الجـــمهوريـــة الجزائريـــة الديمـــقراطيـــــة الشـعبيـة
ⵜⴰⵙⴷⴰⵡⵜ ⵜⴰⵣⵣⴰⵢⵔⵉⵜ ⵜⴰⵎⴰⵙⵙⴰⵖⵜ ⵜⴰⵢⴰⵖⵍⴰⵜ
République Algérienne Démocratique et Populaire
وزارة التعليـــم العـــالي والبـــحث العلمـــي
ⴰⵢⵣⵔⴼ ⵏ ⵜⵓⵙⵎⵔⵙⵉⵏ ⵜⵉⵏⵣⵔⴼⵉⵏ ⴷ ⵜⵓⵙⵙⵏⵉⵡⵉⵏ
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
ⴰⵢⵜⵓⵔⴰⵔ ⴰⵢⵜⵍⵓ ⵜⵉⵣⵣⴰⵢ ⵏ ⵜⵓⵙⵙⵏⵉⵡⵉⵏ
École nationale Supérieure d'Informatique

Rapport du TP

Programmation des GPUs NVIDIA avec CUDA

2 ème année Cycle Supérieur (2CS)

Option : Système Informatique (SQ)

Thème :

Optimisation du calcul de la "Longest Common Subsequence (LCS)" par la parallélisation

Réalisé par :

- Benameur Tarek
- Touil Nihel

b. Stratégie de parallélisation sur le GPU

La parallélisation de l'algorithme de calcul de la Longest Common Subsequence (LCS) repose sur une organisation du calcul des cellules de la matrice en **anti-diagonales**, exploitant ainsi les propriétés d'indépendance au sein de chaque anti-diagonale et respectant les dépendances entre les différentes anti-diagonales. Cette approche est particulièrement adaptée à une implémentation sur GPU avec CUDA, où les threads et blocs sont utilisés pour répartir le calcul.

1. Organisation par Anti-Diagonales :

La matrice LCS est parcourue à l'aide d'**anti-diagonales**, qui sont définies comme des ensembles de cellules dont la somme des indices est constante, c'est-à-dire que pour chaque anti-diagonale, i+j=k, où k est une constante spécifique à chaque anti-diagonale. Chaque cellule d'une anti-diagonale peut être calculée indépendamment, à condition que les cellules des anti-diagonales précédentes aient déjà été traitées. Cette organisation permet de décomposer le calcul en tâches indépendantes, favorisant ainsi la parallélisation.

2. Exécution Parallèle au sein des Anti-Diagonales :

Dans le cadre de l'utilisation de CUDA, chaque **thread** est responsable du calcul d'une cellule dans l'anti-diagonale assignée. Par exemple, pour une anti-diagonale donnée (avec i+j=k), chaque thread traite un indice (i,j) spécifique, permettant un calcul parallèle efficace au sein de l'anti-diagonale. L'indépendance des calculs dans chaque anti-diagonale offre un haut potentiel de parallélisme, où plusieurs threads peuvent s'exécuter simultanément sans interférer les uns avec les autres.

3. Synchronisation entre Anti-Diagonales :

Un défi majeur dans cette approche réside dans les dépendances entre les anti-diagonales. En effet, le calcul des cellules d'une anti-diagonale d dépend des résultats des cellules de l'anti-diagonale d−1. Par conséquent, il est crucial de s'assurer que tous les threads ayant calculé les cellules de l'anti-diagonale précédente aient terminé leur travail avant que les threads de l'anti-diagonale suivante ne commencent leur calcul. Pour cela, une **barrière de synchronisation** est nécessaire à la fin de chaque anti-diagonale, ce qui permet de garantir que tous les threads de l'anti-diagonale d−1 ont terminé avant de débuter les calculs pour l'anti-diagonale d. Cette synchronisation est assurée par la fonction cudaDeviceSynchronize() de CUDA.

4. Utilisation des Blocs et des Threads :

Les threads sont organisés en **blocs** dans CUDA. Chaque bloc est responsable du calcul sur un sous-ensemble d'anti-diagonales de la matrice. Le nombre de threads par bloc doit être choisi de manière à optimiser l'utilisation des ressources matérielles du GPU. Par exemple, un bloc de 256 threads peut être utilisé pour traiter plusieurs cellules dans une ou plusieurs anti-diagonales. L'organisation en blocs permet de maximiser le parallélisme tout en respectant la contrainte de synchronisation entre les anti-diagonales successives.

c. Schéma de la parallélisation sur le GPU

Ce schéma illustre le calcul parallèle de la LCS sur un GPU avec CUDA :

1. Préparation des données (CPU → GPU) : Le CPU alloue la mémoire, lance le kernel et transfère les chaînes et la matrice vers la DRAM du GPU.

2. Exécution du kernel (GPU) : Le GPU organise les threads en grille et blocs, chaque thread calculant une cellule de la matrice sur une anti-diagonale.

3. Calcul parallèle : Les threads exécutent les calculs, stockent les indices dans leurs registres, synchronisent et mettent à jour la matrice.

4. Écriture des résultats (GPU → CPU) : Les résultats sont copiés vers la mémoire CPU pour affichage et traitement final.

