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.

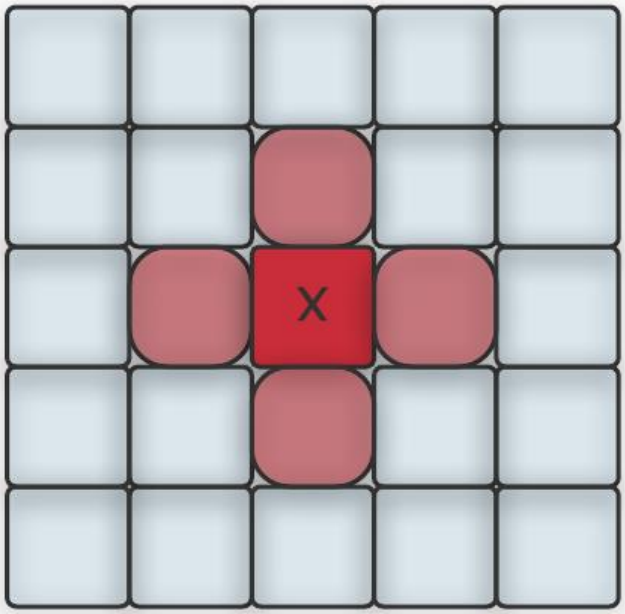


Illustration : Pixel x, en rouge foncé, ainsi que ses voisins (4-adjacence), en rouge clair.

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

L'algorithme de Gauss-Seidel diffère de l'algorithme de Jacobi par le fait qu'il utilise à la fois des pixels de l'itération courante, et à la fois des pixels de l'itération précédente. Il s'agit de la notion d'amont-aval. Les voisins d'une direction (Nord-Est, Nord-Ouest, Sud-Est, ou Sud-Ouest) proviennent de l'itération précédente tandis que les autres pixels proviennent de l'itération courante. C'est-à-dire qu'une partie des pixels doivent avoir déjà été calculés. Ceci est possible pour les mêmes raisons que la gestion des bords : le pixel tout en haut à gauche de l'image n'a pas de voisins Nord-Ouest par exemple, donc il sera possible d'initialiser la descente et continuer ainsi de proche en proche. On voit qu'il sera nettement plus difficile de paralléliser la méthode, du fait de sa définition récursive, avec une propagation de proche en proche, comme un flot d'eau. Finalement, le pixel au centre que l'on doit modifier, est naturellement défini par l'itération précédente car c'est lui que l'on cherche à calculer. Enfin, cette version n'a pas besoin de stockage auxiliaire (modification *en place*), car les pixels écrasés ne sont pas réutilisés.

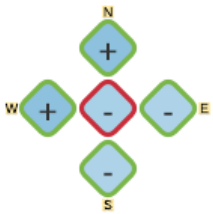
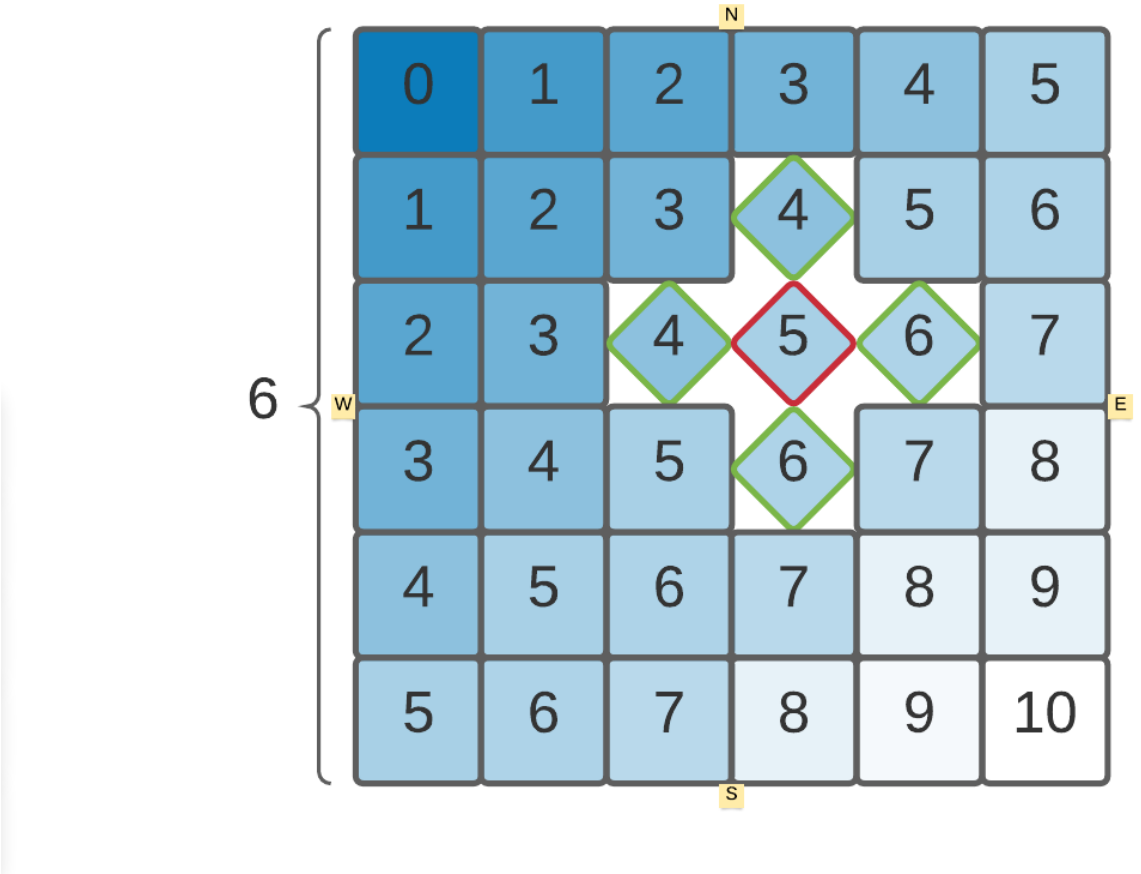


