REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE
ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

Groupe : 2CS - SIQ1 **Année : 2024/2025**

TP3 HPC : Programmation des GPUs NVIDIA avec CUDA

Simulation de la propagation de chaleur dans des structures matérielles

Réalisé par :

- FLISSI Loubna.
- LARBAOUI Yasmine Badr El Houda.

Encadré par :

Madame HAICHOUR Amina Selma.

Table des matières

Introduction

À mesure que les simulations numériques deviennent de plus en plus complexes, il est crucial d'optimiser les méthodes de calcul afin de réduire les temps de traitement et de mieux gérer des données toujours plus volumineuses. La parallélisation des algorithmes, en tirant parti des capacités des processeurs multi-cœurs et des GPU, apparaît comme une solution incontournable pour répondre à ces défis.

Dans ce cadre, ce TP se concentre sur la parallélisation d'un programme de **simulation de la propagation de la chaleur**.
Dans un premier temps, nous avons utilisé la bibliothèque Pthreads pour paralléliser le programme séquentiel du TP1. Maintenant, dans le TP3, nous allons reprendre ce même programme et le paralléliser en utilisant **CUDA**, une technologie qui permet d'exploiter la puissance des cartes graphiques.

L'objectif de ce travail est de comparer les performances de ces deux méthodes de parallélisation (Pthreads et CUDA) avec le programme séquentiel. Nous analyserons dans quelle mesure ces approches permettent de réduire le temps de calcul et d'augmenter l'efficacité des simulations.

a. Rappel du problème à traiter :

Rappelons-nous du programme du premier TP, qui a pour objectif de simuler la propagation de la chaleur sur une grille bidimensionnelle. Chaque cellule de cette grille représente une section d'une structure matérielle, avec une température associée à un instant donné.

Le programme suit ces étapes principales :

- **Initialisation de la grille :** Les températures initiales des cellules sont générées aléatoirement pour simuler des conditions variées.

- **Calcul des nouvelles températures :** À chaque itération, la température de chaque cellule est mise à jour en fonction de la moyenne des températures de ses voisins (haut, bas, gauche, droite), avec l'ajout d'une composante sinusoïdale pour modéliser des variations périodiques.

- **Mise à jour de la grille principale :** Les nouvelles valeurs sont appliquées à la grille pour illustrer l'évolution thermique.

Les principales zones critiques du programme (hotspots) se trouvent dans :

- **Le calcul des nouvelles températures :** Cette étape nécessite des calculs intensifs pour chaque cellule, avec une complexité quadratique $O(n^2)$, ce qui peut ralentir les performances pour les grandes grilles.

- **La mise à jour de la grille principale :** Chaque itération demande un grand nombre d'opérations mémoire pour actualiser la grille.

Ces étapes sont particulièrement adaptées à une parallélisation, car les calculs pour chaque cellule sont indépendants. Pour cela, nous allons utiliser CUDA et comparer les performances obtenues avec celles de la version séquentielle et de la version parallélisée avec Pthreads.

b. La stratégie de parallélisation :

La stratégie de parallélisation dans le code CUDA vise à exploiter le parallélisme massif des GPU pour accélérer les calculs liés à la propagation de la chaleur dans une grille de grandes dimensions.

1. **Division de la grille en threads :**
   La grille est divisée en blocs, chaque bloc contenant un ensemble de threads. Chaque thread est responsable du calcul d'une cellule spécifique, en fonction de ses indices (i, j). Cela permet de paralléliser les calculs sur l'ensemble de la grille.

2. **Calcul des nouvelles valeurs :**
   Le kernel **calculer_nouvelle_grille** est lancé en parallèle par tous les threads. Chaque thread calcule la nouvelle valeur de sa cellule en utilisant celles de ses voisines et une fonction mathématique. Ce calcul concerne uniquement les cellules internes, car les bords restent fixes.

3. **Mise à jour de la grille principale :**
   Le kernel **mettre_a_jour_grill**e copie les nouvelles valeurs calculées dans la grille principale. Cela prépare la grille pour l'itération suivante tout en assurant que les données restent cohérentes.

4. **Synchronisation :**
   À chaque étape, une synchronisation **(cudaDeviceSynchronize)** est utilisée pour s'assurer que tous les threads ont terminé leurs calculs avant de passer à l'étape suivante. Cela garantit que les données sont prêtes et stables pour les prochaines opérations.

5. **Boucle d'itérations :**
   Ces étapes sont répétées pour un nombre fini d'itérations. Chaque itération correspond à une simulation de l'évolution de la propagation de chaleur dans le temps.

6. **Transfert des données :**
   Après les calculs, les résultats finaux sont copiés du GPU vers la mémoire de l'hôte (CPU) pour être analysés ou affichés.

