MS-SIS - Programmation GPU

(2021-2022)

TP

CUDA

adrien.roussel@cea.fr

Les objectifs de ce TP sont :
— Compréhension du modèle déxécution
— debugging
— Ecriture d'un premier code CUDA
— Utilisation efficace d'un GPU

I Modèle d'exécution et SDK CUDA

Dans cette partie nous considérons le fichier Ex1.cu. À tout moment, vous pouvez compiler le fichier l'exécuter (commande **nvcc Ex1.cu -o Ex1.pgr**).

Q.1: Quelle partie du programme doit s'exécuter sur l'hôte ? Quelle partie sur le device ?

Q.2: Que calcule ce programme ? (si vous ne savez pas répondre à cette question, répondre à la suivante pourra vous aider).

Q.3: Combien y a-t-il de blocs au total ? Combien de threads par blocs ? Combien de threads au total ?

Q.4: Émuler sur CPU le comportement du GPU sans utiliser le SDK CUDA. Pour ce faire, réécrire le programme en C/C++ avec les contraintes suivantes :

1. utilisation d'une fonction `kernel`
2. utillisation des grilles de blocs et de threads

II Debugging

Nous considérons à présent le fichier *err1.cu*. Ce programme est censé calculer la somme de deux vecteurs. À la fin de l'exécution, une erreur relative est calculée entre le vecteur issu du GPGPU et celui calculé sur CPU.

Q.5: Compiler et exécuter le programme. Le résultat est-il correct ?

Q.6: Encadrer chaque appel à CUDA par la macro `checkCudaErrors` .

Q.7: Calculer le nombre total de threads. Comparer le à N. Qu'en déduire ?

Q.8: Corriger le code CUDA selon les deux possibilités suivantes :

1. Corriger le nombre de blocs pour traiter tous les indices des tableaux.
2. Sans changer les tailles de la grille et des blocs, modifier le kernel (prendre en compte le nombre total de threads).

III Traduction d'un code C en code CUDA

Dans cette partie nous considérons le fichier Ex3_1.c présent dans le répertoire *Emul_CPU*. Ce code simple réalise l'initialisation d'un tableau `a`, tel que `a[i]=i`, par le biais d'une indirection avec un tableau `b`. Une fonction `check_a` permet de vérifier que l'initialisation du tableau `a` s'est bien passée.

Q.9: Le squelette de la traduction en CUDA du programme contenu dans le fichier Ex3_1.c vous est fourni dans le répertoire *CODE* (fichier *Ex3_1.cu*. A vous de remplir ce squelette (parties notées "A COMPLETER" dans le code) pour réaliser l'initialisation du tableau `a` sur le GPU.

Q.10: Le schéma d'accès aux données du tableau `a` est-il efficace ? pourquoi ?

Q.11: Nous allons vérifier cette hypothèse. Insérer des appels à `gettimeofday` pour mesurer le temps du kernel CUDA (et uniquement du kernel). Relevez le temps mesuré.

Q.12: Nous allons maintenant changer le schéma d'accés aux données du tableau `a` en modifiant les valeurs dans le tableau d'indirection `b`. Remplacez la valeur actuelle de `STEP` par 1. Que cela change-t-il pour les accés au tableau `a` ? Relevez le temps mesuré. Est-il meilleur ? Pourquoi ?

Q.13: Jusqu'à présent, nous n'utilisions qu'un seul bloc de plusieurs threads. Nous allons changé cela. Modifiez la valeur de `nBlocks` pour la mettre à 16. Modifiez votre kernel pour avoir un calcul d'indice correct. Relevez le temps mesuré. Est-il meilleur ? Pourquoi ?

IV Réduction somme en CUDA

Une réduction somme consiste à additionner toutes les valeurs d'une tableau. Une écriture séquentielle d'une réduction pourrait être la suivante :

```
float sum = 0;
for (int i = 0; i < ntot; i++)
    sum += tab[i];
```

On désire écrire une implémentation CUDA de la réduction somme (*tab* se trouve sur le GPU). Pour implémenter la réduction, nous proposons d'opérer en deux étapes :

1. Une première réduction dans chaque bloc. On obtient ainsi à la fin un tableau dimensionné au nombre de blocs et dont les valeurs sont les sommes partielles de chaque bloc.

2. Une seconde réduction sur les sommes partielles. On obtient ainsi la somme totale des éléments du tableau.

Q.14: Implémenter le kernel `reduce_kernel(float *in, float *out)` (voir fichier *reduce.cu*) permettant de faire les sommes partielles par bloc.
in est le tableau de valeurs à réduire dimensionné au nombre total de threads dans la grille, et *out* le tableau de valeurs réduites par bloc, dimensionné au nombre de bloc.
Pour réaliser cette réduction, vous utiliserez une méthode arborescente, ainsi que la fonction `__syncthreads()` qui permet de synchroniser à l'intérieur d'un kernel tous les threads d'un même bloc.
Nous nous placerons sous les hypothèses suivantes :

— Le nombre de blocs et de threads par bloc sont des puissances de 2.
— La taille du tableau est égale au nombre de threads.

Q.15: Utiliser le même kernel pour terminer la réduction (étape 2).

Q.16: Généraliser la réduction à une taille quelconque de tableau.

V Calculs d'indice global

Le calcul de l'indice global d'un thread est généralement très important dans un kernel CUDA. Nous allons augmenté le nombre de dimensions des grilles et des blocs dans le programme précédent pour s'habituer à manipuler plus de dimensions dans notre calcul d'indice global

Q.17: Passez à des tailles de grille et de bloc à deux dimensions. Donnez le calcul de l'indice global du thread avec ces 2x2 dimensions.