Illustration : Les "+" correspondent aux nouvelles valeurs, les "-" aux anciennes.

2. Diagonales



Ordre de parcours des pixels : Les chiffres les plus élevés sont parcourus en dernier. Les pixels aux mêmes chiffres peuvent être parcouru parallèlement On peut considérer que chaque ensemble de pixels d'un même chiffre est une tâche.

Afin d'implémenter l'algorithme de Gauss-Seidel, on doit d'abord introduire des notions en plus que l'on n'a pas eu besoin pour Jacobi. Nous avons choisi ici d'effectuer le traitement avec les nouvelles valeurs prises depuis le Nord-Ouest, la propagation du flot s'effectuant donc vers le Sud-Est ↘.

Les diagonales apparaissent ici, car on doit itérer chaque diagonale de la matrice dans leur ordre d'apparition, comme si l'image était tournée de 45° vers la gauche. On notera que ce qu'o appelle ici une "diagonale", est en fait une antidiagonale au sens mathématique. Les véritables diagonales mathématiques sont tous les éléments (i, i) et ici on considérera les éléments $(hauteur - i, i)$ comme une diagonale, car se sont elles qui apparaissent avec cette direction amont vers aval ↘ .

3. Algorithme

Cette version est pour une seule itération, c'est-à-dire $k = 1$. On notera que dans tous les cas, pour toutes les versions de ces algorithmes, il suffit de répéter l'opération pour obtenir un plus grand lissage, même si des optimisations sont possibles en connaissant à l'avance le nombre k d'itérations souhaitées, comme nous le verrons par la suite.

Algorithme **Gauss-Seidel parallèle** (image) :
    Pour chaque diagonale d de l'image :
        Pour parallèle chaque pixel p de d :
            image [p] = **Lissage pixel** (image, p)

4. Algorithmes pour les diagonales

Pour exécuter de façon optimale cet algorithme, nous avons besoin de savoir différentes informations sur les diagonales des images pour ne pas lancer plus de threads que nécessaire. Pour cette version en tout cas, car déjà il est possible qu'un nombre constant de threads soit plus optimisé, et pour une autre version plus optimisée, nous verrons par la suite que cela n'est de plus pas toujours possible. On cherchera aussi à convertir ce qu'on appelle des *coordonnées diagonales*(id, dj) vers des coordonnées matricielles standards ligne / colonne (i, j).