5. Libération des ressources : Le CPU libère la mémoire GPU.

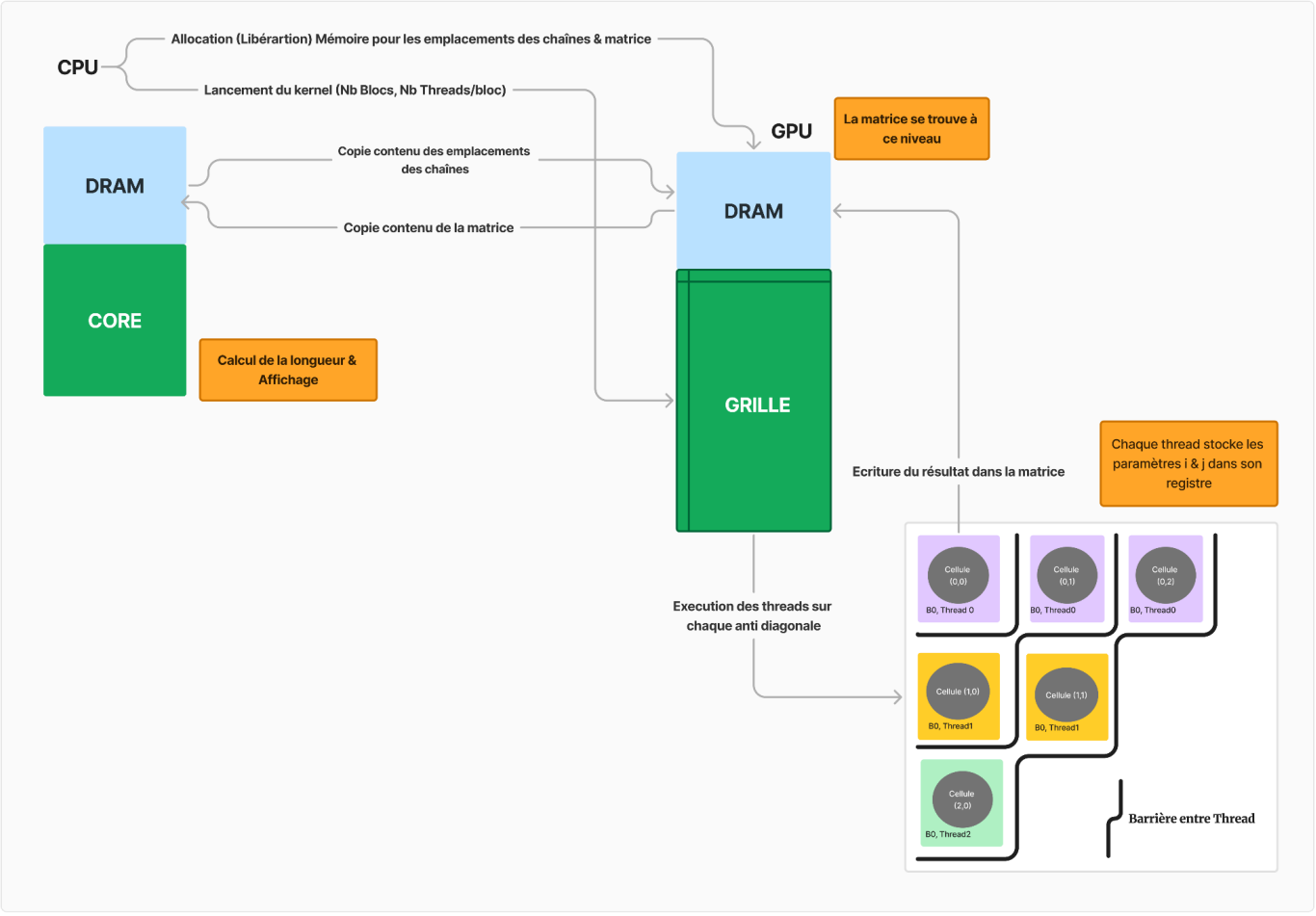

Figure: Schéma de parallélisation pour le calcul de la matrice LCS sur GPU

e. Mesure des Performances de la Solution Parallèle sur le GPU

La figure présente les temps d'exécution (en secondes) pour le calcul de la Longest Common Subsequence (LCS) en mode séquentiel (S), en mode parallèle avec Pthreads (P2, P4, P8, P16) et avec CUDA (pour le GPU), selon différentes tailles de données.

