Programmation GPGPU avec CUDA - TP 3

Principe de réduction en CUDA

neals.fournaise@unilim.fr maxime.maria@unilim.fr

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 mémoire concurrents 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.

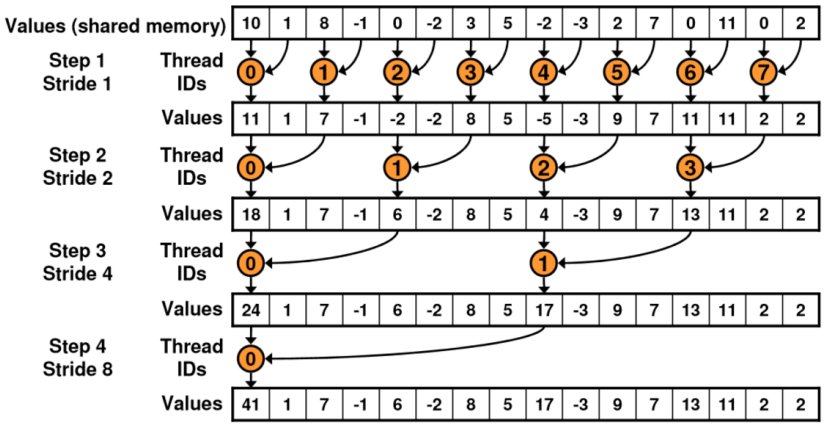
Note technique

Dans le cours, la taille de la mémoire partagée est définie statiquement en utilisant une variable globale. Or, il est possible de l'allouer dynamiquement, en fonction de la taille nécessaire pour exécuter votre algorithme. Ceci se fait en deux étapes :

1. Déclarer la mémoire partagée en utilisant le mot clé `extern` (dans le *kernel*) :
   — `extern __shared__ TYPE sharedMemory[];`.
2. Préciser la taille allouée lors de l'appel du *kernel* :
   — `kernel<<<dimGrid,dimBlock,sizeSharedMemory>>>(...)`

Exercice 1 : Vous avez dit réduction ?

Pour ce premier exercice, la réduction (`maxReduce_v0`) est effectuée comme illustré ci dessous (attention, sur le schéma, il s'agit d'une somme... mais le principe est le même pour un max !) :



Q.1. Compilez et exécutez le programme. Il ne fonctionne pas... Pourquoi ?

Q.2. Modifiez le *kernel* pour que la réduction se passe sans accroc.

Exercice 2 : Gonflé à bloc

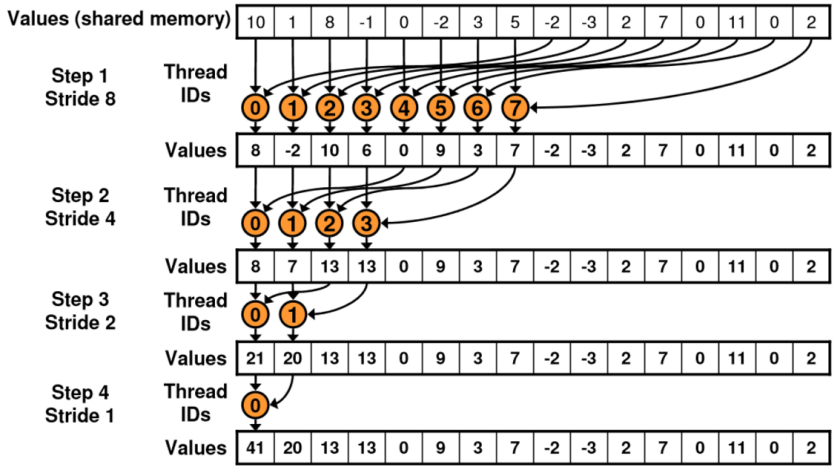
Q.1. Le nombre de *blocks* est défini statiquement dans la fonction `launchKernel`. Connaissant le nombre de *threads* par *block*, modifiez l'affectation de la variable pour qu'elle s'adapte dynamiquement à la taille du tableau. Vérifiez votre résultat en exécutant le programme.

Q.2. Augmentez la taille du tableau (option `-n size` ou `-2n log(size)`) jusqu'à atteindre la limite maximale de *blocks*. Comment pouvez vous pallier ce problème ? Quel est l'impact de cette modification sur les performances (pour des tableaux de taille égales) ? Quelle est la taille limite sur votre GPU ?

Exercice 3 : 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 ».

Q.1. Afin d'éviter les conflits de banque, implémentez le *kernel* `maxReduce_v1` qui effectue la réduction comme illustré ci-dessous :



Q.2. Analysez l'apport de cette optimisation.

Exercice 4 : 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*.

Q.1. Implémentez le *kernel* `maxReduce_v2`. 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*).

Q.2. Comparez le comportement (accès mémoire globale, nombre d'instructions) et les temps d'exécution de l'algorithme avec le précédent.

Exercice 5 : On déroule le wrap !

Il faut savoir que les *threads* travaillent par groupe de 32 appelés *warps*. Au sein d'un même *warp*, tous les *threads* exécutent la même instruction en parallèle.

Pour la réduction, quand `i < 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.

Q.1. Implémentez le *kernel* `maxReduce_v3` 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.

Q.2. Une nouvelle fois, comparez et analysez les résultats.

N.B. : Il est possible de dérouler complètement l'algorithme si le nombre de *threads* par *block* est fixé (ou en utilisant une fonction *template*... ;-)).