ressources de calcul accéléré par GPU.
— **Optimisation du prétraitement :** La gestion des séquences temporelles et du prétraitement pouvait être améliorée.
— **Optimisation de l'optimiseur :** Utilisation de RMSProp au lieu de l'optimiseur Adam, plus rapide pour ce type de données temporelles.

Approches proposées pour l'accélération

Les techniques d'accélération mises en place incluent :

— **Activation du GPU (CUDA) :** Utilisation explicite de TensorFlow pour allouer dynamiquement la mémoire GPU.
— **Optimisation du prétraitement :** Ajustement du découpage des données et normalisation via MinMaxScaler.
— **Optimiseur Adam :** Remplacement de RMSProp par Adam pour une convergence plus rapide.
— **Structure inchangée mais parallélisée :** Conservation de l'architecture LSTM à 4 couches tout en utilisant les ressources GPU.

Résultats et performances

Les résultats de performance avant et après optimisation sont les suivants :

— **Temps d'entraînement avant optimisation :** 847.93 secondes.
— **Temps d'entraînement après optimisation (GPU) :** 53.97 secondes.
— **Réduction du temps :** Un facteur de réduction de **15.7x**.
— **RMSE (Erreur Quadratique Moyenne) :** Réduction à environ 5.82, confirmant la stabilité du modèle optimisé.

Potentiels d'amélioration supplémentaires

Des pistes d'amélioration futures incluent :

— **Utilisation de réseaux Bidirectionnels (BiLSTM) :** Pour capturer des dépendances temporelles dans les deux directions.
— **Quantization :** Réduction de la précision des poids pour accélérer l'inférence.
— **Early Stopping :** Arrêt anticipé de l'entraînement en cas de convergence.

0.6 Optimisation de l'Entraînement d'un Modèle LSTM (Modelling Google Stocks)

0.6.1 Introduction au Projet

Ce projet explore l'optimisation de l'entraînement d'un modèle LSTM pour la prédiction des prix d'actions Google à partir de séries temporelles. Le modèle utilise des réseaux neuronaux récurrents (RNN) de type LSTM (Long Short-Term Memory) en raison de leur capacité à modéliser des données séquencielles et temporelles de manière efficace.

0.6.2 Complexité et Problématiques Identifiées

L'entraînement d'un modèle LSTM est coûteux en termes de temps de calcul et de ressources matérielles, principalement en raison :

— De la taille croissante des jeux de données temporels.
— De la complexité des calculs de gradients dans les architectures récurrentes.
— Des limitations du calcul séquenciel sur CPU.

Pour réduire ces contraintes, des techniques avancées d'accélération GPU ont été explorées.

0.6.3 Approches d'Accélération et Performances

Trois approches principales ont été testées :

— **Précision Mixte et Multi-GPU :** Activation de la précision mixte (`mixed_float16`) et distribution sur plusieurs GPU.
— **Optimiseur Nadam et EarlyStopping :** Utilisation de l'optimiseur Nadam et ajout d'un callback d'arrêt précoce pour limiter le sur-entraînement.
— **Augmentation de la Taille de Batch :** Augmentation progressive de la taille de batch pour maximiser l'utilisation des ressources GPU.

0.6.4 Approche Initiale : Entraînement sur CPU

L'entraînement initial du modèle LSTM a été effectué sur un processeur CPU en utilisant un jeu de données temporel. Le modèle LSTM comportait : - Deux couches LSTM de 50 unités chacune - Deux couches de Dropout de taux 20% - Une couche Dense de sortie

Le modèle a été entraîné sur un jeu de données avec un time step de 30 jours et un batch size de 32 sur 100 époques. Le temps d'exécution mesuré était de :

— **Temps d'entraînement sur CPU : 303.00 secondes**

0.6.5 Première Optimisation : Utilisation de la Précision Mixte et Multi-GPU

La première optimisation a consisté à activer la précision mixte (`mixed_float16`) et à distribuer l'entraînement sur plusieurs GPU via `tf.distribute.MirroredStrategy`. Les modifications apportées incluent : - Activation de la précision mixte pour accélérer les calculs. - Distribution des calculs sur plusieurs GPU. - Utilisation de l'objet `tf.data.Dataset` pour une gestion plus efficace des données avec un batch size de 100.

Le temps d'exécution a été réduit de manière significative :

— **Temps d'entraînement sur GPU : 64.80 secondes**

0.6.6 Deuxième Optimisation : Ajout d'un Arrêt Précoce et Optimiseur Nadam

Pour améliorer encore la performance et réduire le temps d'exécution, les changements suivants ont été apportés : - Remplacement de l'optimiseur Adam par Nadam. - Ajout d'un callback d'arrêt précoce (`EarlyStopping`). - Augmentation de la taille de batch à 200.

Le temps d'entraînement a été considérablement réduit :

— **Temps d'entraînement sur GPU : 21.91 secondes**

0.6.7 Troisième Optimisation : Augmentation de la Taille de Batch

Pour maximiser l'utilisation des ressources GPU, une augmentation de la taille de batch à 500 a été testée, tout en conservant les configurations précédentes : - Utilisation de la précision mixte. - Optimiseur Nadam. - Early Stopping.

