

M2 - Architecture et Programmation d'accélérateurs Matériels.

(APM 2016-2017)

TP2 CUDA



hugo.taboada.ocre@cea.fr
julien.jaeger@cea.fr
patrick.carribault@cea.fr

Les objectifs de ce TP sont :

- Ecriture d'un premier code CUDA
- Utilisation efficace d'un GPU
- Calcul de l'indice global avec plusieurs dimensions

I Réduction somme en CUDA

Une réduction somme consiste à additionner toutes les valeurs d'un 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.1: 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.

Correction

Voir fichier *reduce.cu*.

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

Correction

Il suffit d'appeler à la suite du précédent kernel le même kernel avec cette fois-ci un nombre de thread égal au nombre de blocs du kernel précédent, et avec un seul bloc. De la même façon, le tableau `db_1` qui recevait les résultats partiels est cette fois-ci en lecture, et c'est la variable `d_sum` qui recevra le résultat final de la réduction.

Voir fichier *reduce.cu*.

Q.3: Généraliser la réduction à une taille quelconque de tableau.

Correction

Il suffit de passer le nombre total de threads en paramètre du kernel, et de vérifier dans le kernel que les éléments des sommes partielles sont valides (indice < ntot).

Voir fichier *reduce_anysize.cu*

II Modification d'image : flou et flou itératif

Le programme *tp3_video1.c* présent dans le répertoire *VIDEO* lit une vidéo, la traduit sous forme de tableaux de pixels, puis modifie ces pixels afin de transformer la vidéo couleur en vidéo en niveaux de gris. Vous pouvez compiler et exécuter le programme, en utilisant la petite vidéo test fournie *Wildlife.wmv*, pour voir l'effet du programme sur la vidéo générée *my_copy.wmv*.

Comme pour le TP précédent, dans le même répertoire se trouve le fichier *tp3_video1.cu*.

Q.4: Dans ce TP, nous allons implémenter une nouvelle modification d'image : le flou. Le flou est une opération de stencil : pour mettre à jour la valeur de la case, nous avons besoin des données des cases voisines. Pour effectuer un flou, il suffit de mettre à jour un pixel avec la moyenne des valeurs de ce pixel et des pixels voisins, pour chaque composante. Dans notre cas, nous allons uniquement considérer les voisin directs. Pour un pixel `[i][j]`, il s'agit des pixels `[i-1][j]`, `[i+1][j]`, `[i][j-1]` et `[i][j+1]`.

Implémenter le kernel permettant de faire un flou sur les frames 200 à 400.

Q.5: Ce code est-il efficace ? Pourquoi ? Rendez-le plus efficace.

Correction

Non, car nous réalisons des transferts pour chaque frame. Il serait plus intéressant de récupérer et transférer les frames toutes ensemble, ou au moins pas paquets de frame si la mémoire ne permet pas de toutes les stockées en même temps.

Q.6: Le programme *tp2_video2.c* réalise une autre modification en plus du niveaux de gris. La méthode de *SOBEL* fournie permet de détecter les contours sur une image quelconque. Ces contours sont alors affichés en blanc sur fond noir. Comme la question précédente, écrivez le code CUDA permettant de réaliser la méthode de SOBEL sur le GPU.