Programmation sur Processeur Graphique – GPGPU
TD 3 : kernel multidimension

Centrale Nantes
P.-E. Hladik, `pehladik@ec-nantes.fr`

—

Version bêta (29 novembre 2022)

1 Ensemble de Julia en CUDA

Objectif 1.1

— réaliser un kernel multidimension

(1.1) Travail à faire : Ensemble de Julia

Les ensembles de Julia est un exemple d'ensembles fractals. Considérant deux nombres complexes, c et z_0, l'ensemble de Julia correspondant est la frontière de l'ensemble des valeurs initiales z_0 pour lesquelles la suite

$$z_{n+1} = z_n^2 + c$$

est bornée.

Nous allons représenter cet ensemble par une image où chaque pixel sera considéré comme un nombre complexe z_0 dans un plan où 0 est le centre de l'image. Si la suite (z_n) est bornée pour le pixel z_0 le pixel prend la couleur rouge, sinon il prend la couleur noir.

Un code séquentiel permettant de représenter l'ensemble de Julia est déjà produit (pour $c = -0.8 + 0.156i$. Le but est de passer ce code sur un GPU.

1. Récupérer l'archive TD3.
2. Compilez le code avec `nvcc -o julia julia_bmp.cu`
3. Exécutez le code et observez la belle image produite.
4. Modifier le code pour l'exécuter sur le GPU et observez les performances !

(1.1) Comment faire : Données

Avant toute chose il faut penser à :
— allouer les espaces mémoire de travail sur le *device*,
— faire les copies de l'*host* vers le *device* (si besoin),
— récupérer à la fin les données du *device* vers l'*host*.

(1.2) Comment faire : Nombre de grids et taille

Dans un premier temps, pour ne pas vous embêter avec les test sur les bornes de l'image, prenez un nombre de grids et de thread multiple de 1000*1000 (dimension de l'image).

(1.3) Comment faire : `__device__`

N'oubliez pas que pour appeler une fonction sur le GPU il faut qu'elle soit déclarée comme `__device__`, même les méthodes d'un objet...

2 Multiplication matricielle basique

Objectif 2.1

— implémenter une routine de base de multiplication de matrices denses

(2.1) Travail à faire : Multiplication matricielle

Le code disponible dans le fichier `matrix.cu` fournit deux fonctions :

1. `initMatrix` pour allouer des valeurs à une matrice,
2. `compareMatrix` pour comparer deux matrices.

La fonction `computeMatrixMulCPU` est simplement prototypée mais n'est pas instanciée.

Pour appeler le programme il faut passer comme paramètre les dimension de la matrice A et B, par exemple si l'exécutable est nommé `matrix` :

```
$ ./matrix 10 30 30 40
```

avec 10 le nombre de lignes de A, 30 le nombre de colonnes de A et de lignes de B et 40 le nombre de colonnes de B. Si le nombre de colonnes de A et de lignes de B sont différentes une exception est lancée (le `assert` ligne 19 du code).

Ajoutez les éléments nécessaires dans le code pour :

— écrire le code de la fonction `computeMatrixMulCPU` pour réaliser la multiplication matricielle sur le CPU,
— mesurer le temps d'exécution de `computeMatrixMulCPU`,
— implémenter le calcul sur GPU (allouer la mémoire, initialiser les blocs de threads et les dimensions de la grille du kernel, invoquer le kernel CUDA, copier les résultats du périphérique vers l'hôte, désallouer la mémoire du périphérique),
— mesurer le temps de calcul sur le GPU,
— utilisez la fonction `compareMatrix` pour comparer les résultats produits sur le CPU et sur le GPU et expliquez ce qui vous arrive.

3 Retour sur l'ensemble de Julia (pour ceux qui sont en avance)

Objectif 3.1

— avoir un rendu de l'ensemble de Julia plus beau

(3.1) Travail à faire : Coloriser l'ensemble de Julia

Au lieu d'afficher un point rouge, essayer de trouver une coloration qui prend en compte le nombre d'itération nécessaire pour tester si le point est dans l'ensemble de Julia ou non.