

Projet de HPCA : Transport de neutrons

Parallélisation d'une application physique simple

Mina Pêcheux

MAIN5 - Polytech Sorbonne (Automne 2018)

Encadrée par : Pierre Fortin et Lokmane Abbas Turki

Table des matières

Contexte et objectifs

Parallélisation GPU

Parallélisation sur CPU

Parallélisation hybride

Conclusion

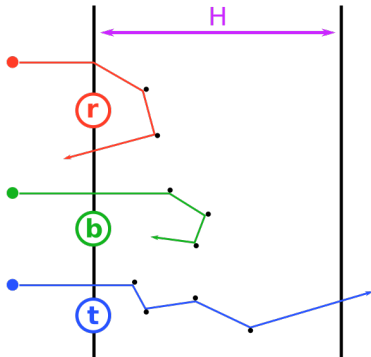
Contexte et objectifs

Présentation du problème

Contexte physique

Etude du **transport de neutrons** à travers une plaque fine.

3 possibilités : **réflexion** (r), **absorption** (b), **transmission** (t)



Paramètres physiques :

- épaisseur de la plaque H
- section efficace de capture C_c
- section efficace de diffusion C_s
- section efficace totale

$$C = C_c + C_s$$

Objectifs du projet

Appréhender et comparer différents outils de parallélisation usuels :

- sur GPU : **CUDA**
- sur CPU : **OpenMP** et **MPI**

Cas de référence :

- $H = 1.0$
- $n = 500000000$
- $C_c = 0.5$
- $C_s = 0.5$

Critères d'efficacité :

- temps total d'exécution
- accélération

Parallélisation GPU

Buts dans cette partie

Parallélisation du code séquentiel avec **CUDA** + appréhension des techniques de base de **programmation sur GPU**

3 problématiques :

- taille/forme de la grille sur GPU ?
- génération de nombres aléatoires ?
- stockage des résultats ?

Implémentation : dimension de la grille

- ① Taille de la grille (nombre de blocs, de threads) ?
- ② Forme de la grille (disposition des blocs et threads en x, y, z) ?
 - pas d'influence sur les calculs
 - préférence du programmeur

→ Travail sur une **grille linéaire** de taille **nbBlocks** × **nbThreads** :





Implémentation : nombres pseudo-aléatoires

Génération de nombres pseudo-aléatoires : 1 générateur par thread

①

Initialisation du générateur

Thread 0 : 

Thread 1 : 

Thread 2 : 

→ graine unique donnée par
threadIdx.x

②

Génération de valeurs

 →  →  → ...

 →  →  → ...

 →  →  → ...

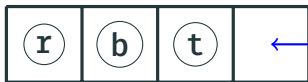
→ tirage d'une valeur aléatoire
+ mise à jour du générateur

Implémentation : gestion de la mémoire

Stockage des résultats :

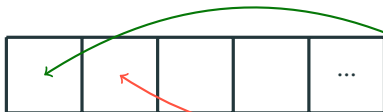
- 1 tableau **res** de 4 cases avec les comptes $\textcircled{\mathbf{r}}$, $\textcircled{\mathbf{b}}$ et $\textcircled{\mathbf{t}}$:

→ algo. de réduction



← indice du dernier neutron absorbé

- 1 tableau **absorbed** pour les positions des neutrons absorbés (stockage **contigu**) :



position du neutron absorbé n°0

position du neutron absorbé n°1

→ `atomicAdd()`

2 phases de transferts mémoire :

- CPU → GPU : initialisation des tableaux sur GPU
- CPU ← GPU : récupération des résultats sur CPU

Implémentation : *batching*

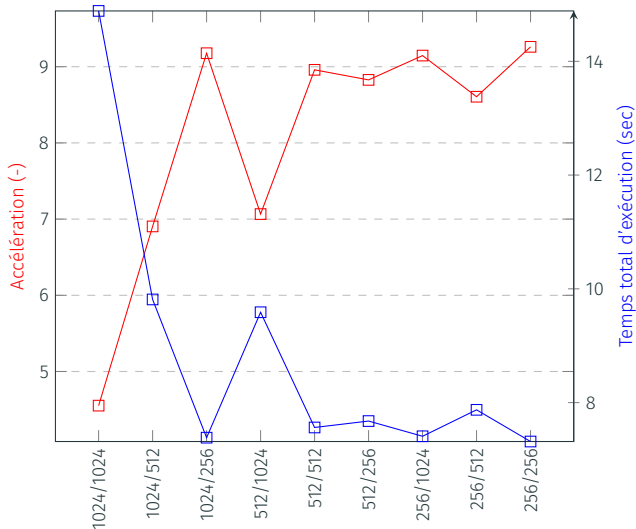
Limites de calcul / mémoire

Nécessité de ***batcher*** (regrouper) le traitement sur les threads GPU !
→ 1 thread traite plusieurs neutrons

