Projet de HPCA : Transport de neutrons

[MAIN 5] Mina Pêcheux *(monôme)* - Automne et Hiver 2018-2019

Remarque préliminaire : A cause de divers incidents techniques sur le réseau Polytech durant les vacances de Noël, je n'ai pas pu mettre en place de nouveaux algorithmes. Ce dossier présente donc les algorithmes de la version 1 et quelques pistes de réflexions (partiellement ou non implémentées) qui pourraient améliorer l'efficacité de nos algorithmes parallèles.

1 Présentation

Ce projet s'intéresse à la simulation parallèle de transport de neutrons à l'aide d'une méthode de type Monte Carlo. On considère un modèle simplifié en 2D où une source émet des neutrons contre une plaque homogène d'épaisseur H et de hauteur infinie.

Comme montré *Figure* 1, un neutron peut être réfléchi par la plaque (noté **`r`**), absorbé dans celle-ci (noté **`b`**) ou transmis à travers celle-ci (noté **`t`**). On souhaite calculer les pourcentages d'occurrence des trois cas **`r`**, **`b`** et **`t`**, et on sauvegarde également les positions (finales) des neutrons absorbés dans la plaque.

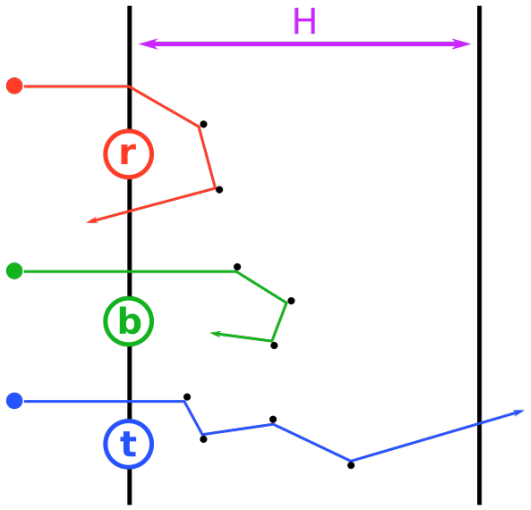


Figure 1 – *Schéma du modèle simplifié : un neutron peut être réfléchi (**`r`**), absorbé (**`b`**) ou transmis (**`r`**)*

On calcule la distance parcourue par les neutrons à l'aide de nombres aléatoires uniformément distribués dans l'intervalle $[0, 1[$. La simulation d'un neutron continue tant que le neutron n'a pas été réfléchi, transmis ou absorbé par un atome. Le problème physique dépend aussi de deux paramètres physiques : c_c (la "section efficace" de capture) et c_s (la "section efficace" de diffusion) tels que : $C = c_c + c_s$ est la section efficace totale.

2 Parallélisation sur GPU

Dans un premier temps, on souhaite réaliser la simulation en **CUDA** sur un GPU.

Pour avoir un repère, on détermine un cas de référence : $H = 1.0$, $n = 500000000$, $c_c = 0.5$ et $c_s = 0.5$. Si on lance la simulation avec le code séquentiel fourni (sur CPU), on obtient les temps d'exécution suivants :

```
Epaisseur de la plaque :     1
Nombre d'echantillons  : 500000000
C_c : 0.5
C_s : 0.5
Pourcentage des neutrons reflechis : 0.11
Pourcentage des neutrons absorbes : 0.43
Pourcentage des neutrons transmis : 0.46

Temps total de calcul: 67.754874 sec
Millions de neutrons /s: 7.4
Result written in /tmp/absorbed.dat
```

L'ensemble des résultats de cette partie ont été obtenus sur le **pc5016**.

2.1 Première implémentation : *batching* automatique

Dimension de la grille de calcul
Pour commencer, déterminons la taille de notre grille de GPUs. On sait que la forme de nos blocs et de nos threads n'a pas d'influence sur le calcul et que ce choix dépend donc principalement de nos préférences en tant que programmeur. Le plus simple, ici, est de construire une grille "linéaire", c'est-à-dire de profiter au maximum de la dimension en x.

Afin de ne pas dépasser la mémoire disponible sur GPU et d'optimiser les calculs, on va "batcher" le traitement : chaque thread va traiter un grand nombre de neutrons. Autrement dit, notre kernel contiendra une boucle **while** qui forcera le thread GPU à traiter de nouveaux neutrons tant que le nombre demandé n'aura pas été atteint. On va donc choisir[1] :

$$\texttt{nbThreads} = (512, 1, 1)$$
$$\texttt{nbBlocks} = (256, 1, 1)$$

La *Figure* 2 représente schématiquement la grille linéaire utilisée pour le calcul sur GPU.

