TD GPGPU n°3

GPGPU Courses
MII 2 option Sciences de l'Image et IMAC 3 option Prog 3D

Réduction en Cuda

Mémoire partagée, synchronisation, optimisation

Dans ce TP, l'algorithme consiste à trouver la valeur maximale dans un tableau de taille N . Cet algorithme est trivial en version séquentielle. En version parallèle, on utilise un mécanisme de réduction pour éviter les accès concurrents à la mémoire et valider le résultat final.

Sur GPU, implémenter efficacement une réduction implique d'utiliser la mémoire partagée et les dispositifs de synchronisation. L'idée (dans ce TP) est de trouver la valeur maximale partielle par *block* (en mémoire partagée) puis de centraliser les résultats sur CPU pour terminer le calcul.

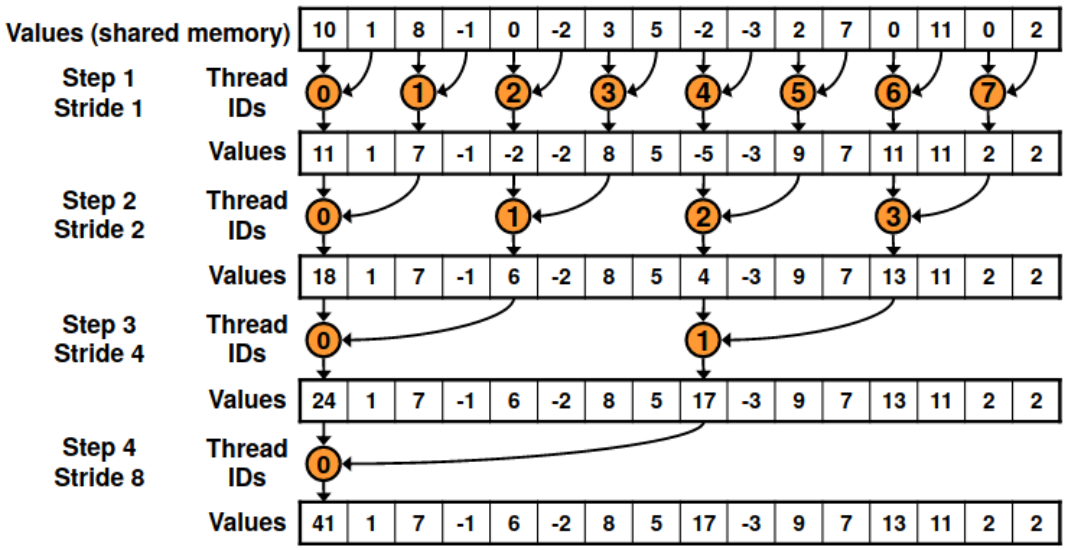
Ces exercices vont aussi nous servir à mettre en avant quelques stratégies d'optimisation importantes en CUDA. Au fur et à mesure, vous devrez implémenter des *kernels* de plus en plus efficaces. Prenez bien le temps de comprendre les problèmes évoqués ainsi que leur solution algorithmique.

Notes

1. Il est possible d'allouer dynamiquement la mémoire partagée, en fonction de la taille nécessaire pour exécuter votre algorithme. Ceci se fait en deux étapes :
   a) Déclarer la mémoire partagée en utilisant le mot clé extern (dans le kernel) :
   `extern __shared__ TYPE sharedMemory[];`
   b) Préciser la taille allouée lors de l'appel du kernel :
   `kernel<<<dimGrid,dimBlock,sizeSharedMemory>>>(...)`
2. Les fonctions `configureKernel` et `reduce` sont des fonctions templates (donc codées dans le header) que vous devrez compléter pour chaque exercice.
3. Regardez l'initialisation pseudo-aléatoire (et bizarre) des données (`main.cu`, l.85). Comme vous pouvez le remarquer, il y a 50% de chance que le max soit placé en dernière position. Essayez donc plusieurs fois chaque kernel.
4. L'option `-l <nb>` spécifie le nombre de fois où le kernel sera lancé pour mesurer un temps de calcul moyen.

Exercice 1 : Vous avez dit « réduction » ?