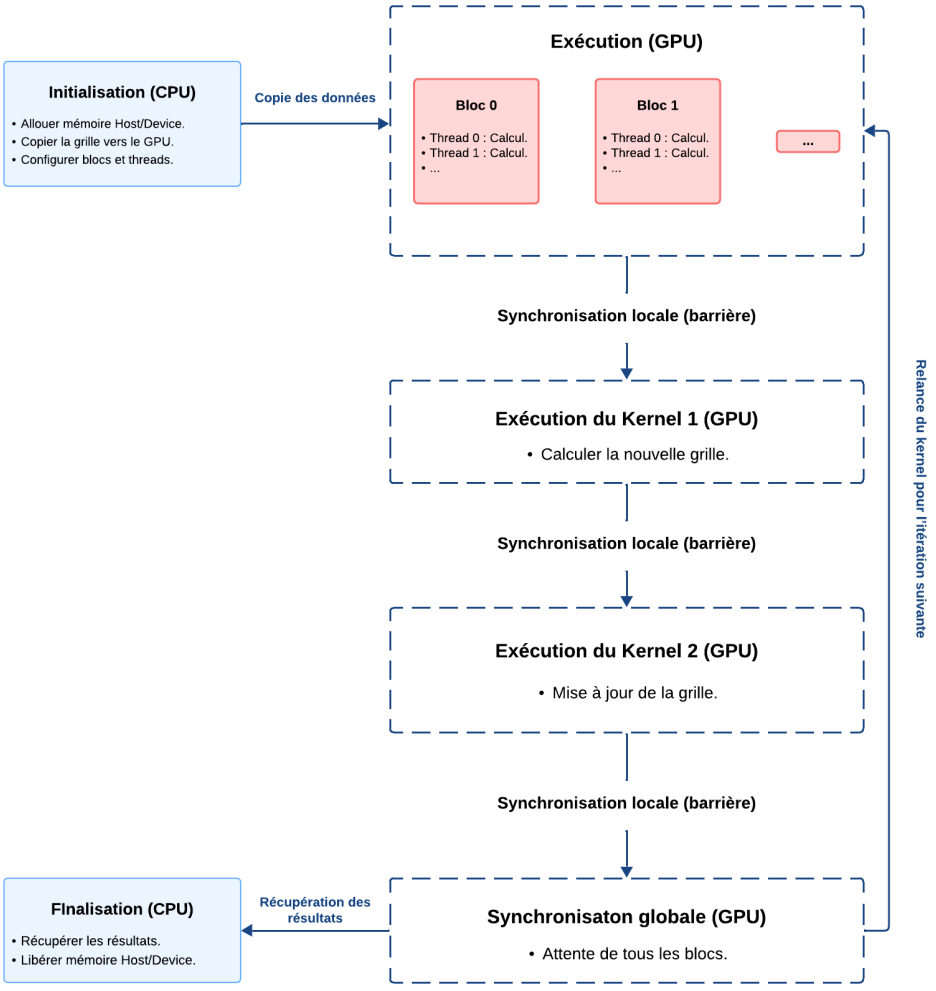
Cette stratégie exploite au maximum la puissance de calcul parallèle du GPU tout en assurant la cohérence et l'efficacité des données entre les itérations.

c. Représentation schématique de la solution parallèle :

Le schéma suivant présente les étapes du calcul parallèle appliqué à la simulation de la propagation de chaleur sur une grille. Il illustre la répartition des calculs entre le CPU et le GPU, en mettant en évidence les différentes phases de l'exécution, notamment l'initialisation, l'exécution des kernels sur le GPU, et la récupération des résultats.

Dans cette approche, le CPU intervient principalement lors des phases d'initialisation (allocation mémoire, copie des données vers le GPU) et de finalisation (récupération des résultats, libération de la mémoire). Le GPU prend en charge les calculs parallèles grâce à une organisation optimisée en blocs et threads, avec deux kernels distincts : l'un pour le calcul des nouvelles valeurs et l'autre pour la mise à jour de la grille. Une synchronisation globale est effectuée entre ces deux kernels pour garantir la cohérence des données avant de relancer l'itération suivante.

Ce schéma offre ainsi une représentation claire et structurée du flux d'exécution, depuis l'allocation de la mémoire jusqu'à la récupération des résultats finaux, tout en illustrant la répétition des kernels nécessaires pour chaque itération.



e. Comparaison des Performances :

1- Les temps d'exécution du programme parallèle version CUDA :

Nombre de threads	Taille de la grille	Temps d'exécution
Code séquentiel	100 x 100	execution time : 2.043 s
	1000 x 1000	execution time : 102.809 s
2 threads	100 x 100	Temps d'execution: 0.02 secondes
	1000 x 1000	Temps d'execution: 7.10 secondes
4 threads	100 x 100	Temps d'execution: 0.01 secondes
	1000 x 1000	Temps d'execution: 3.70 secondes
6 threads	100 x 100	Temps d'execution: 0.01 secondes
	1000 x 1000	Temps d'execution: 2.54 secondes
8 threads	100 x 100	Temps d'execution: 0.01 secondes
	1000 x 1000	Temps d'execution: 1.92 secondes
16 threads	100 x 100	Temps d'execution: 0.01 secondes
	1000 x 1000	Temps d'execution: 1.11 secondes
24 threads	100 x 100	Temps d'execution: 0.00 secondes
	1000 x 1000	Temps d'execution: 0.77 secondes

50 threads	100 x 100	Temps d'execution: 0.00 secondes
	1000 x 1000	Temps d'execution: 0.77 secondes

2 - Analyse des résultats :

- **Temps d'exécution séquentiel :**
  - Dans le programme séquentiel, le calcul s'exécute sur le processeur (CPU), où chaque élément de la grille est mis à jour itérativement. Cela entraîne une complexité O((TAILLE)²×ITERATIONS) qui est coûteuse pour des grilles de grande taille.
  - Le temps d'exécution observé (par exemple 102.809 secondes pour une grille (1000×1000) reflète la nature intensive des calculs dans le modèle séquentiel.