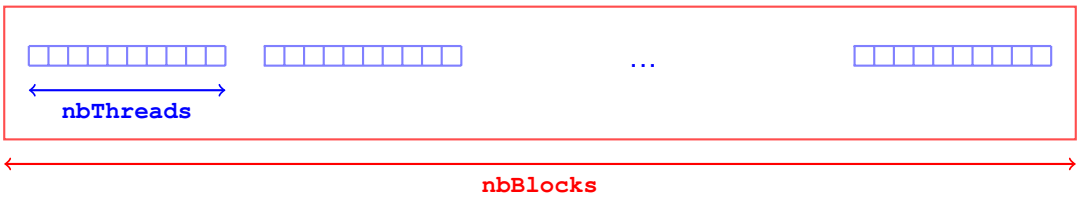


Figure 2 – *Représentation schématique de la forme de la grille utilisée pour le calcul parallèle sur un GPU*

Remarque importante : comme on l'observera tout au long du projet, sur GPU, la majorité du temps d'exécution est due à l'allocation mémoire initiale plutôt qu'aux calculs. Pour notre cas de référence ($H = 1.0$, $n = 500000000$, $c_c = 0.5$, $c_s = 0.5$), le traitement GPU nécessite un temps minimum incompressible pour l'allocation mémoire (d'environ 5 secondes).

Plus le nombre de blocs et threads sur GPU est important, plus ce temps d'allocation est long. Il faut donc faire un compromis entre la taille de notre grille et son efficacité.

[1] D'après la documentation **Nvidia** (https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications), on sait qu'on peut avoir jusqu'à $2^{31} - 1$ blocs en x et jusqu'à 2048 threads en x dans un bloc.

Génération de nombres pseudo-aléatoires
Ensuite, nous devons générer des nombres aléatoires. Pour cela, on utilise la bibliothèque **CUDA** "**cuRAND**". Pour que chaque thread GPU ait sa propre séquence de nombres aléatoires, on va utiliser l'identifiant du thread (unique) comme graine d'initialisation du générateur, où cet identifiant est défini comme :

$$i = \texttt{blockIdx.x} * \texttt{blockDim.x} + \texttt{threadIdx.x}$$

La génération se passe en deux temps : d'abord, il faut initialiser le générateur de nombres aléatoires de chaque thread, puis ensuite on peut y accéder pour obtenir un nouveau nombre aléatoire aisément, dès que nécessaire (voir la *Figure* 3).

La première fonction (**setup_kernel_seeds**) permet d'initialiser le générateur de nombres aléatoires de chaque thread à partir de son identifiant unique. La deuxième fonction (**uniform_random_number**) permet d'obtenir un nombre aléatoire à partir du générateur pour un thread donné (et de faire progresser le générateur pour la prochaine fois). Il nous suffit donc maintenant :

- dans notre fonction de lancement de kernels, d'appeler la routine **setup_kernel_seeds** pour initialiser tous les générateurs des threads
- dans notre kernel lui-même, d'utiliser notre fonction **uniform_random_number** pour obtenir un nouveau nombre aléatoire

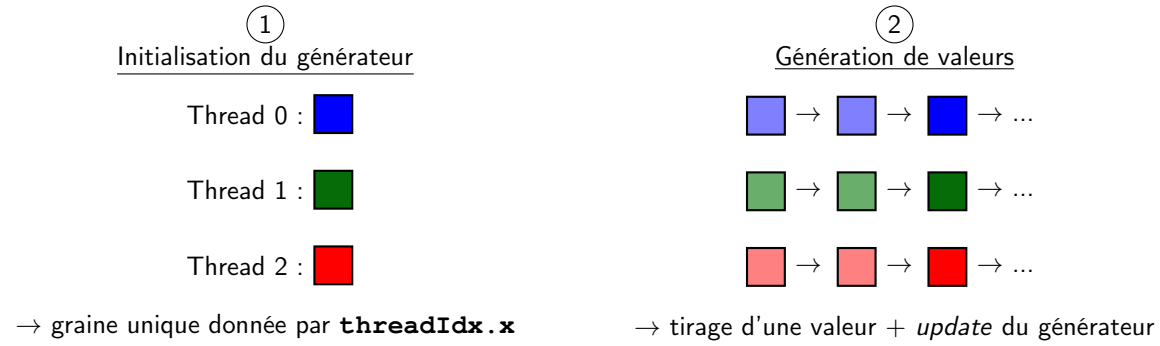


Figure 3 – *Vision globale du processus de génération de nombres pseudo-aléatoires*

Vérification de la validité des résultats
Pour vérifier que notre simulation parallèle sur GPU est correcte, on peut vérifier que l'on a bien traité les n neutrons demandés et que notre fichier `absorbed.dat` de sortie contient bien b lignes différentes (une pour chaque neutron absorbé).

Stockage des résultats
Pour stocker nos résultats, nous allons considérer un tableau global de résultats avec 4 cases (pour les neutrons **r**, **b** et **t** et l'indice du dernier neutron absorbé). Cela permettra la mise à jour correcte des compteurs sur GPU et le stockage contigu des positions des neutrons absorbés. Pour mettre à jour ce tableau, on peut envisager deux options : utiliser des opérations atomiques ou utiliser un algorithme de réduction.

L'avantage de la première technique est qu'elle est relativement simple à implémenter ; le problème est qu'elle "sérialise" les opérations et peut donc singulièrement ralentir le programme si beaucoup de threads doivent accéder à notre tableau de résultats ! Ici, il est donc impossible de procéder à des additions atomiques pour chaque neutron : il y aurait beaucoup trop de ralentissement. Mais, puisque l'on a "batché" notre traitement (en donnant un grand nombre de neutrons à chaque thread), on peut comptabiliser les résultats en local pour ces neutrons puis mettre à jour le tableau global à la fin des itérations.

