PROJET DE PROGRAMMATION PARALLELE AVEC CUDA

Réalisé par :

Kossi Robert MESSAN

Année-scolaire :2024/2025

Introduction

Pour ce projet, nous visons à mettre en pratique les notions vues en cours. Pour ce faire, nous allons implémenter l'algorithme des Kmeans sur CUDA et ensuite sur CPU et faire une analyse comparative des performances et temps d'exécution des deux approches. Nous allons également étudier l'impact des différents paramètres sur les performances de l'algorithme implémenté sur CUDA.

Le projet est hébergé sur github.com via le lien https://github.com/robertmessan/k_means_CUDA . Un fichier readme.md détaillant toutes les étapes pour reproduire l'expérience y est également disponible.

I. Explication de l'algorithme des Kmeans :

L'algorithme des kmeans est l'un des algorithmes de clustering les plus populaires. Cet algorithme non-supervisé consiste à regrouper selon un critère de similarité, une grande quantité de données en plusieurs(k) sous-ensembles appelés clusters.

Les éléments contenus dans les clusters sont similaires les uns aux autres, mais différents des éléments des autres clusters.

Le fonctionnement de l'algorithme est assez intuitif et se fait selon les étapes suivantes :

- Initialisation des K centres de clusters : après avoir défini le nombre de cluster K, il faut les initialiser soit aléatoirement, soit

avec des méthodes avancées. La qualité du clustering dépend souvent de cette initialisation.

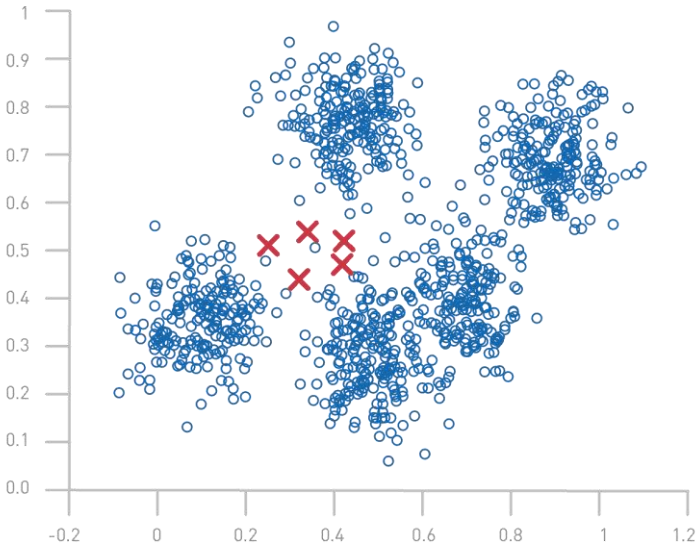

fig1 : initialisation des K centres de clusters

- **Assignation des points à un centre de cluster :** ici, il s'agit d'affecter chaque point du dataset au cluster le plus proche. On utilise souvent la notion de distance pour trouver le cluster le plus proche de chaque point.

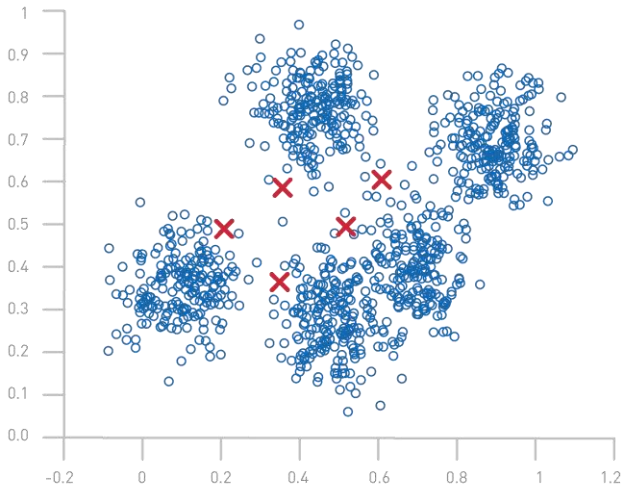

Fig2 : Assignation des points à un centre de cluster

- **Mise à jour des centres de clusters :** pour cette étape, on va recalculer les centres de clusters pour qu'il représentent mieux les données du cluster(centroïd). Cette étape inclut le calcul de moyenne sur les point du cluster. C'est d'ailleurs pourquoi l'algorithme est nommé « kmeans »

