TP Convolution 2D

Traitement d'image avec *pthread*

Programmation concurrente - Remédiation

Etienne Guignard

Année 2016-2017

24/08/2016

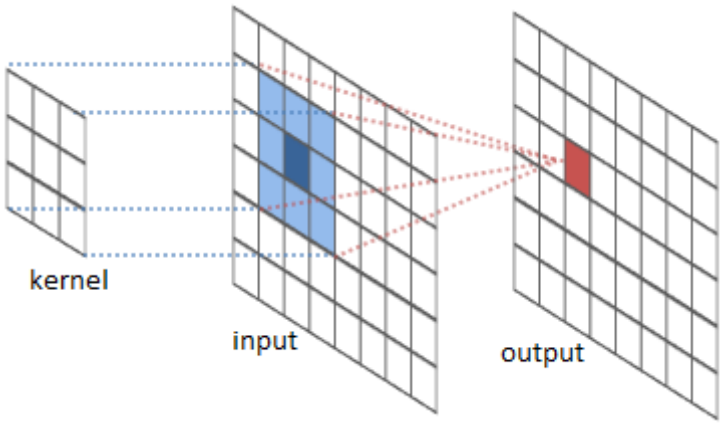


Table des matières

Introduction

Dans ce rapport, nous allons nous intéresser au traitement d'image de type convolution 2D, ce qui nous permettra de modifier l'aspect visuel d'une image en fonction de paramètres (filtres) prédéfinis.

Objectifs

Dans un premier temps, nous allons étudier le principe de fonctionnement de la convolution 2D et mettre en place des fonctions qui permettent d'alléger le traitement de ce type d'opération en utilisant le modèle *boss/worker*. Ensuite, nous allons comparer chaque résultat de convolution (temps d'exécution) avec une version séquentielle et multi threads afin de constater s'il y a une amélioration du temps de traitement.

Description

Le programme est structuré en quatre fichiers :

- main
- kernel
- ppm
- queue

Le fichier principal *main* contient les informations suivantes : la demande à l'utilisateur d'informations concernant le nombre de thread qui va définir un des coefficients et ainsi, définir en combien de parties l'image sera séparée ; le filtre que à appliquer sur notre image ; et enfin, l'image à partir de laquelle on va travailler. À la suite de quoi, on appellera les différentes fonctions qui sont implémentées dans les autres fichiers liés à cette dernière. D'autre part, le traitement complet de convolution a été ajouté dans le *main* afin de faciliter d'implémentation. Finalement, la partie la plus intéressante est celle qui a permis de réduire le temps de traitement de l'image et la combinaison du code séquentiel avec des threads.

Le fichier *ppm* fournit par le professeur contient les fonctions qui permettent de lire l'image donnée en paramètres et de la charger en mémoire dans une structure de données définie afin de pouvoir effectuer un traitement dessus. L'une de ces fonctions permet aussi de sauver l'image en format *ppm* pour reconstituer l'image sur laquelle la convolution a été appliqué.

Le fichier *kernel* comporte la fonction qui va permettre d'importer les informations relatives au kernel (filtre). Ces dernières pourront ainsi être utilisées pour faire le traitement de convolution sur l'image. Les filtres ont été mis dans un fichier *<nom_du_filtre.k>* pour faciliter leurs utilisation et éviter de les mettre en dur dans le code, ce qui n'aurait pas été très judicieux. De manière similaire, la structure du fichier dans laquelle était stockée l'image a été reprise afin de définir sa taille évitant à l'utilisateur de devoir entrer non seulement le fichier mais aussi la taille du filtre.

Le fichier *queue* comporte les fonctions utilisées pour crée une queue, empiler et dépiler des éléments qui seront destinés à être traiter par la suite par les worker et le boss. Il contient

aussi une structure de données permettant de stocker les variables relatives à la queue et à la convolution.