En revanche, si l'on souhaite stocker les positions des neutrons absorbés de manière contiguë, on est obligé d'utiliser un opérateur atomique pour récupérer et incrémenter l'indice du dernier neutron absorbé à chaque fois qu'un neutron est absorbé, et que le tableau doit être mis à jour.

Les opérations atomiques sérialisent effectivement le code et ralentissent donc l'exécution ; on peut le vérifier

en demandant au GPU d'utiliser plus de blocs et de threads, puisque cela implique que les neutrons sont répartis sur plus de threads GPU différents qui vont chacun faire une opération atomique pour rassembler leurs résultats avec le reste. Quand on a des *batchs* plus petits, l'exécution est plus longue : il y a bien un ralentissement à cause des **`atomicAdd`**.

Validation des résultats

On peut vérifier que l'on obtient bien, à la fin, un nombre total de neutrons traités correct :

```
...
Nb blocs : 256    Nb threads par bloc : 512
(Avec reduction)
[GPU] Total treated: 54921969 + 216135018 + 228943013 = 500000000
[GPU] Temps calcul : 1.3977839947 seconde(s)

Pourcentage des neutrons reflechis : 0.11
Pourcentage des neutrons absorbes : 0.43
Pourcentage des neutrons transmis : 0.46

Temps total de calcul: 7.4983621 sec
Millions de neutrons /s: 67
```

On note également une accélération non négligeable, puisque le temps d'exécution est passé d'environ 70 secondes à seulement 7.5 secondes !

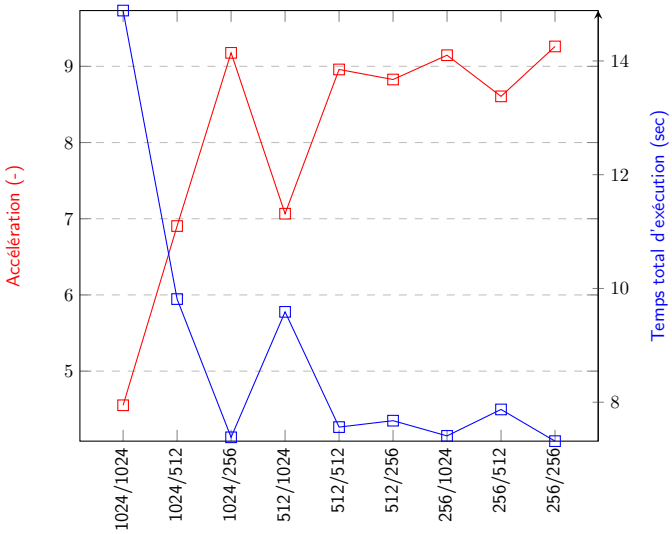
2.2 *Tuning* du nombre de blocs et de threads

Pour évaluer l'efficacité de l'algorithme, nous allons utiliser, entre autres, le critère de l'accélération. Celle-ci correspond au ratio de temps d'exécution entre algorithme séquentiel et algorithme parallèle. Rappelons que l'on a comme temps de référence séquentiel 67.754874 secondes.

La *Figure* 4 présente les résultats de temps d'exécution total et d'accélération pour différentes tailles de grille GPU[2]. On distingue dans notre étude deux temps (en secondes) :

- le temps de calcul (TC) : temps des calculs seuls sur le GPU
- le temps total d'exécution (TTE) : temps des allocations mémoire, de l'initialisation des générateurs, des calculs et des transferts mémoires (CPU vers GPU et GPU vers CPU)

Pour nos résultats, on utilise le TTE (en particulier, l'accélération est calculée par rapport à ce temps) pour prendre en compte le *bottleneck* d'allocation mémoire et comparer de manière plus "juste" avec le calcul sur CPU.



nbBlocks	nbThreads	TTE (sec)	Accélération
1024	1024	14.885916	4.552
1024	512	9.813597	6.904
1024	256	7.3832319	9.177
512	1024	9.5887001	7.066
512	512	7.562762	8.959
512	256	7.6755581	8.827
256	1024	7.4081511	9.146
256	512	7.872915	8.606
256	256	7.3158171	9.261

Figure 4 – *Temps total d'exécution et accélération sur un seul GPU selon la dimension de la grille GPU (**`nbBlocks/nbThreads`**)*

[2]Pour pallier aux différences d'ordres de grandeur, l'axe des abscisses est en échelle logarithmique.

Finalement, nos résultats semblent indiquer que la configuration optimale pour cet ordinateur est :

$$\texttt{nbThreads} = (256, 1, 1) \qquad \texttt{nbBlocks} = (256, 1, 1)$$

2.3 Seconde implémentation : *batching* "manuel"

Une autre possibilité est de choisir nous-mêmes la taille de nos *batchs* et de considérer que chaque thread GPU va traiter le même nombre de neutrons N. Ce sera alors la taille de la grille qui s'adaptera en fonction du nombre total d'échantillons à traiter. Même si cela est moins habituel, cela nous permet d'avoir un meilleur contrôle sur le traitement des neutrons par notre GPU.