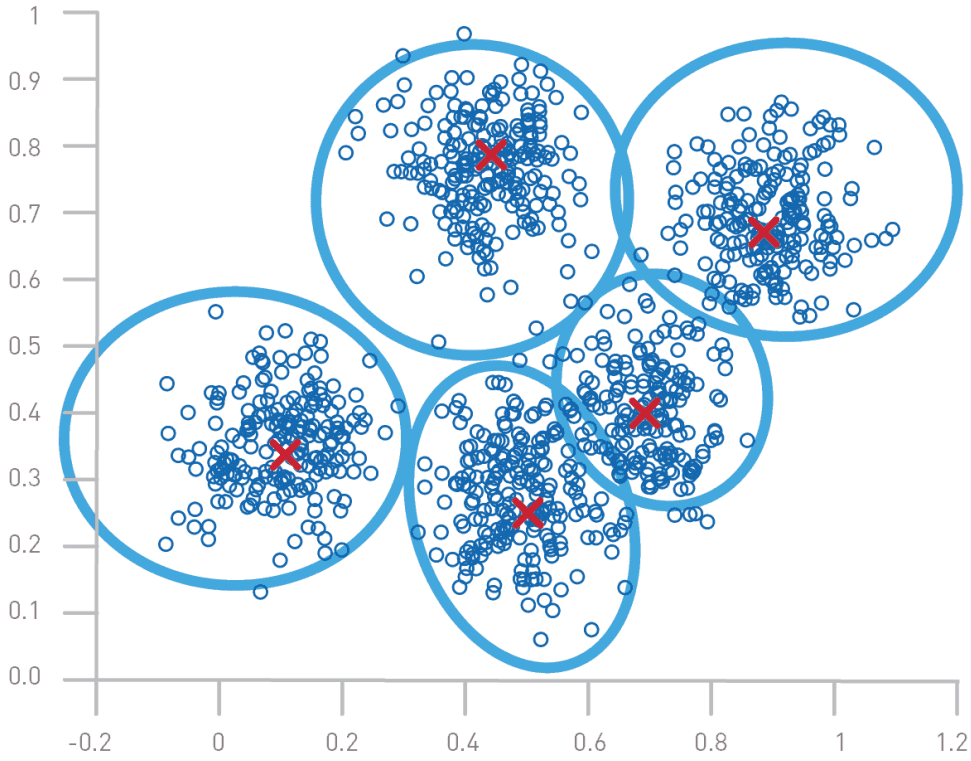

Fig3 : Mise à jour des centres de clusters

Après cette étape, il faut répéter les deux dernière étapes jusqu'à convergence (les clusters restent sensiblement les mêmes d'une itération à une autre).

Pour notre cas d'usage, vu qu'on a un jeu de données de plus de 50000 images, nous allons diviser nos données en batchs avant d'appliquer l'algorithme des kmeans. Cette approche est nommée miniBatchKmeans qui n'est rien d'autre que kmeans exécuté sur les morceaux de notre dataset.

II. Les parties de l'algorithme qui bénéficient d'un parallélisme

Plusieurs parties de l'implémentation de l'algorithme des kmeans sur CUDA bénéficient du parallélisme.

1. Calcul des distances entre les points et les centroïdes

- **Fonction :** Kmeans_find_nearest_cluster
- **Parallélisation :**
  - Chaque thread traite une combinaison point-centroïd
  - Cela permet de calculer simultanément les distances de tous les points à tous les centroïds, ce qui est une opération coûteuse en temps dans un algorithme implémenté sur CPU.

2. Assignation des points au cluster le plus proche

- **Fonction :** assignCluster
- **Parallélisation :**
  - Chaque thread traite un point pour déterminer à quel cluster il appartient.
  - Utilisation d'opérations atomiques pour mettre à jour les tailles des clusters (atomicAdd) et accumuler les contributions des points au calcul des nouveaux centroïds.

3. Recalcul des centroïds

- **Fonction :** recalculate_centre
- **Parallélisation :**
  - Les threads calculent simultanément les nouvelles coordonnées des centroïds.
  - Cette étape est divisée entre les threads pour éviter les goulets d'étranglement.

