CHPS0801

-

Modèles de programmation parallèle

Kokkos

Table des matières

I. Introduction

Au cours de ce semestre au Master CHPS de l'université de Reims Champagne-Ardenne, nous avons été introduits à certains modèles de programmations parallèle, dont *Kokkos* (une bibliothèque C++) et la programmation par tâche avec *OpenMP*.

Nous allons ici nous intéresser plus particulièrement à *Kokkos* avec lequel nous tenteront d'implémenter différents algorithmes de traitement d'image, plus précisément de lissage d'images. Les deux algorithmes implémentés seront les algorithmes de **Jacobi** et de **Gauss-Seidel**.

Ces deux algorithmes consistent à lisser une image en moyennant chaque pixel avec ses voisins. La différence se situe dans le fait de prendre comme valeurs des pixels l'itération courante k ou l'itération précédente $k-1$.

II. Lissage

1. Principe

Le lissage d'un pixel est basé toujours sur la même méthode : la notion de voisin d'un pixel. Un pixel est voisin d'un autre s'il est adjacent, c'est-à-dire s'ils partagent une arête commune. Cette adjacence est aussi appelée la 4-adjacence.

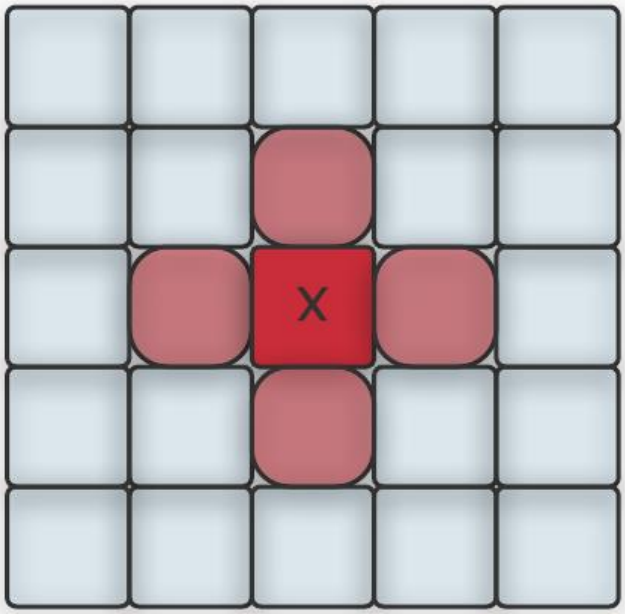

Illustration : Pixel x, en rouge foncé, ainsi que ses voisins (4-adjacence), en rouge clair.

2. Algorithme

Algorithme **Lissage pixel** (image, i, j) :

$x \leftarrow$ image [i, j]

n $\leftarrow$ 1

Pour chaque voisin *p* :

$x \leftarrow x$ + image [p]

n $\leftarrow$ n + 1

$x \leftarrow x \div n$

returner x

3. Gestion des bords

Comme de nombreux algorithmes de traitement d'image et en particulier de lissage, une stratégie doit être appliquée pour la gestion des bords.

En effet, on effectue la moyenne des voisins mais tous les pixels n'ont pas le même nombre de voisins. Ceci est vrai sur les premières et dernières colonnes et lignes, qui ont moins de 4 voisins car ceux-ci sont en dehors de l'image.

Dans l'algorithme donnée, il a été choisi de faire la moyenne en prenant compte des pixels manquant. C'est la méthode qui donne les meilleurs résultats mais aussi celle qui est la plus coûteuse en ressource, car on doit vérifier à chaque fois si les voisins existent. Cependant, il est possible d'optimiser cette gestion en traitant au cas par cas les lignes et colonnes du bord.

D'autres stratégies auraient été possibles, comme ne pas effectuer le lissage sur les bords, ou étendre l'image virtuellement en ajoutant des pixels de même couleur que le bord pour que tous les pixels de l'image aient 4 voisins. Il est possible que cette stratégie n'ait pas un impact flagrant sur les performances. Les autres stratégies peuvent aussi obtenir un résultat visuellement différent, mais éloigné de ce que l'on aurait souhaité obtenir, avec un rendu des bords final plus sombres.