L'idée est alors simplement de remplacer notre boucle **`while`** par une boucle **`for`** qui traite N neutrons et de lancer n/N threads sur notre GPU[3]. Pour notre *batching* "manuel", on va maintenant choisir :

$$\texttt{nbThreads} = (256, 1, 1) \qquad \texttt{nbBlocks} = (\lceil (n/N)/\texttt{nbThreads.x} \rceil, 1, 1)$$

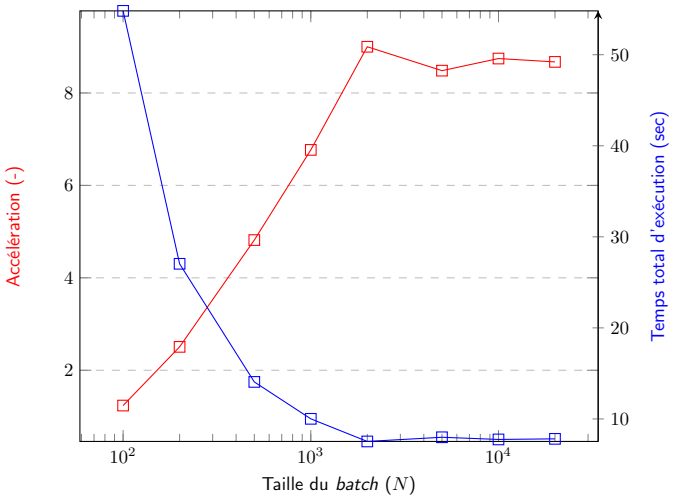
Cela va nous obliger à rechercher la taille optimale de *batch* pour obtenir le meilleur temps d'exécution possible.

2.4 *Tuning* de la taille du *batch* "manuel"

Dans un premier temps, on peut demander à nos threads d'utiliser des *batchs* assez petits de seulement $N = 200$ neutrons. Mais on obtient alors des résultats beaucoup moins bons qu'avec le *batching* automatique implémenté dans la première partie : environ 27 secondes au lieu de 8 secondes.

Nous allons donc maintenant étudier les performances de notre nouvelle implémentation selon la taille de *batch* pour déterminer un N optimal.

La *Figure* 5 présente les résultats de temps d'exécution total et d'accélération pour différents N[4]. Comme précédemment, l'accélération est calculée entre le temps de référence séquentiel et le temps total d'exécution (TTE) de notre code GPU.



Taille du *batch* (N)	TTE (sec)	Accélération
100	54.815639	1.236
200	27.0371	2.506
500	14.065251	4.817
1000	10.011685	6.768
2000	7.5284359	9.000
5000	7.9868741	8.483
10000	7.7476161	8.745
20000	7.813313	8.672

Figure 5 – *Temps total d'exécution et accélération sur un seul GPU selon la taille du batch*

On remarque qu'à partir de $N = 2000$, augmenter la taille des *batchs* n'a plus beaucoup d'influence. Afin de profiter au maximum de la parallélisation et d'exploiter au mieux notre GPU, on choisit donc comme N optimal : $N = 2000$.

[3] Pour simplifier, on suppose que le nombre d'échantillons à traiter est un multiple du nombre de *batchs* (sinon, on prévient l'utilisateur et on arrête l'exécution).

[4] Pour pallier aux différences d'ordres de grandeur, l'axe des abscisses est en échelle logarithmique.

2.5 Algorithme de réduction

En utilisant un algorithme de réduction, on peut réduire le nombre d'opérations atomiques et donc la "sérialisation" du code.

L'idée est d'utiliser la mémoire partagée au sein d'un bloc, puis de "réduire" les résultats à la fin des calculs et de demander à un seul thread du bloc de metter à jour le tableau de résultats global. De cette façon, au lieu d'avoir **nbBlocks** $\times$ **nbThreads** opérations atomiques, on n'en aura plus que **nbBlocks** (on divise donc le nombre d'opérations atomiques par la taille d'un bloc, soit 256 threads).

Cependant, l'utilisation de l'algorithme de réduction ne semble pas avoir d'effet sur notre cas de référence car le nombre de neutrons est probablement trop faible pour que l'écart soit suffisamment important. En effet, les **atomicAdd** sont malgré tout relativement bien gérés par le GPU : diviser le nombre d'opérations atomiques par 256 n'est pas suffisant pour faire une différence observable.

3 Parallélisation sur CPU

On souhaite maintenant effectuer une parallélisation de notre code sur un noeud de CPU multi-coeurs. Nous allons donc utiliser **OpenMP**.

Description générale du code OpenMP
On reprend l'idée du traitement "batché" et on garde le $N = 2000$ optimal identifié précédemment. Notre boucle procède bien à n itérations mais le **schedule** des threads **OpenMP** est fait de manière statique, avec des *chunks* de taille N.

Pour avoir un repère, on reprend le cas de référence : $H = 1.0$, $n = 500000000$, $c_c = 0.5$ et $c_s = 0.5$. Si on lance la simulation avec le code séquentiel fourni (sur CPU), on obtient un temps d'exécution de 76.78 secondes (l'ensemble des résultats de cette partie ont été obtenus sur le **pc4145**).

