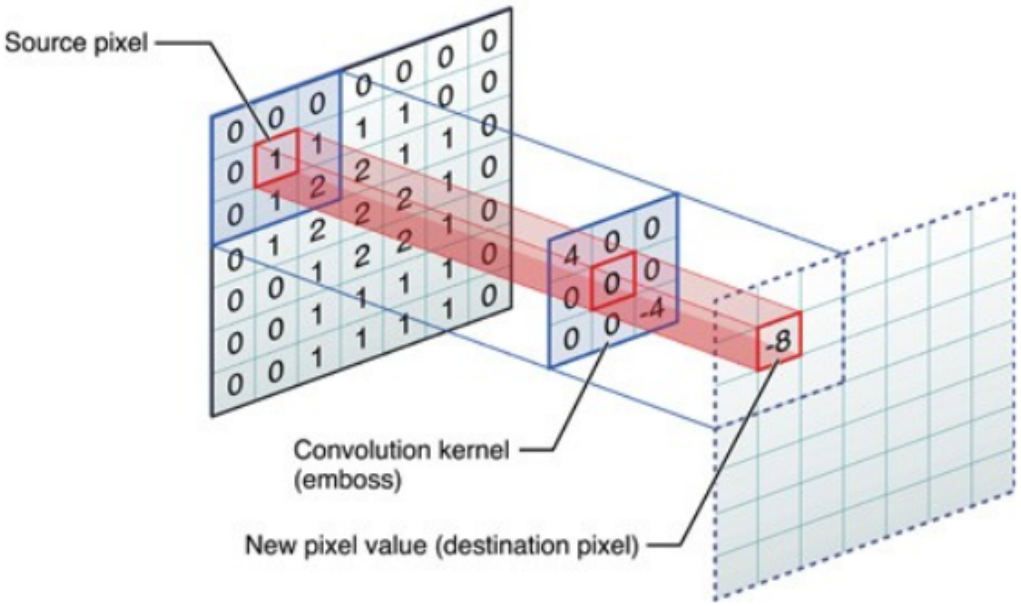
h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

HEPIA - ITI 2 Formation du soir
Programmation concurrente

Convolution 2D avec Pthread

TRAVAIL PRATIQUE 1 - RAPPORT



Enseignant : Florent GLÜCK
Auteur: Christopher REJAS, Fiorenzo POROLI

Genève, le 18 octobre 2015

Table des matières

CHAPITRE 1

Introduction

1.1 Consigne

La convolution 2D discrète est une opération très utilisée en traitement d'image, car elle permet d'appliquer un large éventail de filtres à une image, comme par exemple la détection de contours, les effets de flou, l'accentuation du piqué, etc.

Le désavantage de celle-ci est qu'elle est relativement lourde à calculer. Le but de ce travail pratique est de paralléliser le calcul de la convolution 2D discrète afin d'en accélerer le calcul, ceci grâce à l'utilisation de threads sur les architectures processeurs modernes multi-coeurs.

La convolution 2D discrète C entre le kernel K (filtre) de dimension N et l'image I est définie au point (x,y) de l'image par l'équation suivante :

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

Dans le cadre d'un travail pratique, nous sommes amenés à développer un programme, fonctionnant en ligne de commande, qui appliquera la convolution 2D d'un filtre sur une image de type ppm. Le programme doit permettre à l'utilisateur de spécifier :

- L'image pour laquelle faire la convolution
- Le filtre à appliquer
- Le nombre de threads à utiliser
- L'image de sortie

Le programme sera réalisé en C avec la librairie PThread.

CHAPITRE 2

Kernels

2.1 Structure de données

Pour structurer les données d'un kernel, nous avons créer la structure `kernel_st` qui contient les éléments suivants :

– La taille du kernel
– Le tableau contenant les valeurs

```
1 typedef struct kernel_st{
2   int size;
3   float* matrix;
4 }kernel_t;
```

Listing 2.1 – Structure de kernel_st

Nous avons choisi que le champ `matrix` soit de type pointeur sur float afin que ce soit un tableau dynamique à 1 dimension et ainsi garder le même principe de structure de donnée utilisé dans la librairie `<ppm.h>` pour stocker une image.