III. Jacobi

1. Algorithme séquentiel

L'algorithme de Jacobi est le plus simple des deux algorithmes. Il consiste à toujours utiliser les pixels de l'image précédente pour former la nouvelle image. Chaque pixel est moyenné avec ses voisins.

```
Algorithme Jacobi séquentiel ( image, k ) :
        image_avant ← Copie de image

        Pour k' allant de 1 à k :

                Pour i allant de 0 à ( image.hauteur - 1 ) :
                        Pour j allant de 0 à ( image.largeur - 1 ) :
                                image [ i, j ] ← Lissage-Pixel ( image_avant [ i, j ] )

                Inverser ( image, image_avant )

        retourner image_avant
```

2. Algorithme parallèle

Par sa nature, la version parallèle de l'algorithme ne pose pas de problème d'ordonnancement pour le parcours des pixels qui peut être effectué parallèlement sans aucun problème. Cependant, comme on a besoin de tous les voisins d'un pixel de l'étape précédente pour calculer sa nouvelle valeur, on ne peut donc pas paralléliser les étapes k. Elles doivent être effectuées séquentiellement, sans que trop d'autres optimisations ne soient possibles.

```
Algorithme Jacobi parallèle ( image, k ) :
    image_avant ← Copie de image

    Pour k' allant de 1 à k :

        Pour parallèle i allant de 0 à ( image.hauteur - 1 ) :
            Pour parallèle j allant de 0 à ( image.largeur - 1 ) :
                image [ i, j ] ← Lissage-Pixel ( image_avant [ i, j ] )

        Inverser ( image, image_avant )

    returner image_avant
```

IV. Gauss-Seidel

1. Introduction

2. Diagonales

3. Algorithme

4. Algorithmes pour les diagonales

5. Optimisation par fenêtre

V. Conclusion

1. Difficultés

Plusieurs difficultés ont été rencontrées. Tout d'abord, la bibliothèque *OpenCV* n'était pas, en l'état, compatible avec CUDA et les classes *cv::Mat* et *cv::Vec3* étaient donc inutilisables, ou alors avec des avertissements. Nous avons donc dû développer une mini-bibliothèque qui soit compatible avec, c'est-à-dire qui rajoute les macros *__host__* ou *__device__* nécessaires. Cette classe gérait tous les accès aux pixels de façon transparentes, que se soit avec les tampons mémoire de Kokkos ou bien les matrices d'OpenCV.

Il a été possible d'utiliser les buffers mémoires non-managés pour permettre un transfert plus rapide de la mémoire hôte vers la mémoire du *device*. Les alignements choisis ont été *LayoutLeft* car ils correspondent à un alignement standard de la mémoire en C et donc la convention intuitive *[ligne, colonne]* pour les indices a pu être gardée.

Pour Gauss-Seidel, il a été difficile d'estimer le nombre d'éléments à actualiser à chaque fois pour la méthode par fenêtre. Cependant, il suffit de majorer le nombre d'éléments, car tant que l'on ne dépasse pas les capacités du nombre de threads CUDA (de l'ordre de 1024 sur ROMEO), la vitesse d'exécution ne devrait pas beaucoup changer. On peut même s'attendre à une optimisation par le fait que le nombre de threads lancés soit toujours le même à chaque étape.

L'un des principaux intérêts de l'algorithme de Gauss-Seidel est qu'il permet un traitement *en-place*, c'est-à-dire sans avoir besoin d'utiliser une image auxiliaire, coûteuse en mémoire.

2. Protocole d'expérimentation

La version qui a été considérée la mieux « optimisée » ne l'est pas forcément. En fait, cela dépend du contexte d'exécution. Bien sûr, les méthodes considérées comme mieux optimisées fonctionnaient mieux sur GPU, l'objectif initial, mais pouvaient parfois ralentir l'exécution sur CPU, car il y a plus de threads lancés qui nécessaires, et ceux-ci n'exécutent aucun code. Cela ne pose pas de problème sur GPU car de toute façon tous les threads doivent s'exécuter en parallèle, mais cela ralentit l'exécution sur CPU, car ceux-ci sont exécutés séquentiellement par le nombre faible de cœurs comparés au nombre de threads. Les tests ont été effectués pour k=10, en local, avec une GTX 1650 Ti et une machine dotée de 6 cœurs physiques, et sur ROMEO. L'image de test (687x888px) provient de la base de données en ligne *unsplash.com*.

Figure 1 : Image originale.

Figure 2 : Image lissée pour k=10

Il y a peu, voir aucune différence entre les images obtenues selon la version de chaque algorithme.

3. Résultats obtenus en local

Méthode	**OpenMP**	**CUDA**	**Séquentiel**
Jacobi	0.919s	**0.00997s**	4,516s
Gauss-Seidel par étapes	1.096s	0.16s	4,615s
Gauss-Seidel par glissement de fenêtre	1.55s	**0.0167s**	X

4. Résultats obtenus sur ROMEO

Méthode	**OpenMP**	**CUDA**	**Séquentiel**
Jacobi	0.019s	**0.0059s**	1.005s
Gauss-Seidel par étapes	0.519s	0.134s	1.022s
Gauss-Seidel par glissement de fenêtre	0.0728s	**0.0094s**	X