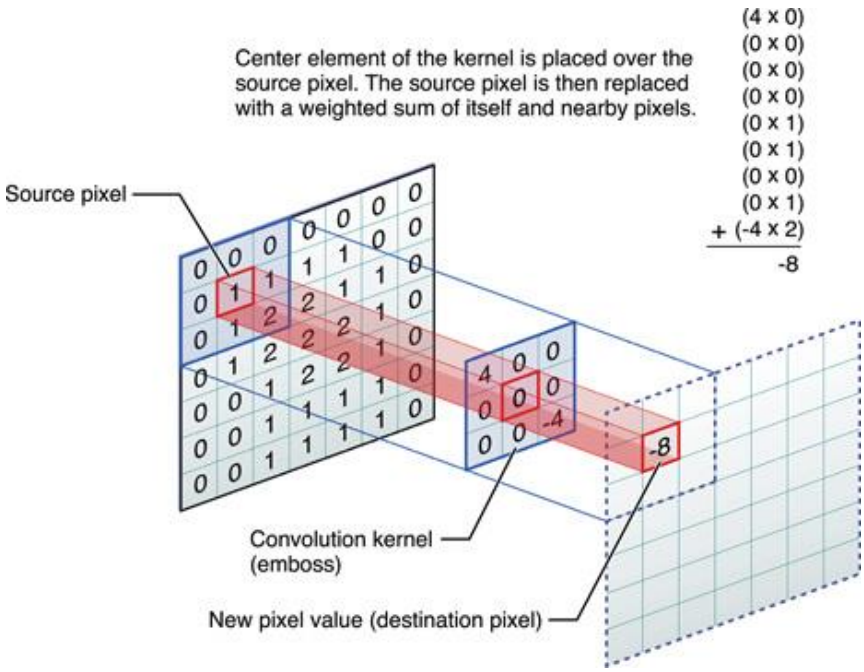
Convolution 2D

Principe de fonctionnement

Pour effectuer la convolution en appliquant le kernel sur notre image, on s'est inspiré de la fonction suivante :

$$C(x,y)=\sum_{j=\frac{-N}{2}}^{\frac{N}{2}}\sum_{i=\frac{-N}{2}}^{\frac{N}{2}} K\left(i+\frac{N}{2},j+\frac{N}{2}\right)I(x+i,y+j)$$

Voici de manière plus concrète comment la fonction précitée a été utilisée pour traiter l'image.



Démarche employée (convolution)

Parcours de l'image

Pour parcourir l'image, on est resté sur l'idée d'un tableau à une dimension, car cela permet d'effectuer le parcours en une seul boucle *for* (de l'indice 0 jusqu'à la taille de l'image -1). Avec cette méthode, il est donc simple de diviser le nombre de pixels de l'image en changeant l'indice de départ et celui de fin. Le désavantage de cette technique, est que la gestion des bords de l'image s'avère plus compliquée que si elle était convertie en un tableau à deux dimensions.

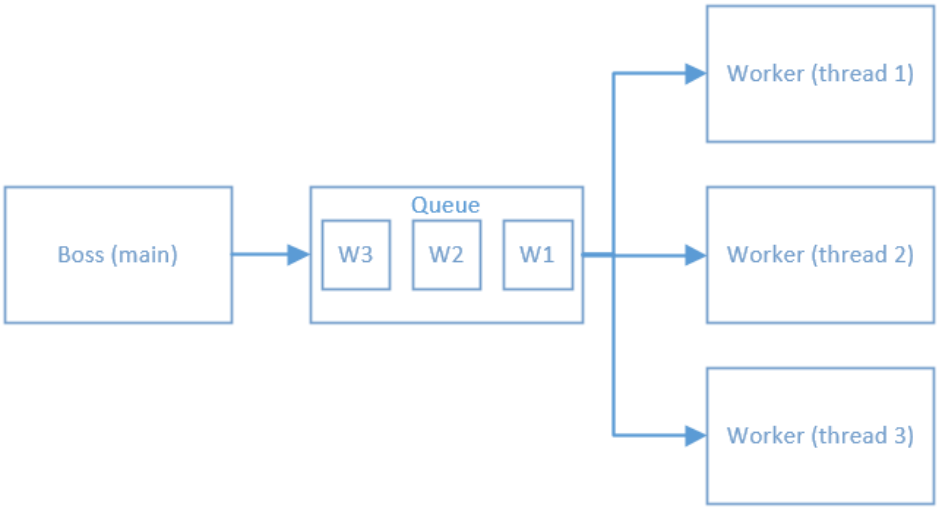
Délimitation des pixels

La structure composée de pixels et chargée à partir de l'image a permis de décomposer chaque pixel en 3 sous-pixels *r,g,b*. Chaque sous-pixel contient une valeur entre 0 et 255. Quand un kernel est appliqué, il est possible que la valeur de chaque sous-pixel multipliée puis additionnée avec le kernel dépasse les 255. Par conséquent, on a été contraint de borner cette valeur à 255, car le format *ppm* ne supporte pas de valeur supérieure.