4. Initialisation des données sur GPU

- **Fonctions :** deviceFill et deviceCopy
- **Parallélisation :**
  - deviceFill : Remplit des tableaux en parallèle.
  - deviceCopy : Copie des données entre tableaux en mémoire partagée du GPU.

III. Le jeu de données CIFAR-10

Pour tester notre implémentation, nous avons utilisé le jeu de données CIFAR-10 qui contient 10 catégories d'images labellisées. On n'utilisera pas directement les labels, mais ça nous servira à évaluer notre clustering avec des métriques supplémentaires comme l'accuracy.

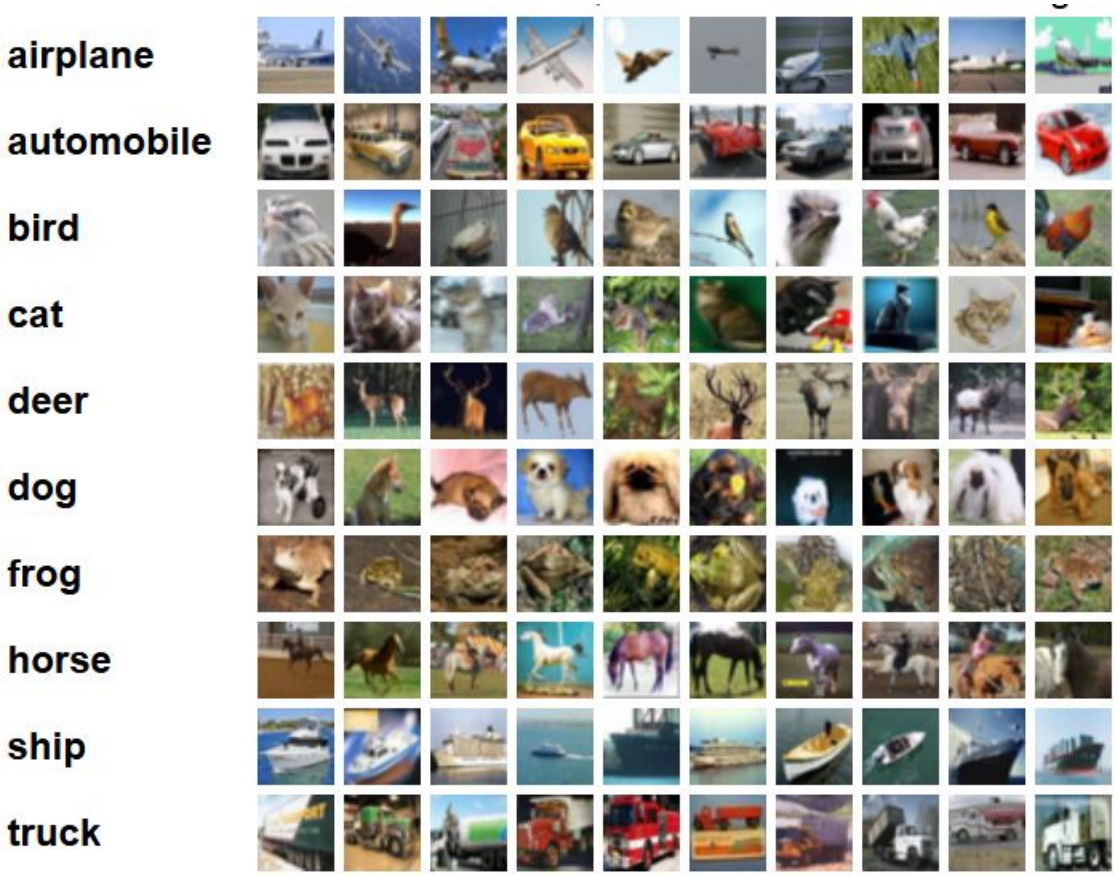

Fig4 : Jeu de données cifar-10

- Nous allons comparer nos deux implémentations sur CPU et CUDA en utilisant l'indice de Davies-Boulding qui indique à quel point les clusters sont compacts et bien séparés les uns des autres.

- Les autres points de comparaison seront principalement le temps d'exécution et l'accuracy qui est secondaire (adapté au cas supervisé).