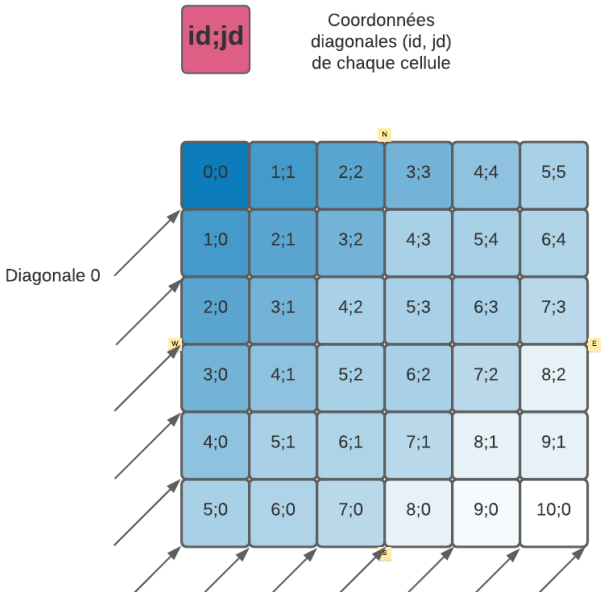


Illustration des coordonnées diagonales

Nous nous poserons donc différentes questions. Tout d'abord, combien de diagonales a une matrice ? Nous étudierons pour différentes tailles de matrices afin d'en visualiser le comportement.

Taille	Diagonales
1	1
2	3
3	5
4	7

Un motif se dessine, une matrice de taille n possède $2n - 1$ diagonales. Pour une matrice non-carrée un schéma nous montre qu'elle possède $largeur + hauteur - 1$ diagonales.

Maintenant, combien d'éléments possède la d-ième diagonale ?

Taille	Diagonale						
	0	**1**	**2**	**3**	**4**	**5**	**6**
1	1 élément						
2	1	2	1				
3	1	2	3	2	1		
4	1	2	3	4	3	2	1

On peut déduire un algorithme à partir de ce motif :

Algorithme **Nombre d'éléments sur la diagonale** (taille, d) :
    nb. diags ← **Nombre de diagonales** (taille)
    Si d < taille :
        retourner d + 1
    Sinon :
        retourner nb. diags - d

Maintenant, pour savoir comment convertir des coordonnées diagonales (di, dj) en coordonnées matricielles (i, j), on pourra agir de la façon suivante. Il est simple de connaître les coordonnées du premier et du dernier élément d'une diagonale quelconque. Comme l'élément suivant sur une diagonale est toujours au Nord-Est de l'élément courant, il suffira de calculer la différence de lignes (ou de colonnes) entre le premier élément d'une diagonale et le dernier élément (plus un) pour en connaître le nombre total d'éléments.

```
Algorithme Coordonnées de la première cellule de la diagonale ( taille, d ) :
        last_row ← taille - 1
        Si d < taille :
                retourner ( d, 0 )
        Sinon :
                retourner ( last_row, d - taille + 1 )
```

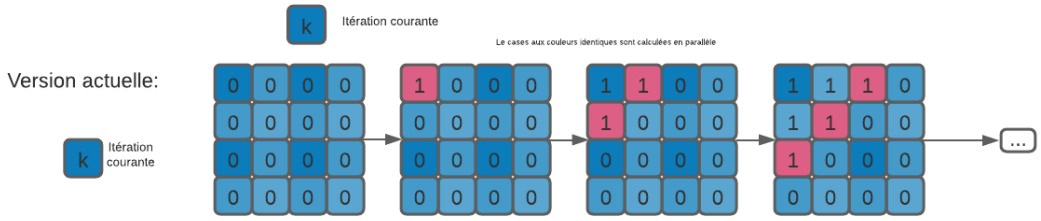
```
Algorithme Coordonnées de la dernière cellule de la diagonale ( taille, d ) :
        last_col ← taille - 1
        Si d < taille :
                retourner ( 0, d )
        Sinon :
                retourner ( d - taille + 1, last_col )
```

Coordonnées d'une cellule quelconque d'une diagonale en connaissant les coordonnées de la première cellule de la diagonale : comme la cellule suivante est situé en haut à droite, il suffit de faire -1 à la ligne et +1 à la colonne.