Pour ce premier exercice, la réduction est effectuée par adressage entrelacé, comme illustré ci dessous (attention, sur le schéma, il s'agit d'une somme... mais le principe est le même pour trouver le maximum !) :



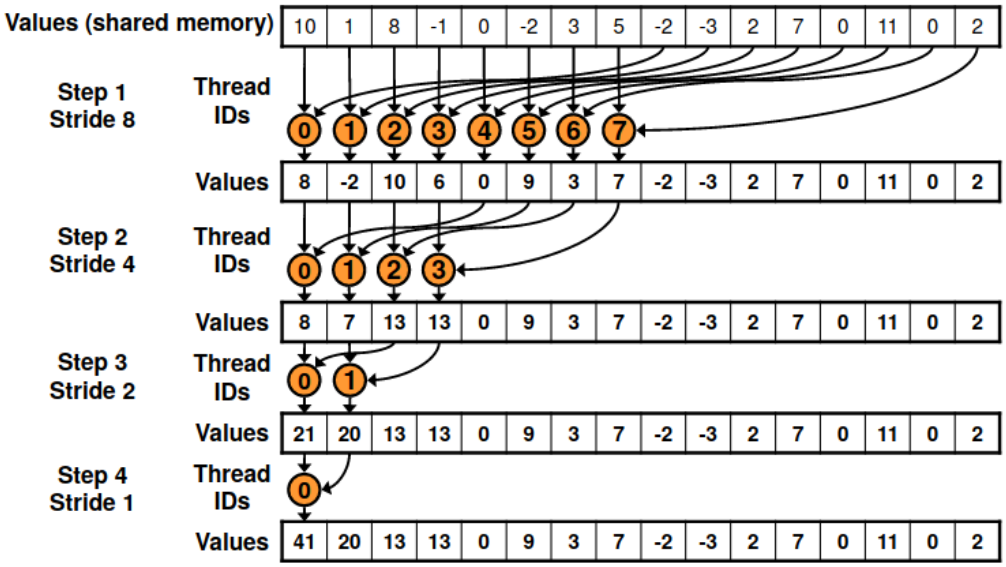
Les TD ont été écrit très majoritairement par Maxime Maria (maintenant Maître de Conférences au XLIM à Limoges)

1. Implémentez le kernel `maxReduce_ex1` et complétez la fonction `reduce` afin d'effectuer la réduction par adressage entrelacé (utilisez `umax`). La mémoire partagée doit être allouée dynamiquement.
2. Par défaut, la taille du tableau est fixée à 2^{23} . Augmentez la taille du tableau (via l'option `-n <size>` ou `-2n <log(size)>`). À partir de 2^{26}, le résultat devient faux… Pourquoi ? Modifiez la fonction `configureKernel` pour que la réduction fonctionne avec de grands tableaux.
3. Modifiez la fonction `configureKernel` pour éviter de lancer trop de threads, si la taille du tableau est plus petite que le nombre de threads fixé (une fonction `nextPow2` est fournie dans le fichier `common.hpp`).

Exercice 2 : Braquage de banque

La mémoire partagée est divisée en plusieurs parties de même taille appelées « banques ». Les accès aux adresses mémoires situées dans des banques différentes peuvent se faire simultanément. Cependant si deux adresses pointent vers la même banque, les accès se font de manière séquentielle. La parallélisation est donc moins efficace : il y a « conflit de banque ».

1. Afin d'éviter les conflits de banque, implémentez le kernel `maxReduce_ex2` qui effectue la réduction par adressage séquentiel, comme illustré ci-dessous :



Exercice 3 : Évitons la famine !

Lors du premier tour de boucle de l'algorithme précédent, la moitié des *threads* est inactive. Pour profiter complètement de la parallélisation, il est nécessaire de maximiser l'activité des *threads*.

1. Implémentez le kernel `maxReduce_ex3`. Il effectue le premier niveau de la réduction dès le chargement des données depuis la mémoire globale, profitant ainsi de l'ensemble des threads. (Notez qu'il nécessite donc deux fois moins de blocks).

Exercice 4 : On déroule le wrap !

Il faut savoir que les *threads* travaillent de façon SIMD (*Single Instruction Multiple Data*) par groupes de 32 appelés *warps*. En d'autres mots, tous les *threads* exécutent la même instruction en parallèle au sein d'un même *warp*. Pour la réduction, quand `id < 32`, il ne reste qu'un seul *warp* en cours d'exécution. Il n'est donc plus nécessaire de s'encombrer de la boucle, du test conditionnel et de la synchronisation... On peut donc dérouler les 6 dernières itérations de la boucle.

1. Implémentez le kernel `maxReduce_ex4` dans lequel le dernier *warp* de chaque *block* est déroulé. Attention, vous devez déclarer une copie volatile du pointeur vers la mémoire partagée pour éviter que le compilateur n'optimise les accès et induise un comportement erroné.

N.B. : À partir de la version 7.0 de CUDA, il est possible d'utiliser l'instruction compilateur `#pragma unroll` (vous pouvez l'essayer si possible, mais pas pour répondre à l'exercice).

Exercice 5: (Bonus) C’est qui le patron ?

Si le nombre d’itérations est connu au moment de la compilation, il est facile de dérouler complètement l’algorithme. Sur nos GPUs, le nombre de threads par block est limité à 1024, et nous utilisons uniquement des blocks de taille 2^n ...

1. Déroulez complètement l’algorithme ! Utilisez une fonction template pour avoir un code générique... ;-)