	5	25	150	500	1000	2500	5000	10000	20000	25000	30000	40000
S	0.000001	0.000004	0.000090	0.001314	0.005381	0.033639	0.135007	0.526286	2.126123	3.479031	4.741321	8.606111
P2	0.000469	0.000336	0.001629	0.007003	0.020946	0.114609	0.399255	1.874070	7.559803	10.855699	16.075705	33.076165
P4	0.000627	0.000728	0.003567	0.015255	0.037706	0.173266	0.626624	2.518414	8.875448	13.941209	19.948204	36.553770
P8	0.001590	0.001531	0.007319	0.029155	0.069225	0.260029	0.834362	3.212283	12.097762	18.798692	27.288627	46.962532
P16	0.002806	0.003468	0.015883	0.057949	0.127111	0.399566	1.226591	4.275336	17.142909	26.708189	39.044995	68.912485
CUDA	0.000075	0.000299	0.001776	0.005992	0.012340	0.032323	0.065970	0.133794	0.314917	0.421054	0.620773	1.386594

Figure: Temps d'exécution séquentiel, parallèle (sur Pthreads & CUDA) pour différentes tailles de données dans le calcul du LCS (Longest Common Subsequence)

Analyse des comparaisons de performances

Sur le graphique, on observe que **les performances de CUDA surpassent celles des autres méthodes parallèles (Pthreads)** à partir d'une certaine taille des données . Cela indique que l'utilisation du GPU devient plus avantageuse pour les calculs intensifs lorsque la taille des données augmente, grâce à une meilleure exploitation de la parallélisation massive des cœurs du GPU.

En revanche, pour les **implémentations avec Pthreads**, les performances ne montrent pas d'amélioration significative même en augmentant le nombre de threads (P2, P4, P8, P16). Cela pourrait être dû à plusieurs facteurs :

1. **Synchronisation excessive** entre les threads : la parallélisation introduit un coût supplémentaire lié à la gestion des communications et des verrous.
2. **Granularité du problème** : si le travail à diviser est trop petit, les gains de parallélisme sont limités par le **surcoût d'organisation des threads**.

Par ailleurs, pour **de petites tailles de données**, l'implémentation **séquentielle** reste plus performante que les versions parallèles (Pthreads et CUDA). Ceci s'explique par le **surcoût lié à l'initialisation et à la gestion des threads ou des cœurs GPU et le transfert entre les mémoires du hôte (CPU) et le device (GPU)** pour des entrées de taille réduite. Le temps nécessaire pour préparer et distribuer les tâches dépasse alors les gains obtenus par le parallélisme.

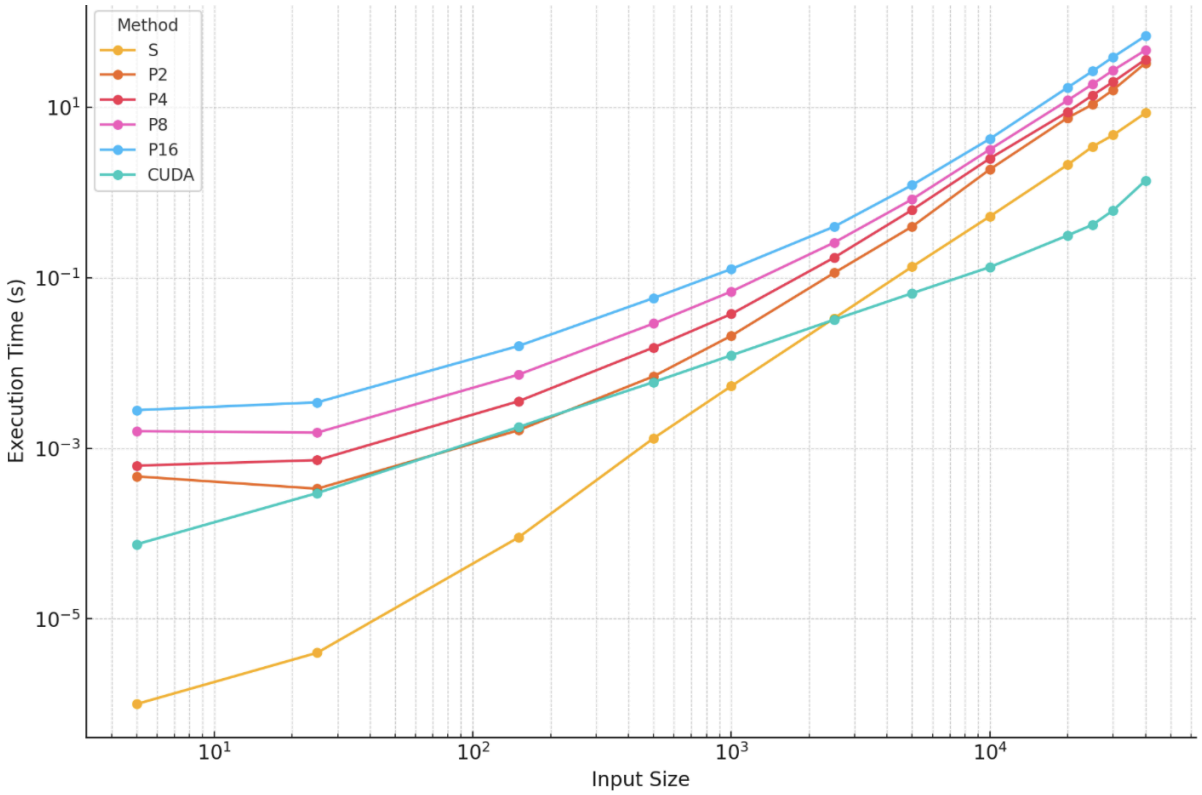

Figure: Temps d'exécution du LCS en séquentiel et parallèle (Pthreads & GPU)