IV. Comparaison de l'implémentation avec la baseline (CPU)

Pour cette étape, nous avons implémenté l'algorithme des kmeans sur CPU par rapport auquel nous comparons notre implémentation.

- Les résultats de l'algorithme sur CUDA :

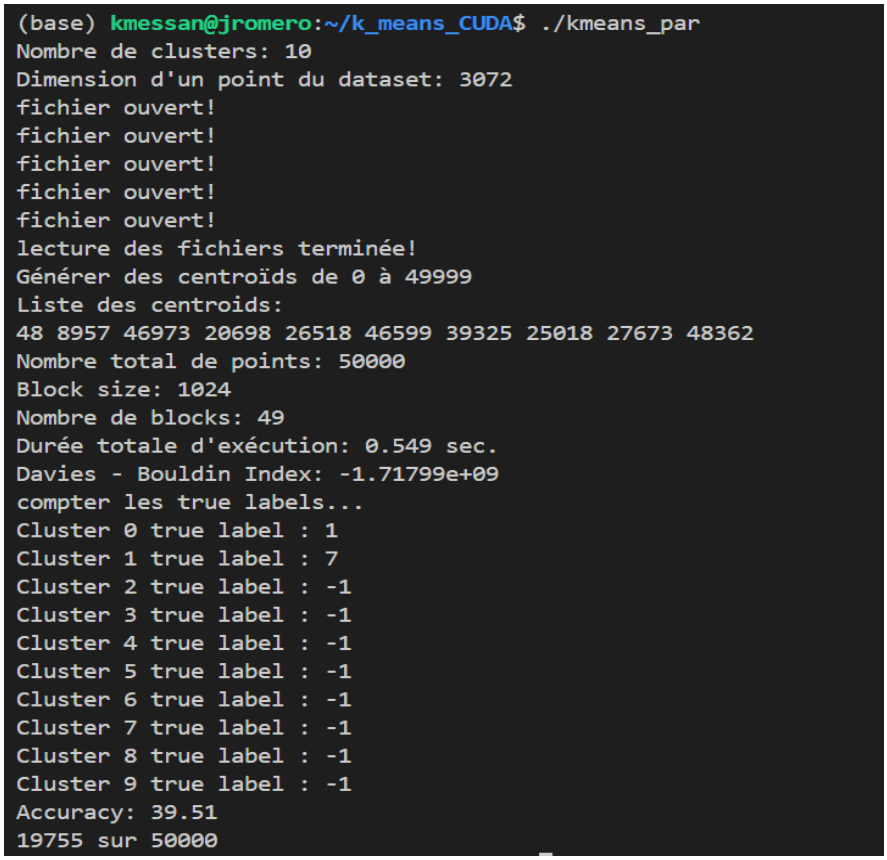

Fig5 : Résultats de l'implémentation sur CUDA

Pour ces résultats, nous pouvons dire que le temps d'exécution de notre code est très bas par rapport à la quantité des données traitée. L'indice de Davies-Boulding est faible, mais reste encore discutable puisqu'il y a encore 8 clusters qui n'ont pas de classes majoritaires(true label : -1), donc mal regroupés. D'autre part, l'accuracy (la proportion des images correctement affectées aux

cluster) montre que le clustering n'est pas forcément parfait. Nous ferons une analyse comparative avec l'implémentation CPU avant de tirer une conclusion.

- Les résultats de l'algorithme sur CPU :

```
Exécution terminée!(base) kmessan@jromero:~/k_means_CUDA$ ./kmeans_seq
Nombre de clusters: 10
Data point dimension: 3072
fichier ouvert!
fichier ouvert!
fichier ouvert!
fichier ouvert!
fichier ouvert!
Lecture des fichiers terminée!
générer des centroïds de 0 à 49999
Liste des centroids:
48 8957 46973 20698 26518 46599 39325 25018 27673 48362
Nombre total des points: 50000
Durée totale d'exécution: 151.824 sec.
Cluster 0 true label : 0
Cluster 1 true label : 5
Cluster 2 true label : 9
Cluster 3 true label : 4
Cluster 4 true label : 1
Cluster 5 true label : 6
Cluster 6 true label : 7
Cluster 7 true label : 0
Cluster 8 true label : 0
Cluster 9 true label : 0
Cluster 10 true label : 0
Accuracy: 39.662
19831 sur 50000
Davies - Bouldin Index: -6.44245e+08
```

Fig6 : Résultats de l'implémentation sur CPU

Par rapport aux résultats de CUDA, nous avons une légère amélioration de l'accuracy, presque 6 clusters sont bien représentés, mais l'indice de davies-boulding demeure pire que le cas précédent. En ce qui concerne le temps d'exécution, les résultats montrent que l'implémentation sur CPU est beaucoup plus lente que celle sur CUDA.