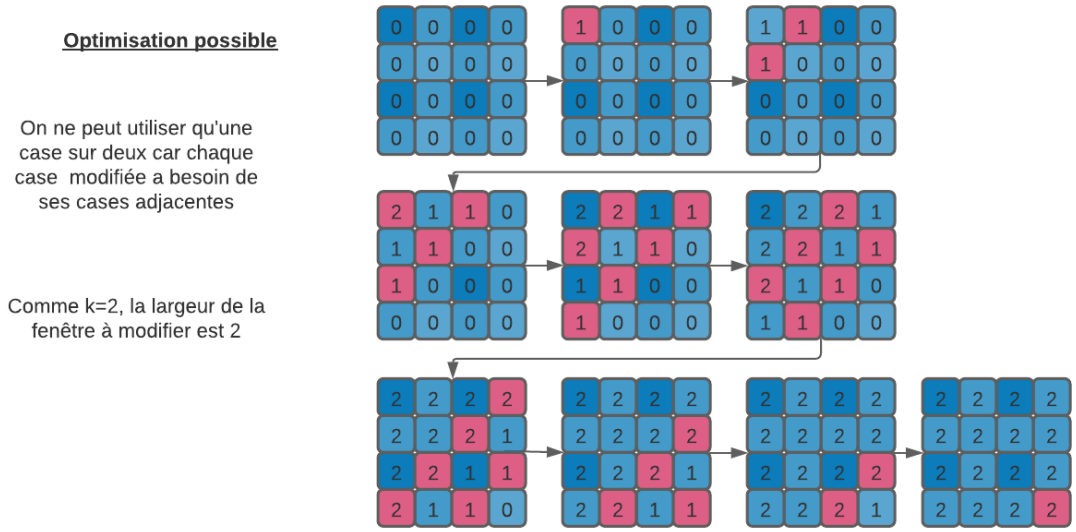
```
Algorithme Coordonnées cellule quelconque de la diagonale ( taille, d, id, jd ) :
        i0, j0 ← Coordonnées première cellule ( taille, d )
        retourner ( i0 − id, j0 + jd )
```

5. Optimisation par fenêtre

La version présentée traite chaque étape k itérativement. Est-il possible d'optimiser ce traitement ? Oui, car on remarque que plusieurs pixels peuvent être actualisés en même temps. Plus précisément, un pixel sur deux dans le quadrillage. Cela peut être géré par un même thread sans cas particulier car le traitement est identique quelle que soit l'étape. Que ce soit pour passer de l'étape k à l'étape $k+1$, ou pour passer de l'étape $k+1$ à l'étape $k+2$, on applique toujours le même algorithme **Lissage Pixel**, sans que l'algorithme ai besoin de connaître k.



On voit sur cette illustration l'exécution de la première version. On voit ici le séquence d'exécution pour la première étape du lissage $k = 1$. Peu de cœurs sont exécutés en parallèle, avec au maximum le nombre d'éléments maximum sur une diagonale, c'est-à-dire le nombre d'éléments sur la diagonale de la matrice.



On appellera cette nouvelle version le **glissement de fenêtre**, car on glisse une fenêtre de traitement de taille constante qui utilisera un pixel sur deux. La fenêtre réactualisera plusieurs fois le même pixel, exactement k fois, c'est-à-dire exactement le nombre souhaité d'applications du traitement de lissage. Cette version est plus optimisée car beaucoup plus de threads sont lancés en parallèle. Or, pour une implémentation sur GPU, on cherchera la plupart du temps à maximiser le nombre de kernels, ce qui est fait ici.

On a maintenant un algorithme afin de convertir des coordonnées diagonales aux coordonnées matricielles, on peut calculer chaque étape de l'algorithme de Gauss-Seidel. Cet algorithme possède 3 phases pour une gestion par glissement de fenêtre :

1. La taille de la fenêtre augmente
2. La taille de la fenêtre se décale et reste de taille constante
3. La taille de la fenêtre diminue

Les phases 1 et 3 sont les mêmes que la phases 2 en ignorant les cellules en dehors du tableau. Ici, on généralisera on ne s'occupera pas des cas particuliers.

Algorithme **Gauss-Seidel parallèle par fenêtre** (image, k) :

$tf \leftarrow 2 * k - 1$ // tf : Taille de la fenêtre

// f : Diagonale de départ de la fenêtre. Négatif pour la phase 1.

Pour f allant de $(-tf + 1)$ à $(nb\_diags - 1)$:

Pour parallèle d allant de f à $(f + tf - 1)$ par pas de 2 :

Si ($d \geq 0$ et $d <$ nb_diags) :

Actualiser la diagonale d en parallèle

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