CUDA TP3
Synchronisation des threads, mémoires, streams

1 Exécution des threads

Dans les exercices précédents, les threads accèdent toujours à des zones mémoires distinctes. Que se passe-t-il si différents threads doivent échanger des données / accéder à des zones mémoires en lecture et écriture ?

1.1 Inversion des éléments d'un tableau - version "out of place"

Cette version dite "out of place" utilise des tableaux d'entrées et sorties distincts.

1. Créer le kernel implantant cette algorithme : chaque thread récupère l'élément `size - tid -1` dans le tableau d'entrée et le place à la position `tid` dans le tableau de sortie.

2. Tester sur des tableaux de grandes tailles avec des configurations de grilles et de blocs de threads différentes.

Dans ce cas il n'y a aucun problème de synchronisation !

1.2 Inversion des éléments d'un tableau - version "in place"

Cette version dite "in place" inverse les éléments directement dans le tableau d'entrée.

1. Essayer l'algorithme précédent en donnant le même tableau en entrée et en sortie et en créant plusieurs blocs de threads. Quel est le résultat ?

2. Le problème est que le thread `tid` peut écrire une donnée dans le tableau alors que le thread `size - tid - 1` va lire cette donnée. Si les 2 threads se trouvent dans 2 warps différents il n'y a pas de garantie sur l'ordre d'exécution de ces threads.

3. La seule méthode disponible pour synchroniser des threads pendant l'exécution d'un kernel est de faire appel à la fonction `__syncthreads` qui permet seulement de synchroniser les threads (warps) d'un même bloc. Reprendre l'exercice en instantiant uniquement un bloc de threads et en plaçant judicieusement un appel à la fonction `__syncthreads`.

4. Une autre solution sans besoin de synchronisation est d'utiliser 2 fois moins de threads, chaque thread s'occupant de lire les valeurs aux positions `tid` et `size - tid - 1` et effectuant l'échange à l'aide d'une variable temporaire.

Un autre exemple pour illustrer la synchronisation est celui de la réduction.

2 Shared memory

La shared memory est une zone mémoire qui se trouve dans le processeur et qui est aussi rapide d'accès que les registres. Chaque multiprocesseur possède une partie de cette shared memory qui permet donc aux threads d'un même bloc de stocker et d'échanger des données. Cette mémoire doit être gérée explicitement par le développeur. La taille de la mémoire chared peut être définie suivant 2 méthodes :

- en dur dans le kernel,
- lors de l'appel au kernel.

Tester les 2 versions de l'exemple Using Shared Memory in CUDA C/C++.

3 Streams

Les streams permettent de découper un flot de traitement (copies mémoires, exécution de kernels) en plusieurs sous flots pour effectuer du recouvrement entre le calcul et les communications. Utiliser la documentation(lien) relative aux streams pour adapter le code de traitement d'image du TP précédent. Créer 2 streams s'occupant chacun d'une moitié de l'image.

4 Profiling

Pour étudier l'impact de vos optimisations sur les performances, il est possible soit d'utiliser les `cudaEvents` pour mesurer les temps de communications et d'exécution des kernels. Cependant pour étudier l'impact des streams, le plus simple est d'utiliser le profiler `nvprof` qui permet d'obtenir ces informations. La sortie de `nvprof` peut également être analysée à l'aide d'une interface graphique appelée Visual Profiler `nvvp`.

Un exemple d'utilisation est disponible ici.