Caractéristique	Séquentiel (kmeans_seq)	Parallèle (kmeans_par)

Nombre de clusters	10	10
Dimension des points	3072	3072
Nombre total de points	50 000	50 000
Durée d'exécution	151.824 sec	0.549 sec
Index Davies-Bouldin	-6.44245e+08	-1.71799e+09
Précision (Accuracy)	39.662% (19 831/50 000)	39.51% (19 755/50 000)

Fig6 : Tableau comparatif des implémentations CPU et CUDA

V. Optimisation du code CUDA

Pour cette partie, nous avons pensé à utiliser des mémoires partagées dans les fonctions ou méthodes *Kmeans_find_nearest_cluster, assignCluster* et *recalculate_centre.* Nous avons procédé également à la réorganisation des accès mémoire dans ces fonctions, ce qui a significativement améliorer le temps d'exécution et l'indice de Davies-Boulding comme on peut le voir sur l'image suivante :

```
(base) kmessan@jromero:~/k_means_CUDA$ ./shared_kmean_par
Nombre de clusters: 10
Dimension d'un point du dataset: 3072
fichier ouvert!
fichier ouvert!
fichier ouvert!
fichier ouvert!
fichier ouvert!
lecture des fichiers terminée!
Générer des centroïds de 0 à 49999
Liste des centroids:
48 8957 46973 20698 26518 46599 39325 25018 27673 48362
Nombre total de points: 50000
Block size: 1024
Nombre de blocks: 49
Durée totale d'exécution: 0.263 sec.
Davies - Bouldin Index: -1.93274e+09
compter les true labels...
Cluster 0 true label : 7
Cluster 1 true label : -1
Cluster 2 true label : -1
Cluster 3 true label : -1
Cluster 4 true label : -1
Cluster 5 true label : -1
Cluster 6 true label : -1
Cluster 7 true label : -1
Cluster 8 true label : -1
Cluster 9 true label : -1
Accuracy: 39.484
19742 sur 50000
```

Fig6 : Optimisation du code CUDA avec shared memory et coalescing

En effet, nous avons gagné **0,286** seconde en utilisant les mémoires partagées et le coalescing. Ce qui représente une diminution de 52% du temps d'exécution sur CUDA pour l'algorithme des kmeans. Pour l'indice de Davies-Boulding, on a une diminution de 0,22 ce qui améliore légèrement notre algorithme.

VI. L'impact de la taille des blocs et de la grille

La taille des blocs et de la grille impactent surtout le temps d'exécution de l'algorithme. Une diminution de la taille des

blocs nous a permis d'observer une amélioration relativement faible du temps d'exécution. Cependant, il convient de noter que j'ai choisi après plusieurs essais des tailles de bloc différentes pour les kernels deviceFill, deviceCopy et le kernel principal pour laquelle l'amélioration a été remarquée. Pour les deux kernels précédent, le temps d'exécution s'empire lorsqu'on diminue la taille des blocs.

Conclusion

Pour ce projet, après avoir passé plusieurs heures sur l'implémentation du code qui générait un bug sur la taille des blocs (limité à 1024), j'ai réussi à surmonter l'erreur et à comprendre aussi l'impact de ces paramètres sur l'implémentation de mon algorithme. D'autre part, ce projet m'a permis de revoir certaines notions des vidéos vues en classe sur lesquelles je suis passé très rapidement.

Enfin, bien que les résultats obtenus par rapport à la qualité du clustering me semblent encore perfectible, j'ai réussi à faire la comparaison de l'implémentation avec la baseline qui est l'implémentation sur CPU, j'ai également pu implémenter les notions de shared memory, coalescing qui ont permis de gagner quelques secondes en temps d'exécution.