- **Temps d'exécution parallèle avec CUDA :**
  - Dans l'implémentation CUDA, les calculs sont répartis sur des milliers de cœurs du GPU, chaque thread traitant une partie de la grille. Cela réduit considérablement le temps d'exécution.
  - L'augmentation du nombre de threads par bloc permet d'améliorer les performances, mais seulement jusqu'à un certain niveau, au-delà duquel elles cessent de progresser et se stabilisent.

    les raisons de ces limites:

    Sous-utilisation des ressources GPU : Avec un faible nombre de threads par bloc (par exemple, 2 ou 4), une partie des cœurs du GPU reste inactive, ce qui entraîne un temps d'exécution plus élevé.

    Saturation du GPU : Lorsque le nombre de threads par bloc augmente (par exemple, 16 ou 24), les ressources du GPU sont mieux exploitées, atteignant un point optimal où la parallélisation est maximale.

    Overhead lié à la gestion des threads : Un trop grand nombre de threads par bloc peut entraîner des conflits d'accès mémoire ou une latence accrue due à la synchronisation.

- **Effet de la parallélisation :**
  - Le programme parallèle montre une accélération notable par rapport au séquentiel. Par exemple :
    - ➔ Avec 16 threads par bloc, le temps d'exécution est significativement réduit (par exemple 1.11 secondes pour une grille 1000×1000
    - ➔ Cela représente un facteur d'accélération d'environ 102.809/1.11≈92.6, soit une réduction de temps de près de 93 %.
  - L'implémentation CUDA bénéficie également de l'architecture mémoire hiérarchique du GPU :
    - ➔ Les calculs intensifs sont effectués dans les threads, limitant les transferts mémoire coûteux entre le CPU et le GPU.
    - ➔ Les données sont stockées dans des mémoires rapides (mémoire partagée) pour réduire les latences.

3- Comparaison avec Pthreads [TP1] :

- **Observation des temps d'exécution : Grille (1000 x 1000)**

Nombre de threads	Temps avec Pthreads	Temps avec CUDA
2 threads	**54.234 s**	**7.10 s**
4 threads	**31.838 s**	**3.78 s**
6 threads	**25.973 s**	**2.54 s**
8 threads	**23.010 s**	**1.92 s**
16 threads	**21.973 s**	**1.11 s**
24 threads	**20.922 s**	**0.77 s**

50 threads	**21.524 s**	**0.77 s**

- **Interprétation des résultats :**

a. Pthreads :

Le temps d'exécution diminue avec l'augmentation du nombre de threads, mais se stabilise à partir de 16 threads, avec des valeurs proches de 21 s (20,9 s pour 16 et 24 threads).

Ceci est dû à :

➔ **La surcharge de gestion des threads**.
➔ **La saturation des cœurs CPU disponibles** : Le CPU a un nombre limité de cœurs physiques et logiques, au-delà desquels ajouter des threads n'apporte pas d'amélioration.

Conclusion : Pthreads fonctionne efficacement jusqu'à **un certain seuil**, mais atteint des limites dues à l'architecture CPU.

b. CUDA :

Avec CUDA, les temps d'exécution sont significativement plus bas :

➔ **2 threads** : 7.10 s (déjà meilleur que pthreads avec 2 threads).
➔ **24 threads** : 0.77 s (plus de 25 fois plus rapide que pthreads à 24 threads).

Cela s'explique par :

➔ **L'architecture GPU massivement parallèle** : Contrairement au CPU, le GPU est optimisé pour exécuter des milliers de threads simultanément.
➔ **La réduction de la surcharge de synchronisation** : CUDA organise efficacement les calculs parallèles et utilise des blocs et des grilles de threads.
➔ **La mémoire GPU rapide** : Le transfert de données vers la mémoire partagée GPU est plus performant dans des contextes intensifs.

Conclusion : CUDA surpasse pthreads dans cette comparaison grâce à sa capacité à exploiter un parallélisme massif.

Conclusion :

- **CUDA est plus performant que pthreads** pour des tâches hautement parallélisables comme celle-ci, en raison de l'architecture spécialisée des GPU.
- **Pthreads est limité par l'architecture CPU** et ne peut pas rivaliser avec le parallélisme massif du GPU.
- Pour des tâches nécessitant un grand nombre d'itérations et de calculs, **CUDA est le meilleur choix**, malgré la complexité de programmation supplémentaire.