Génération de nombres pseudo-aléatoires
Nous devons générer des nombres aléatoires. Pour cela, on se base sur les fonctions fournies qui utilisent les fonctions Unix disponibles **srand_48** et **drand_48**. On va se servir de l'indice (unique) du thread **OpenMP** comme graine d'initialisation du générateur. Comme précédemment, la génération se passe en deux temps : d'abord, il faut initialiser le générateur de nombres aléatoires de chaque thread, puis ensuite on peut y accéder pour obtenir un nouveau nombre aléatoire aisément, dès que nécessaire.

On reprend les deux fonctions fournies dans le code séquentiel en ajoutant la possibilité de passer une graine au générateur.

La première fonction (**init_uniform_random_number**) permet d'initialiser le générateur de nombres aléatoires de chaque thread à partir de son identifiant unique. La deuxième fonction (**uniform_random_number**) permet d'obtenir un nombre aléatoire à partir du générateur pour un thread donné (et de faire progresser le générateur pour la prochaine fois). Il nous suffit donc maintenant, dans chacun de nos thread (dans la section parallèle de notre code) :

- d'initialiser les générateur de chaque thread avec son identifiant unique
- dans la suite de sa procédure, d'utiliser notre fonction **uniform_random_number** pour obtenir un nouveau nombre aléatoire

Réduction OpenMP et opérations atomiques
On va utiliser le principe de réduction disponible avec **OpenMP**. En effet, une fois que chaque thread **OpenMP** a fini de calculer son *batch* de N neutrons, il lui suffit d'ajouter le total de neutrons réfléchis, absorbés et transmis aux compteurs globaux. On peut donc "réduire" par addition les compteurs **r**, **b** et **t** à l'aide de l'instruction **reduction(+:r,b,t)**.

On a ici 5 variables privées (qui dépendent de chaque thread) : **x**, **d**, **L**, **u** et l'indice i qui parcourt la boucle. Nous devons donc préciser dans notre région parallèle, définie par un **#pragma omp parallel**, l'instruction **private(x,d,L,u,i)**.

En revanche, le compteur global j (pour le stockage contigu) et le tableau **absorbed** sont partagés entre les threads (et on les laisse donc à leur état par défaut de variable **OpenMP** *shared*).

Comme pour la parallélisation sur GPU, nous sommes confrontés à une difficulté pour le stockage contigu des positions de neutrons absorbés : si on souhaite que l'indice global soit incrémenté correctement, nous devons utiliser une opération atomique **OpenMP**, avec l'utilisation d'un **#pragma omp atomic**[5].

On peut vérifier que l'on obtient bien, à la fin, un nombre total de neutrons traités correct :

```
...
(Nb threads: 2)
r = 54553518, b = 215611191, t = 229835291
Total treated: 500000000

Pourcentage des neutrons reflechis : 0.11
Pourcentage des neutrons absorbes : 0.43
Pourcentage des neutrons transmis : 0.46

Temps total de calcul: 67.316801 sec
Millions de neutrons /s: 7.4
```

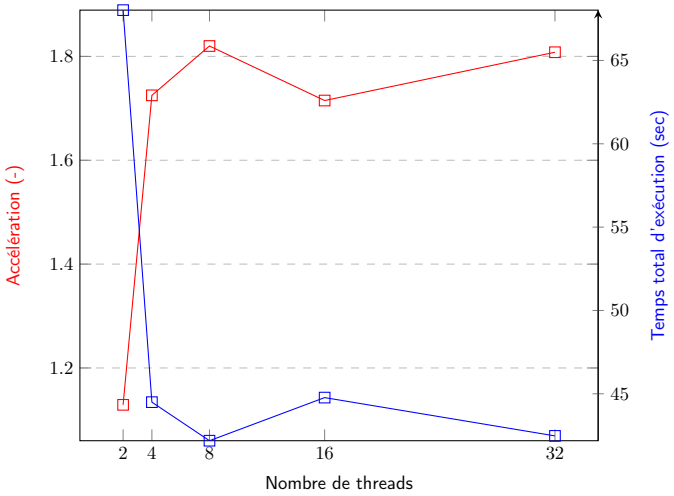
On note également une accélération intéressante (même si elle est moins importante que l'accélération sur GPU), puisque le temps d'exécution est passé d'environ 76 secondes à seulement 67 secondes.

La limitation sur l'accélération est probablement due aux opérations atomiques et aux accès concurrents des threads au tableau partagé **absorbed**.

Equilibrage de charge
L'équilibrage de charge est fait de manière statique en considérant, comme dans la parallélisation sur GPU, une taille optimale de *batch* de 2000 neutrons. En effet, même si le traitement est théoriquement hétérogène pour chaque neutron (dans le sens où l'un peut être absorbé très vite, un autre être transmis après un temps très long, etc.), cette taille est suffisamment grande pour amortir ces variations et globalement équivalent à un équilibrage de charge dynamique.

3.1 *Tuning* du nombre de threads **OpenMP**

Une rapide nous montre (voir la *Figure* 6) qu'il est inutile de dépasser 8 threads **OpenMP** : au-delà, on travaille avec des processeurs virtuels et il y a donc en réalité une sérialisation de notre code par la machine.