Le temps d'entraînement a encore été réduit :

— **Temps d'entraînement sur GPU : 12.09 secondes**

0.6.8 Comparaison des Temps d'Exécution

Approche	**Temps (secondes)**
CPU Basique	303.00
Précision Mixte + Multi-GPU	64.80
Optimisation Nadam + EarlyStopping	21.91
Augmentation Batch Size (500)	12.09

0.6.9 Conclusion

Cette étude a démontré que l'utilisation conjointe des techniques avancées d'accélération GPU permet une réduction significative du temps d'entraînement des modèles LSTM. L'application de la précision mixte, de la distribution multi-GPU et de l'utilisation d'un optimiseur avancé comme Nadam a permis de réduire le temps d'exécution de 303.00 secondes à seulement 12.09 secondes. Ces résultats soulignent l'importance des choix de configuration matérielle et logicielle dans l'optimisation des modèles de deep learning, rendant les expérimentations plus efficaces et accessibles pour des jeux de données volumineux.

0.7 Optimisation d'un CNN pour classifcation des maladies du goyavier (Guava Disease Classification Using CNN)

0.7.1 Introduction du Projet

Ce projet se concentre sur l'entraînement d'un modèle CNN (Convolutional Neural Network) appliqué à la classification des maladies de la plante goyavier . Le jeu de données utilisé contient un ensemble d'images pour prédire si une plante est malade ou pas. L'objectif est de réduire le temps d'entraînement tout en maintenant une précision optimale du modèle.

0.7.2 Complexité et Problématiques Identifiées

L'entraînement des réseaux de neurones convolutifs (CNN) pour des problèmes de classification peut être extrêmement coûteux en termes de temps de calcul et de ressources. Les principaux défis identifiés sont :

- La complexité du modèle ANN avec plusieurs couches , entrainant une grande quantité de calculs matriciels.
- La gestion inefficace de la mémoire et des données d'entráinement.
- L'utilisation non optimale des ressources GPU disponibles.

0.7.3 Approche Initiale : Entraînement sur CPU

L'entraînement initial du modèle CNN a été effectué sur un processeur CPU . Le modèle CNN était composé des couches suivantes :

- Trois couches convolutives avec 32, 64 et 128 filtres respectivement, une activation ReLU et un pooling 2x2.
- Une couche Flatten suivie de trois couches Dense (128, 64 et 32 neurones) avec activation ReLU.
- Une couche de sortie Dense avec activation softmax.

Le modèle a été entraîné sur un batch size de 64 avec un total de 10 époques. Le temps d'exécution mesuré à été :

- **Temps d'entraînement sur CPU : 3240.00 secondes**

0.7.4 Première Optimisation : Utilisation du GPU

La première optimisation a consisté à activer l'utilisation du GPU pour accélérer les calculs sans modification du code initial. Cette simple activation a permis de réduire considérablement le temps d'exécution :

- **Temps d'entraînement sur GPU : 131.00 secondes**

0.7.5 Deuxième Optimisation : Gestion Optimisée de la Mémoire et Multi-GPU

Pour améliorer davantage les performances, les optimisations suivantes ont été apportées :

— Activation de la stratégie `tf.distribute.MirroredStrategy` pour l'utilisation de plusieurs GPU.
— Gestion avancée de la mémoire via `tf.config.experimental.set_memory_growth`.
— Optimisation du chargement des données avec la préfetching et le caching (`tf.data.Dataset`).

Cette stratégie a permis une réduction significative du temps d'entraînement :

— **Temps d'entraînement avec gestion avancée du GPU : 61.81 secondes**

0.7.6 Comparaison des Temps d'Exécution

Approche	**Temps (secondes)**
CPU Basique	3240.00
Utilisation du GPU	131.00
Multi-GPU et Optimisation Mémoire	61.81

Table 1 – Comparaison des Temps d'Exécution du Modèle CNN

0.7.7 Conclusion

L'utilisation du GPU, puis l'optimisation multi-GPU et de gestion de mémoire, ont permis une amélioration significative des performances du modèle CNN. Le temps total d'entraînement a été réduit de 3240.00 secondes à 61.81 secondes, démontrant l'efficacité des techniques de parallélisation et de préfetching des données.

0.8 Classification du Diabète avec Réseau de Neurones Artificiels et Optimisation d'Hyperparamètres

Ce projet vise à développer un modèle d'intelligence artificielle basé sur des réseaux de neurones artificiels (ANN) pour la classification binaire des patients atteints de diabète (*positif* ou *négatif*). Le modèle intègre deux phases principales :

- Une première phase effectue une recherche d'hyperparamètres (Grid Search) pour déterminer les meilleurs paramètres du modèle, notamment le nombre de couches cachées, les unités par couche, le taux de dropout et le taux d'apprentissage.
- Une deuxième phase utilise ces hyperparamètres optimaux pour construire et entraîner le meilleur réseau de neurones.