Démarche employée (thread)

Model boss/worker

Comme indiqué précédemment, afin d'améliorer les performances de ce traitement de convolution qui pourrait s'avérer long avec un programme séquentiel, il nous a été demandé d'utiliser le modèle *boss/worker* afin de diviser ce travail en différents threads. De prime abord, bien que l'on puisse accéder à une explication de ce modèle sur Cyberlearn, cette dernière n'était pas suffisamment claire. C'est pourquoi, comme les consignes étaient très générales, on a décidé de le faire comme on avait compris tout en respectant le modèle.

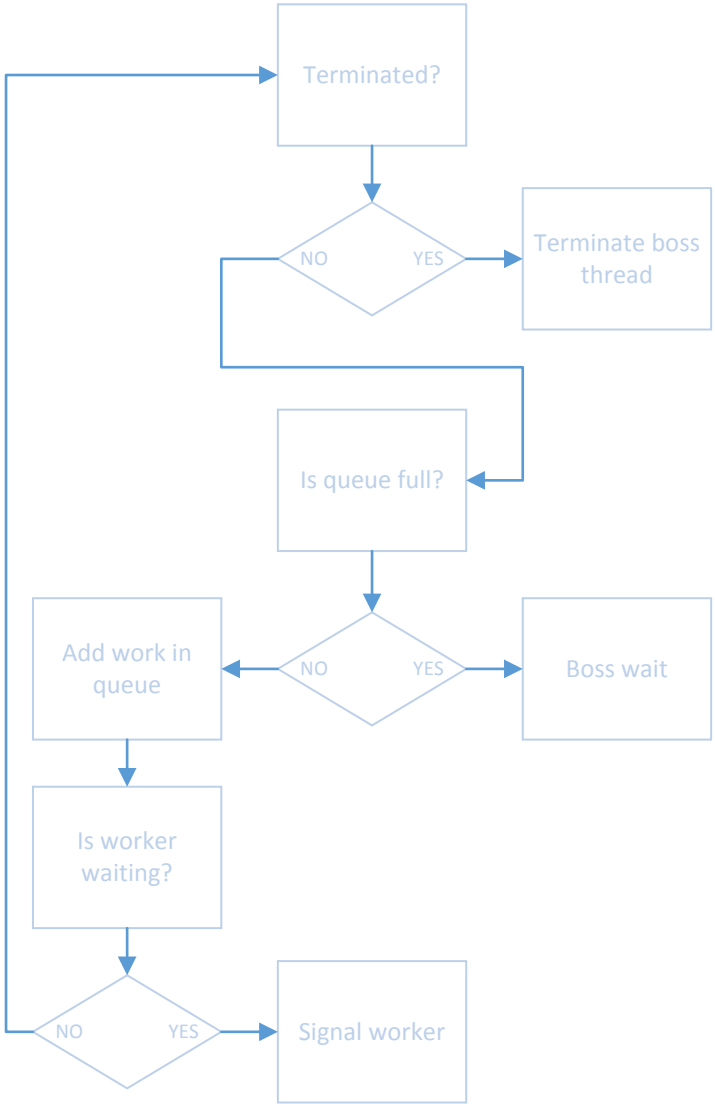


On est parti du principe que le boss donnait des instructions au worker et ceux-ci effectuaient les tâches demandées. Etant donné que le travail donné par le boss n'est pas constant et que tous les worker peuvent effectuer des tâches, il est impératif de pouvoir stocké les instructions