Nombre de threads	TTE (sec)	Accélération
2	68.019003	1.129
4	44.498004	1.725
8	42.18426	1.820
16	44.771385	1.715
32	42.472989	1.808

Figure 6 – *Temps total d'exécution et accélération avec parallélisation* ***OpenMP*** *selon le nombre de threads*

[5] On aurait également pu utiliser un **#pragma omp critical**, qui s'applique à plus de cas, mais dans notre situation cela ralentit l'exécution. On note cependant que l'opération atomique n'est possible que parce qu'ici on ne fait qu'une incrémentation simple.

3.2 Influence de la capacité d'absorption

On cherche ici à étudier l'impact de la probabilité d'absorption des neutrons sur le temps d'exécution pour notre code sur GPU en **CUDA** et notre sur CPU en **OpenMP**. On choisit ici **NB_OMP_THREADS** = 8 et on prend notre première implémentation sur GPU avec *batching* automatique.

Nous allons donc augmenter la variable c_c et diminuer c_s (on sait que les deux probabilités doivent sommer à 1). On essaie, par exemple, les valeurs suivantes (et on indique à côté l'accélération GPU et l'accélération CPU) :

c_c	c_s	TTE sur GPU (sec)	Accélération (GPU)	TTE sur CPU (sec)	Accélération (CPU)
0.1	0.9	8.1157441	8.349	55.02609	1.395
0.3	0.7	7.5799749	8.939	47.187851	1.627
0.5	0.5	7.468183	9.072	42.184265	1.820
0.7	0.3	7.270118	9.320	37.945734	2.023
0.9	0.1	7.2252488	9.378	32.260286	2.380

On rappelle que nos temps séquentiels de référence sont de 67.754874 secondes pour le ***pc5016*** *(code GPU) et 76.775952 secondes pour le* ***pc4145*** *(code CPU). L'accélération est toujours calculée à partir du temps total d'exécution (****TTE****).*

On remarque que, sur CPU, augmenter la probabilité d'absorption réduit beaucoup le temps d'exécution (on a une accélération 1.706 fois plus importante). En effet, beaucoup de neutrons sont traités plus rapidement (car on calcule moins d'itérations : on s'arrête dès qu'ils ont été absorbés !) et donc l'exécution est globalement plus rapide.

En revanche, la différence est beaucoup moins notable (l'accélération est seulement 1.123 fois plus importante) pour notre parallélisation GPU avec **CUDA** car nos threads parallélisent déjà de manière très efficace et que le temps de calcul est déjà très faible (presque tout le temps de traitement vient des allocations mémoire).

4 Parallélisation hybride : CPU + GPU

4.1 Utilisation de OpenMP

Dans cette dernière partie, nous allons chercher à équilibrer la charge de calcul entre CPU et GPU. On va de nouveau utiliser **CUDA** et **OpenMP** en demandant à notre CPU de traiter une partie des neutrons à l'aide de plusieurs threads **OpenMP**, et en même temps à notre GPU de traiter le reste des neutrons.

Malheureusement, nos tests ont vite montré qu'il y avait une séquentialisation de l'appel GPU et des threads **OpenMP** avec une monopolisation du thread **OpenMP** *master* (autrement dit, d'indice 0). Celui-ci, après avoir lancé la simulation sur GPU, attend la fin de cet appel pour continuer le processus et on ne peut donc pas améliorer nos performances de cette manière.

Ce problème vient *a priori* d'une attente "agressive" côté GPU, qui prend la main pour (re)lancer et exécuter rapidement les kernels. Notamment, cela force le thread qui gère le GPU à monopoliser le processus, même pour finalement attendre les résultats du calcul !

Plutôt que de séparer au départ l'ensemble de neutrons à traiter entre GPU et CPU, on pourrait envisager (comme l'ont fait d'autres groupes) un équilibrage dynamique. L'idée serait de créer une "*pool*" de neutrons à traiter, à laquelle tous les threads **OpenMP** peuvent accéder (y compris le thread *master*). De cette manière, on relance un calcul de *batch* de neutrons sur GPU, ou bien un calcul de neutron seul sur CPU, tant qu'il reste des neutrons à traiter dans la *pool*.

Cela permettrait de forcer l'interaction entre le thread **OpenMP** *master* (monopolisé par le GPU) et les autres.

4.2 Utilisation de **MPI**

4.2.1 Description de l'algorithme

Pour éviter le problème de séquentialisation, on peut aussi ruser en lançant plusieurs processus (au lieu de plusieurs threads) à l'aide de la bibliothèque **MPI**.

On va définir un ratio d'équilibrage r entre GPU et CPU : si r vaut 0, alors tous les calculs s'effectueront sur le GPU ; si r vaut 1, alors tous les calculs s'effectueront sur le CPU. On va considérer deux cas :

- $r = 1$: il n'y a aucun traitement GPU
- $0 < r < 1$: une partie des neutrons est traitée par le CPU (répartie entre plusieurs processus **MPI**), le reste par le GPU

Dans le premier cas, tous nos processus **MPI** vont se répartir les neutrons à traiter, puis le processus *master* (d'indice 0) récupèrera les résultats pour les rassembler. Dans le second cas, les processus *non-masters* gèreront le traitement d'une partie des neutrons et le processus *master* gèrera le traitement GPU du reste puis le rassemblement des résultats. Dans les deux cas, le processus *master* commence par envoyer à chacun des autres processus le nombre de neutrons qu'il doit traiter.