De plus, l'allocation mémoire est plus simplifiée et s'effectue sur une ligne. Cela nous évite également d'effectuer une boucle `for` dans le cas d'un tableau à 2 dimensions.

2.2 Chargement d'un mask

Le chargement d'un kernel se fait depuis un fichier portant l'extension `.msk`. Le fichier n'est rien d'autre qu'un simple fichier CSV comprenant la taille du kernel ainsi que ses données.

Cette solution permet à notre code d'être entièrement indépendant de la taille d'un kernel. Cela nous permet également, indépendamment du code, d'ajouter un kernel quelconque en ajoutant simplement le fichier `.msk` au projet.

```
3
0;0;0
0;1;0
0;0;0
```

Listing 2.2 – Fichier identity.msk qui contient le kernel Identity

En pratique lors du démarrage du programme, l'utilisateur entre en paramètre le nom du kernel qu'il souhaite appliquer à son image. Le nom du kernel passé en paramètre sera en réalité utiliser pour charger le fichier `.msk`.

Le programme se chargera alors, grâce au nom du kernel en paramètre, de charger le fichier correspondant et d'appliquer ce dernier à l'image.

2.3 Lecture du kernel

Grâce à la librairie `<kernel.h>` que nous avons développé, nous effectuons la lecture du kernel à l'aide de la fonction `get_kernel_value()` en lui donnant comme paramètre :

– Un pointeur sur une structure `kernel_st`
– La position en X de l'élément du tableau
– La position en Y de l'élément du tableau

Pour respecter l'équation de la convolution discrète, nous faisons la lecture du tableau de $-\frac{N}{2}$ jusqu'à $\frac{N}{2}$. Par exemple, pour une matrice de 3x3, pour obtenir la première valeure du tableau (l'élément à la position 0), nous devons entrer en paramètre $(x : -1, y : -1)$.

	-1	0	1
-1	0	1	2
0	3	4	5
1	6	7	8

FIGURE 2.1 – Représentation du tableau d'un kernel

Pour retrouver la position exacte de l'élément par rapport aux paramètres, nous appliquons la relation suivante :

$$D(x,y) = K((y + \frac{N}{2}) * N + (x + \frac{N}{2}))$$

$$x, y \in [-\frac{N}{2}, \frac{N}{2}]$$

Cette équation permet de retrouver la bonne valeur du tableau en donnant une valeur x et y entre $-\frac{N}{2}$ à $\frac{N}{2}$. Cela va nous être très pratique lorsqu'on appliquera la formule de convolution discrète (chapitre suivant).

CHAPITRE 3

Convolution

3.1 Approche sequentielle

Pour effectuer la convolution de manière séquentielle, nous avons séparé la tâche 2 deux partie : La convolution sur 1 pixel donné ainsi que la convolution appliquée sur l'image entière. La seconde fonctionnalité fait appel à la première pour chaqu'un des pixels de l'image.

Lors du calcule du nouveau pixel, les valeurs de débordement (hors image) valent 0.

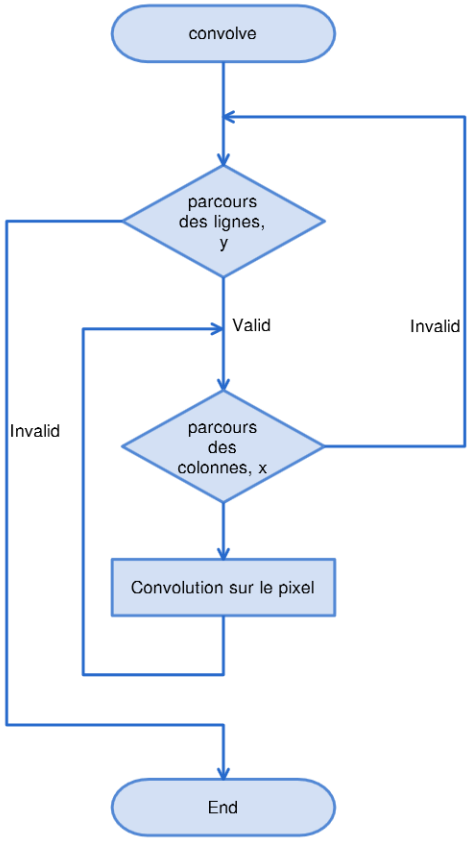


FIGURE 3.1 – Organigramme de la convolution séquentielle

3.2 Approche multi-thread

Si l'on spécifie à la méthode séquentielle une ligne de départ et une ligne d'arrivée, cela nous permettra d'effectuer une convolution partielle sur l'image.