L'exécution du code sur un CPU de base a révélé la complexité computationnelle importante de cette méthode, avec un temps de recherche de 1324 secondes et un temps d'entraînement de 41 secondes. Pour accélérer ces calculs, plusieurs approches d'optimisation ont été mises en œuvre, notamment l'utilisation de la précision mixte (*mixed precision*), la stratégie multi-GPU et le compilateur XLA.

- **Précision Mixte :** Activation de la précision `mixed_float16` pour réduire l'utilisation de la mémoire et accélérer les calculs.
- **Optimisation des Données :** Utilisation de la bibliothèque `tf.data` pour un chargement de données plus efficace.
- **Compilation XLA et Multi-GPU :** Activation du compilateur XLA et utilisation de la stratégie `MirroredStrategy` pour l'entraînement parallèle.

Les performances des différentes approches sont résumées dans le tableau ci-dessous :

Approche	**Temps de recherche (s)**	**Temps d'entraînement (s)**
CPU de base	1324	41
Précision mixte	1165.91	18.80
Multi-GPU + XLA	281.06	6.47

Table 2 – Comparaison des performances des approches d'optimisation.

Ces résultats montrent une amélioration significative des temps de calcul grâce aux optimisations. L'utilisation conjointe de la précision mixte et du multi-GPU avec XLA a permis une réduction drastique des temps de recherche et d'entraînement, rendant cette approche particulièrement adaptée pour des applications nécessitant des calculs intensifs.

0.9 Accélération de l'Entraînement d un Quantum-Enhanced CNN

0.9.1 Introduction

La présente partie traite de l'amélioration des performances d'entraînement d'un modèle Quantum-Enhanced Convolutional Neural Network (QECNN). L'entraînement initial de ce modèle était limité par la complexité de la phase de prétraitement et l'exécution des circuits quantiques simulés, nécessitant une optimisation pour réduire le temps de calcul.

0.9.2 Contexte et Problématique

Les réseaux de neurones convolutifs améliorés quantiquement (QECNN) utilisent des circuits quantiques pour extraire des caractéristiques avancées des images. Cependant, la complexité du calcul et la simulation des circuits quantiques entraînent un temps d'entraînement prolongé. Ce projet vise à explorer l'impact de techniques d'accélération sur l'entraînement de ce modèle.

0.9.3 Performance sans Accélération

Sans optimisation, les performances observées lors de l'entraînement initial sont les suivantes :

— **Modèle Quantique :** 28.14 secondes
— **Modèle Classique :** 25.61 secondes

La différence entre les deux est principalement due au coût de la simulation des circuits quantiques, nécessitant une parallélisation plus efficace.

0.9.4 Optimisations et Accélération

Plusieurs modifications ont été apportées au code initial :

— **Parallélisation Optimisée :** Utilisation d'un calcul du nombre optimal de threads basé sur la mémoire disponible.
— **Prétraitement Optimisé :** Réduction de la taille des lots lors du prétraitement des images.
— **Gestion de la Mémoire :** Libération explicite de la mémoire avec des appels à `gc.collect()` et `clear_session()`.
— **Modèle Allégé :** Réduction des couches cachées et des paramètres du modèle quantique.

0.9.5 Performances après Accélération

Après application des optimisations :

— **Modèle Quantique :** 18.73 secondes
— **Modèle Classique :** 20.15 secondes

L'amélioration est notable, avec une réduction du temps de calcul d'environ 33%.

0.9.6 Conclusion

L'accélération de l'entraînement d'un modèle QECNN peut être significativement améliorée par l'utilisation de stratégies de parallélisation et de gestion de la mémoire. D'autres optimisations telles que la réduction de la complexité des circuits quantiques et l'utilisation de bibliothèques spécialisées comme TensorFlow Quantum pourraient encore améliorer les performances.

0.10 Conclusion

Ce projet a permis d'explorer de manière approfondie les techniques d'accélération des modèles d'apprentissage profond sur GPU. En utilisant des technologies telles que CUDA, la parallélisation avancée et la précision mixte, nous avons montré comment l'optimisation du traitement parallèle peut significativement améliorer les performances de calcul.

L'analyse de six projets Kaggle a permis de mettre en évidence l'impact concret de ces techniques sur la vitesse d'entraînement et l'efficacité globale des modèles d'apprentissage profond. Chaque projet a illustré des approches distinctes d'optimisation, adaptées à des cas d'utilisation variés, allant de la classification d'images à l'analyse séntimentale et la prédiction de séries temporelles.

En conclusion, l'utilisation stratégique des technologies GPU pour l'entraînement de modèles d'apprentissage profond représente un levier essentiel pour réduire les temps de calcul et permettre des expérimentations plus complexes et de plus grande envergure. Ces travaux ouvrent la voie à des recherches futures sur l'optimisation de modèles encore plus sophistiqués et l'exploration d'architectures parallèles innovantes.