4.2.2 Résultats

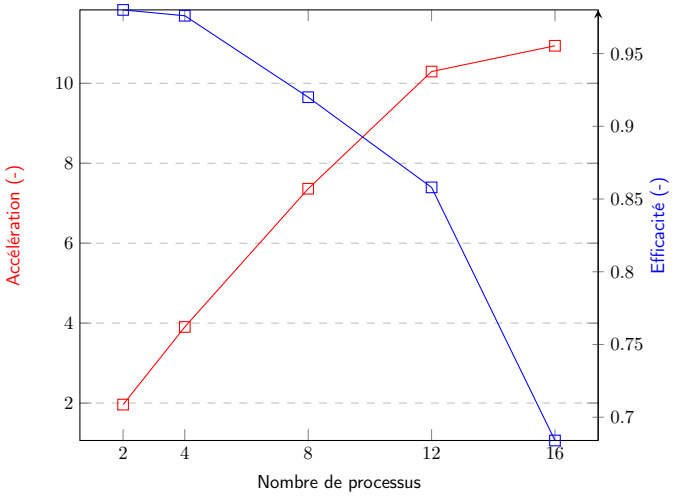
On peut maintenant faire varier 2 facteurs :

- le nombre de processus **MPI** (**NB_MPI**)
- le ratio d'équilibrage (r)

A cause du temps incompressible lié à l'allocation mémoire sur GPU mentionné plus tôt dans le dossier, on peut espérer une amélioration du temps d'exécution avec la parallélisation hybride mais seulement jusqu'à un certain point : dès lors qu'une partie du traitement est donnée au GPU, il y aura ce délai.

Les tests de cette partie ont été effectués sur le **pc5016**. Pour rappel, on a donc un temps d'exécution séquentiel de 67.754874 secondes.

Pour commencer, déléguons 0% du travail au GPU, 100% aux CPUs et faisons varier le nombre de processus **MPI**. Etant donné que la charge de travail est entièrement sur CPU, tous les processus (y compris le *master*) participent au traitement.

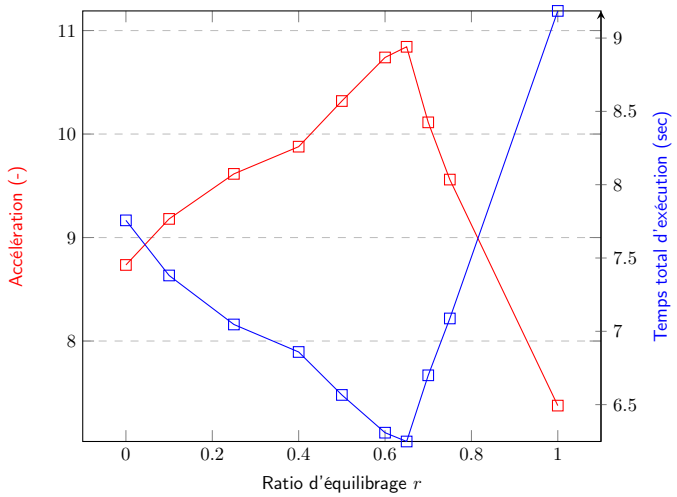


NB_MPI	TTE (sec)	Accélération	Efficacité
2	34.552466	1.961	0.980
4	17.355921	3.904	0.976
8	9.2038031	7.362	0.920
12	6.5819759	10.294	0.858
16	6.193943	10.939	0.684

Figure 7 – *Temps total d'exécution et accélération avec parallélisation **CUDA**+**MPI** selon le nombre de processus **MPI***

On voit que, pour **NB_MPI** $= 8$, on a une bonne accélération et une efficacité correcte (pour un nombre plus élevé de processus, le gain n'est pas très important mais on consomme en revanche beaucoup plus de ressources). Retenons donc **NB_MPI** $= 8$ comme nombre optimal de processus **MPI**.

Nous allons maintenant chercher le ratio optimal d'équilibrage de la charge de travail entre GPU et CPU. On prend **NB_MPI** $= 9$ processus car, lorsque le traitement est réparti entre GPU et CPU, seulement **NB_MPI** $- 1$ processus **MPI** traitent des neutrons sur CPU (le processus *master* gère le GPU).



Ratio d'équilibrage r	TTE (sec)	Accélération	Efficacité
0.0 (GPU seul)	7.7567852	8.735	1.092
0.1	7.3808231	9.180	1.147
0.25	7.0466199	9.615	1.202
0.4	6.8589909	9.878	1.235
0.5	6.5660269	10.319	1.290
0.6	6.3085289	10.740	1.343
0.65	6.2485471	10.843	1.355
0.7	6.699934	10.113	1.264
0.75	7.087625	9.560	1.195
1.0 (CPU seul)	9.1852648	7.376	0.922

Figure 7 – *Temps total d'exécution et accélération avec parallélisation* ***CUDA+MPI*** *selon le ratio d'équilibrage GPU/CPU*

D'après nos résultats montrés sur la *Figure* 7, il semble qu'un ratio de 35%/65% entre GPU et CPU est optimal : par rapport à notre implémentation sur GPU seul (avec une taille de grille optimisée, **nbBlocks** $= 256$, **nbThreads** $= 256$), on est passés d'une accélération de 9.261 à une accélération de 10.843 !