2 idées :

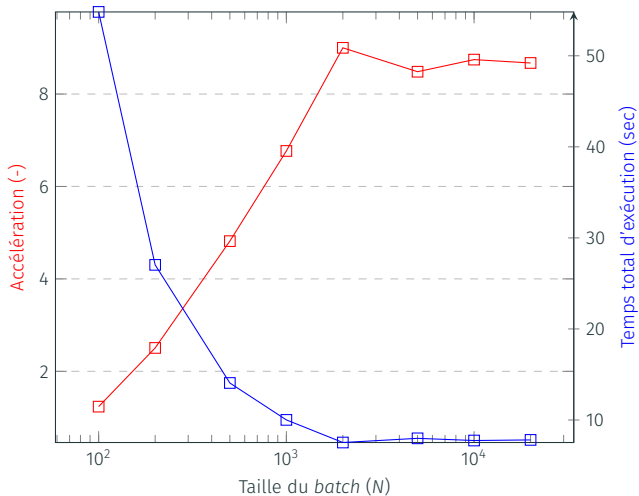
- *batching* automatique : grille fixe, *batches* adaptatifs
- *batching* “manuel” : *batches* fixes, grille adaptative

Résultats (*batching* automatique)



→ accélération optimale : **9.261**

Résultats (*batching* "manuel")



→ accélération optimale : **9.000**

Parallélisation sur CPU

Objectifs

Comparer **OpenMP** à **CUDA**

NB_OMP_THREADS threads se partagent les calculs avec un **comportement homogène** entre chaque thread :

- initialisation du générateur pseudo-aléatoire (avec le numéro de thread **OpenMP**)
- traitement de neutrons (accès concurrent à **absorbed**)
`#pragma omp atomic`
- réduction des compteurs avec les comptes globaux (**r, b, t**)
`#pragma omp parallel ... reduction(+:r,b,t)`

Par rapport à CUDA ?

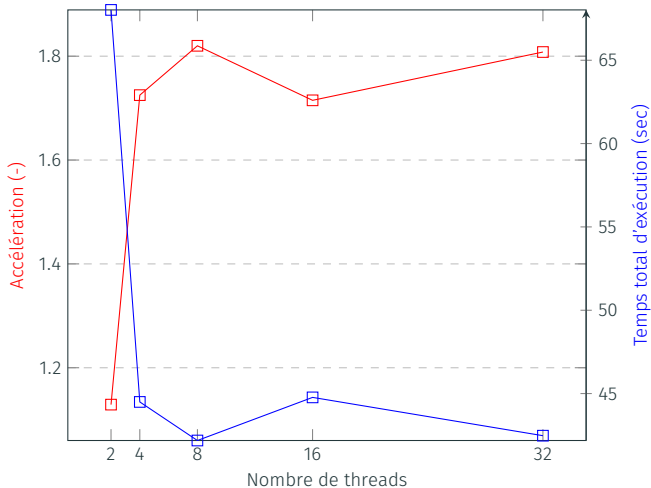
Avantages

- très facile à implémenter
- ne nécessite pas de GPU

Inconvénients

- moins d'accélération !

Résultats (OpenMP)



→ accélération optimale : **1.820**

Parallélisation hybride

Hybride, quésaco ?

Idée de base

Mélanger les outils de parallélisation GPU et CPU pour **combiner** leur puissance

Comment gérer la **concurrence** des traitements sur GPU et CPU ? des accès mémoire ?

①

CUDA + OpenMP

Parallélisation
inter-threads

②

CUDA + MPI

Parallélisation
inter-processus

Problème de sérialisation

Impossible de paralléliser l'exécution GPU et le traitement sur CPU par les threads **OpenMP** !