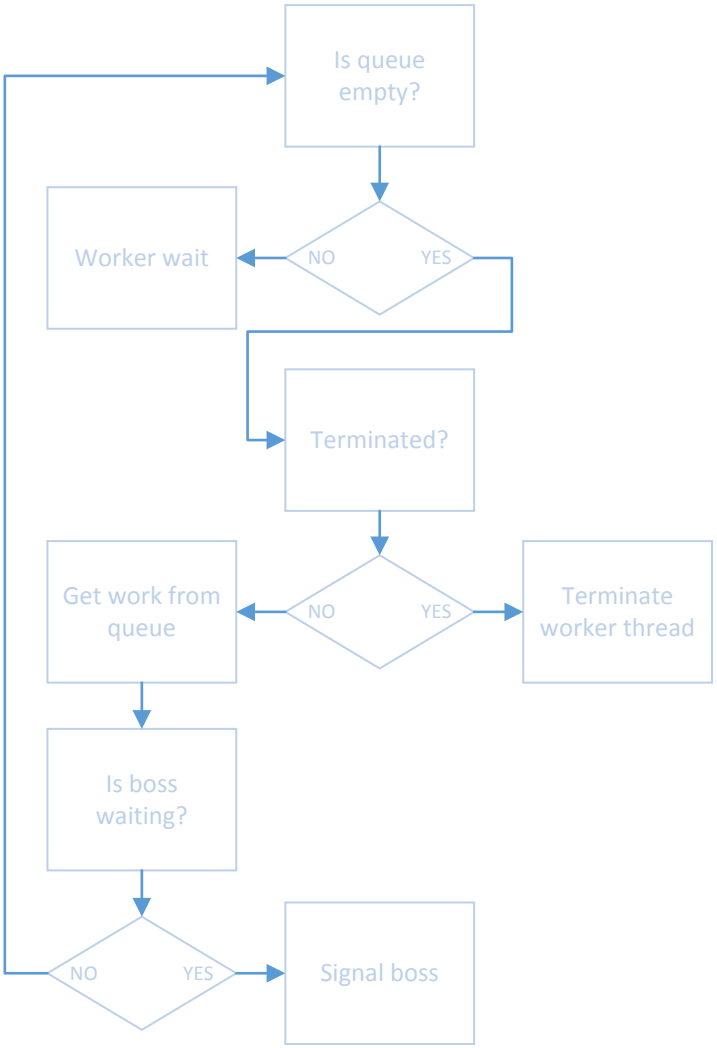
du boss dans une queue en attendant qu'un worker soit libre. Quand le boss n'a plus d'instructions à donner (dans ce cas, la convolution de l'image est finie), le boss indique au worker que le travail est terminé. A la suite de quoi, l'exécution des threads worker et boss se termine. Le thread boss est représenté par la fonction main qui se terminera que dans la condition citée précédemment.

Bien sûr, il a été important de synchroniser le tout pour éviter des débordements. Par exemple, quand le boss ajoute du travail dans une queue pleine ou qu'un worker effectue une tâche alors qu'il ne dispose d'aucune instruction dans la queue. Afin de mieux expliquer la démarche employée pour le mécanisme de synchronisation entre le boss et les worker, on a décidé de faire deux organigrammes qui correspondent respectivement au boss et au worker.

Organigramme boss



Organigramme worker



Structure de données

Pour clarifier le programme, on a décidé de mettre toutes les variables qui seraient utilisées par la fonction de convolution ainsi que les primitives de synchronisation dans une structure globale qui se trouve dans les fichiers *queue*.