5 Pistes de développement

Ces différentes pistes de réflexion n'ont pas été développées, ou pas entièrement. Ce sont pour la plupart des ébauches qui, en théorie, pourraient encore améliorer l'accélération de notre parallélisation.

5.1 Contournement de l'`atomicAdd()`

On peut chercher à contourner l'**atomicAdd** jusqu'ici nécessaire pour le stockage contigu des positions de neutrons absorbés. En s'inspirant de la documentation Nvidia[6], on peut mettre en place un algorithme de somme préfixée pour regrouper des résultats stockés de manière éparse dans un tableau en un stockage contigu. L'algorithme repose sur la création d'un tableau de "masque" qui repère les positions remplies dans le tableau épars, puis sur l'utilisation de ce masque pour réaliser des sommes partielles qui réalisent le compactage des valeurs.

[6] https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html

5.2 Parallélisation avec plusieurs *streams*

En restant sur un seul GPU, on pourrait essayer d'augmenter sa capacité de travail en parallélisant le traitement sur plusieurs *streams* **CUDA**. Au lieu d'avoir seulement le *stream* 0 d'actif, on pourrait en utiliser plusieurs et ainsi recouvrir une partie des calculs et des accès mémoire. Cependant, cela complexifierait la gestion de la mémoire dans notre code.

5.3 Parallélisation multi-GPUs

Pour pallier au problème de la sérialisation évoqué précédemment, on peut décider de plutôt profiter de la puissance GPU de notre ordinateur qui dispose de 3 GPUs.
Note : on travaille toujours sur le ***pc5016****.*

L'idée est ici de lancer notre simulation sur GPU en répartissant la charge de travail entre les différents GPUs. Cette idée a été **presque complètement implémentée** ; cependant, elle donne une erreur : "`all CUDA-capable devices are busy or unavailable`"[7].

Une autre solution serait de partir de notre version hybride **CUDA–MPI** pour lancer plusieurs processus en parallèle, chacun lançant un kernel sur un GPU différent.

5.4 Optimisation des communications MPI

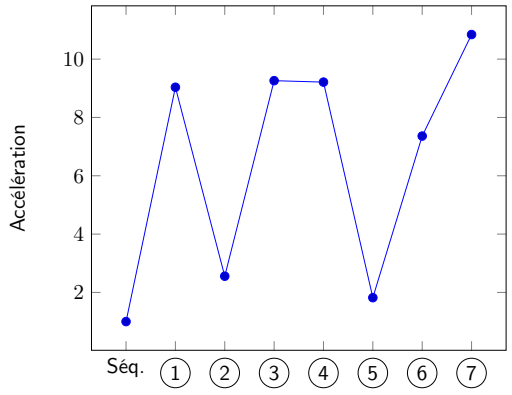
Enfin, notre code hybride **CUDA–MPI** n'est pas optimisé du point de vue des communications. En particulier, il serait plus efficace de remplacer les multiples **MPI_Send** et **MPI_Recv** par des routines collectives comme le **MPI_Broadcast** et le **MPI_Gather**. En effet, ces routines sont étudiées pour implémenter des arbres de communication beaucoup plus rapides à parcourir. Mais, encore une fois, cela complexifierait la gestion de la mémoire dans le code.

6 Conclusion

A travers ce projet, on a pu étudier et comparer différentes techniques de parallélisation sur un problème physique simple : **CUDA** sur GPU, **OpenMP** sur CPU et **MPI** sur CPU. On a ainsi réduit significativement le temps d'exécution avec, pour notre implémentation hybride la plus performante, une accélération de 10.

On a pu remarquer qu'il est relativement simple d'implémenter une parallélisation à l'aide d'**OpenMP** mais que si l'on veut obtenir de meilleurs résultats, l'utilisation du GPU avec un code en **CUDA** est bien plus intéressante. En particulier, coupler le GPU à une parallélisation de processus **MPI** améliore encore notre temps d'exécution. Il y a cependant des limites à l'accélération possible à cause des transferts mémoire entre CPU et GPU.

La *Figure* 8 montre l'accélération entre le temps séquentiel et le temps total d'exécution de notre code parallèle pour les différents algorithmes implémentés pendant le projet.



1	GPU (*batching* automatique simple)
2	GPU (*batching* "manuel" simple)
3	GPU (*batching* automatique optimisé)
4	GPU (*batching* "manuel" optimisé)
5	CPU (parallélisation **OpenMP**)
6	CPU (parallélisation **MPI**)
7	Hybride (**CUDA**+**MPI**)

Figure 8 – *Accélération pour les différents algorithmes du projet*

[7] D'après des recherches Internet sur les forums Nvidia, il semble que cela puisse être lié à l'installation de **CUDA** et notamment à des problèmes de *drivers*...