- idée abandonnée
- utilisation de **MPI**

Idée n°2 : CUDA + MPI

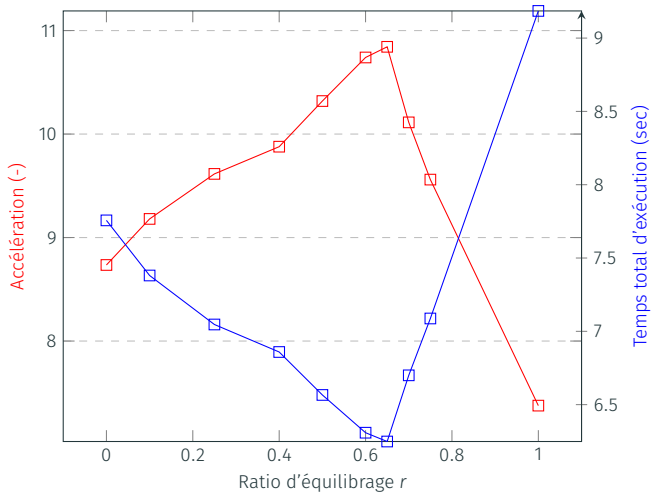
Equilibrage de la charge de travail : spécification par l'utilisateur de la proportion de neutrons traités sur GPU/CPU

(GPU seul =) 0 ← Ratio d'équilibrage \mathfrak{x} → 1 (= CPU seul)

Processus master p_0 : répartition des neutrons à traiter, gestion du GPU, récupération et rassemblement des résultats

Autres processus p_i ($i \neq 0$) : récupération du nombre n_i de neutrons à traiter, traitement de n_i neutrons, renvoi des résultats "locaux" à p_0

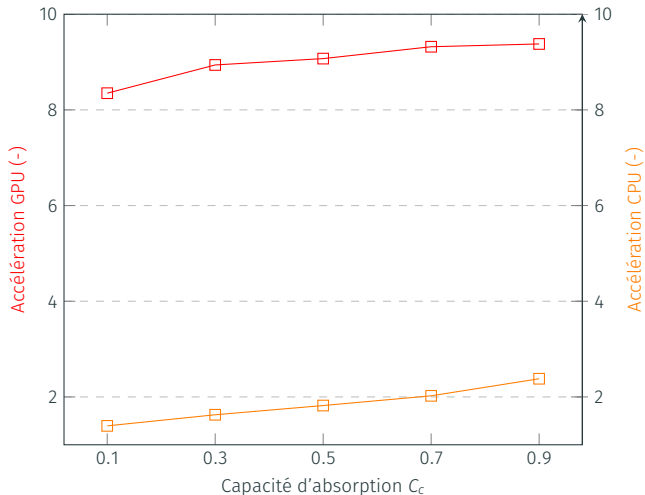
Résultats (CUDA + MPI)



→ accélération optimale (pour 8+1 processus) : **10.843**

Conclusion

Influence de la capacité d'absorption



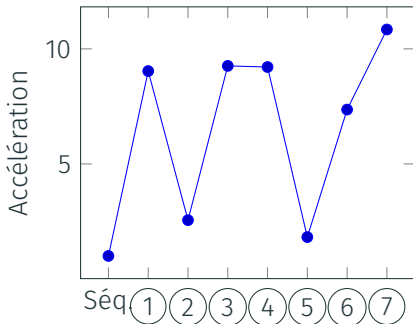
→ petite accélération : neutrons absorbés (+) vite ($\times 1.123$, $\times 1.706$)

4 idées d'amélioration :

- Contournement de l'**atomicAdd()**
- Parallélisation avec des *streams* GPU
- Parallélisation multi-GPUs
 - complique la gestion de la mémoire...
 - problèmes de *drivers* ?
- Optimisation des communications MPI (routines collectives)

Conclusion du projet

- Comparaison de 3 outils de parallélisation : **CUDA**, **OpenMP**, **MPI**
- Appréhension des exécutions concurrentes
- Etude de l'accélération pour différents modes de parallélisation



①	GPU (<i>batch</i> . "auto" simple)
②	GPU (<i>batch</i> . "manuel" simple)
③	GPU (<i>batch</i> . "auto" optimisé)
④	GPU (<i>batch</i> . "manuel" optimisé)
⑤	CPU (parallélisation OpenMP)
⑥	CPU (parallélisation MPI)
⑦	Hybride (CUDA+MPI)

Merci pour votre attention !