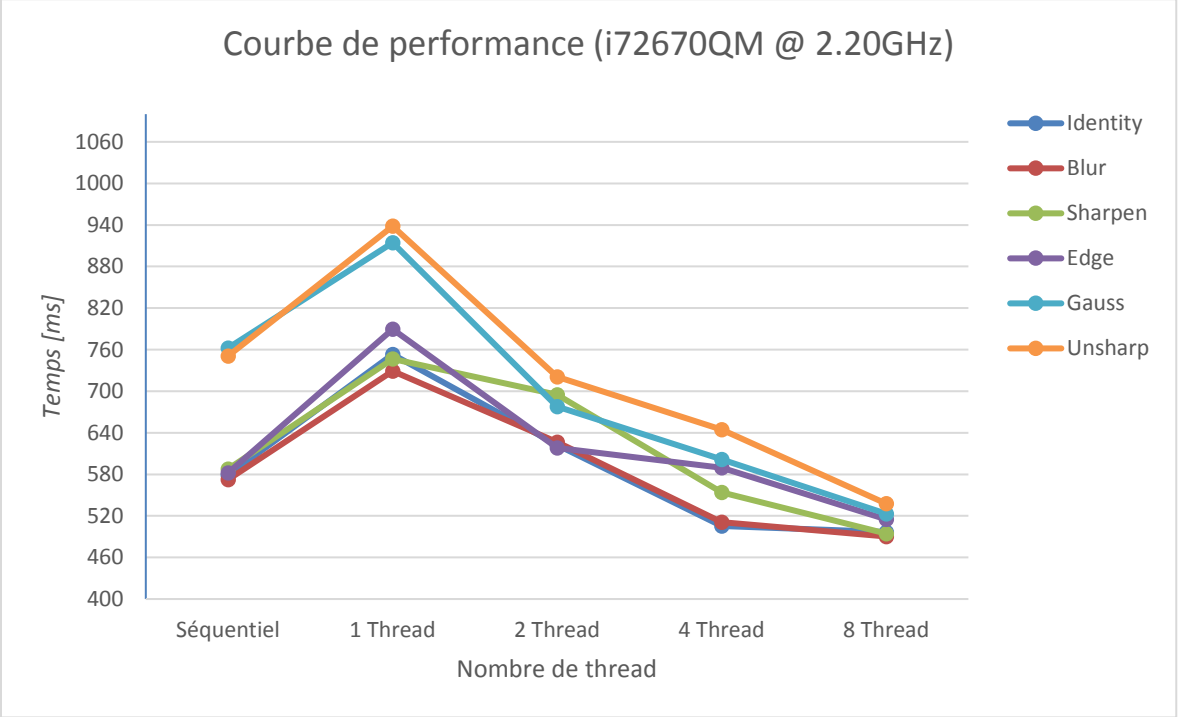
Division de l'image

Il est important de prendre en compte la taille de la queue et le nombre de threads entrés par l'utilisateur, car ces deux paramètres définissent en combien de morceaux l'image sera traitée. Par conséquent, on a arbitrairement décidé d'attribuer à la queue un maximum de 8 éléments, car on n'a pas trouvé de méthode appropriée.

Calcul des performances

Les tests de performance ont été effectué avec l'image fournie par un ordinateur doté d'un processeur quad core (i72670QM @2.2GHz). Chaque filtre a été appliqué trois fois sur l'image afin d'obtenir les résultats cohérents suivants en milliseconde :

	Identity	Blur	Sharpen	Edge	Gauss	Unsharp
Séquentiel	578.79	572.22	587.53	582.22	762.54	750.61
1 Thread	753.12	729.11	746.53	789.53	914.38	938.22
2 Thread	622.44	626.51	694.83	618.18	677.58	720.83
4 Thread	505.51	511.17	553.71	589.43	601.58	644.35
8 Thread	496.77	490.32	494.51	515.03	523.17	537.77



Conclusion

Suite aux résultats obtenus, on observe 2 phénomènes :

- Le temps d'exécution de la version séquentielle est inférieur à celle avec un thread. Autrement dit, le mécanisme qui a été mis en place ralentit la convolution pour un seul thread.
- Sur la machine utilisée, on peut observer que le gain de temps avec et sans thread semble censé. D'autre part, le temps gagné par l'exécution en multithreads semble linéaire puis diminue en se rapprochant du nombre de cœur de la machine.

Références

- http://maxim.int.ru/bookshelf/PthreadsProgram/htm/r_19.html
- http://www.informit.com/articles/article.aspx?p=169479&seqNum=11
- http://www.mcs.csueastbay.edu/~tebo/Classes/6560/threads/design.html
- http://web.vtc.edu/users/pcc09070/TutorialPThread/pthread-Tutorial.pdf
- https://github.com/flaviusone/APP