Notre approche multi-threadée est la suivante : L'idée est que chaque thread puisse effectuer une convolution partielle de l'image. Pour cela, nous divisons l'image horizontalement par le nombre de thread demandé en paramètre lors de l'execution du programme.

FIGURE 3.2 – Exemple de division horizontale de l'image avec 4 threads

Pour déterminer les lignes de départ et d'arrivée, nous effectuons la relation suivante :

$$d = \frac{height}{Nthreads}$$

$$y_{start} = id_{thread} * d$$

$$y_{end} = y_{start} + d$$

– d étant la hauteur de l'image divisée, qui s'obtient en divisant la hauteur de l'image source par le nombre de thread désiré

– y_{start} la ligne de départ de l'image à convoluer, qui s'obtient à multiplier d par le numéro du thread courant.

– y_{end} la ligne d'arrivée de l'image à convoluer, qui s'obtient en additionnant y_{start} par d. (offset)

3.3 Cas particulier

Le résultat du calcul de la hauteur de l'image divisée par le nombre de thread retourne une valeure entière. Dans le cas ou la hauteur n'est pas un multiple du nombre de thread, il se peut qu'une partie de l'image ne soit pas prise en compte à cause du résultat de la division entière.

FIGURE 3.3 – Cas particulier illustré dans le cas ou l'image n'est pas un multiple de 6

Pour remédier à cela, nous avons décider que le dernier thread effectue la convolution jusqu'à la fin de l'image, d'où l'expression conditionnelle à la ligne 6.

```
1  void* convolve_thread(void* c) {
2    convolve_param_t* p = (convolve_param_t*) c;
3
4    int bloc = p->c->img_src->height / p->c->n_thread;
5    int row_start = p->current_thread * bloc;
6    int height_size = (p->current_thread != p->c->n_thread-1) ? ↩
       (row_start + bloc) : p->c->img_src->height;
7
8    for(int y = row_start; y < height_size; y++)
9      for(int x = 0; x < p->c->img_src->width; x++)
10       convolve_pixel(p->c->img_src, p->c->img_dst, p->c->k, x, y);
11
12   return NULL;
13 }
```

Listing 3.1 – Fonction appelée par un thread pour effectuer la convolution

CHAPITRE 4

Performances

4.1 Conditions de mesure

A l'aide de la librairie `<time.h>`, nous avons pu mesurer le temps qu'effectue la convolution dans notre programme. Pour cela, nous avons effectué 2 types de mesures :

– Mesure du temps avec un kernel 3x3
– Mesure du temps avec un kernel 5x5

Les mesures suivantes ont été réalisée avec la configuration suivante :

– Intel Core i3, 4 Core
– 8 GB RAM

Pour comparer également les performances par rapport aux taille des images, nous avons également effectué les mesures sur 2 images, dont une petite (1024x711 px) et une autre relativement grande (7359x4531 px).

4.2 Mesures des performances

Nous avons commencé par effectuer une mesure de la convolution sur une image de taille 1024x711px. Nous avons effectués des mesures pour 1 threads (séquentiel) à 16 threads.

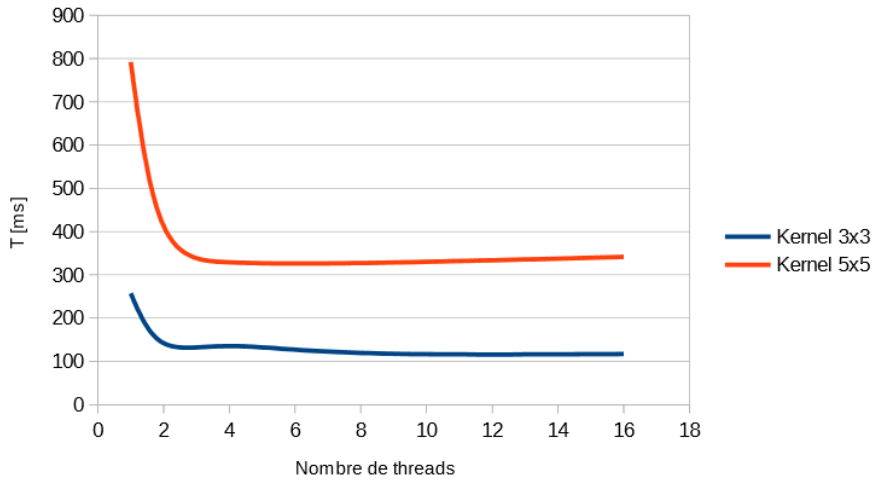


FIGURE 4.1 – Mesures du temps de la convolution par rapport aux nombre de threads

Sur le graphique ci-dessus, les performances mesurées pour un kernel de 3x3 est représentée par la courbe bleue ainsi que le kernel 5x5 par la courbe rouge.

Nous constatons pour les deux courbes que nous obtenons un gain de performance non-négligable lorsque l'on effectue l'opération à partir de 2 et 3 threads.

N threads	Kernel 3x3 [ms]	Kernel 5x5 [ms]
1	257.4	792.23
2	142.23	412.92
4	135.45	329.09
8	119.58	327.4
16	117.05	341.5

FIGURE 4.2 – Tableau des mesures pour une image de taille 1024x711 pixels

Nous avons également tenté d'augmenter le nombre de threads mais le temps de convolution reste plus ou moins identiques.

La mesure pour la grande images s'est effectuée dans les mêmes conditions que les premières mesures.

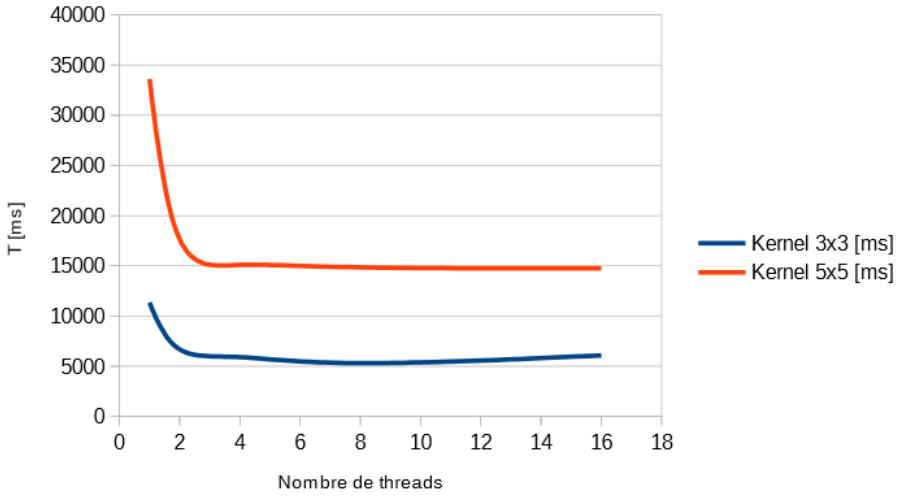


FIGURE 4.3 – Mesures du temps de la convolution par rapport aux nombre de threads

On constate que le rapport de performance est similaire à celui de la mesure effectuée sur la petite image.

N threads	Kernel 3x3 [ms]	Kernel 5x5 [ms]
1	11353.89	33625.23
2	6697.1	17675.4
4	5917.18	15096.34
8	5315.12	14850.83
16	6086.94	14759.43

FIGURE 4.4 – Tableau des mesures pour une image de taille 7359x4531 pixels

CHAPITRE 5

Conclusion

5.1 Commentaires et conclusion

Durant ce projet, nous avons appris à utiliser et mettre en oeuvre la librairie `<Pthread.h>` pour effectuer une convolution 2D sur une image de type `ppm`.

Afin de partager la tâche sur plusieurs threads, nous avons décider de diviser l'image horizontalement de manière équitable pour chaque thread afin d'accélérer le processus de convolution.

Pour terminer, nous constatons grâce aux mesures effectuées que nous gagnons un gain de temps d'environ 50% à partir de 2 threads. Nous pouvons obtenir plus de 55% de gain de temps avec plus de 4 threads.

Christopher Rejas
Fiorenzo Poroli