Gizel HADJI (2152684)

Quentin LEGRET (2150852)

# Rapport
# Convolution avec CUDA

**Généralités**

Pour la compréhension du sujet, nous nous sommes appuyés sur les explications de Naoyuki Ichimura, ce blog nous a grandement aidés à développer notre travail ainsi que sur le livre *Cuda by examples* de Jason Sanders et Edward Kandrot.

Le principe du programme est d'appliquer un filtre qui est une matrice sur une image afin de créer différent effet sur celle-ci. Tout d'abord il est essentiel de préparer l'image avant le traitement, nous devons ajouter des bordures a celle-ci au début nous l'avons implémenté à la main puis nous avons découvert qu'OpenCv peut le faire. Ensuite les filtres sont faits maison également, pour des soucis de simplicité nous avons testé seulement 2 filtres, Simple Blur (de la taille voulue, ce qui nous permet de faire des tests de performance) et un Left Sobel (forcément de taille 3). Une fois le traitement de préparation est fait nous pouvons envoyer les données sur le GPU. (Pour plus de détails sur le code veuillez lire les commentaires du fichier conRgb.cu)

**Réalisations**

Pour nous familiariser avec la convolution nous avons commencé par réaliser le travail en nuances de gris, en effet nous appliquons un grayscale sur l'image afin d'avoir qu'une information par pixel à traiter. (C'est le fichier conGray.cu)

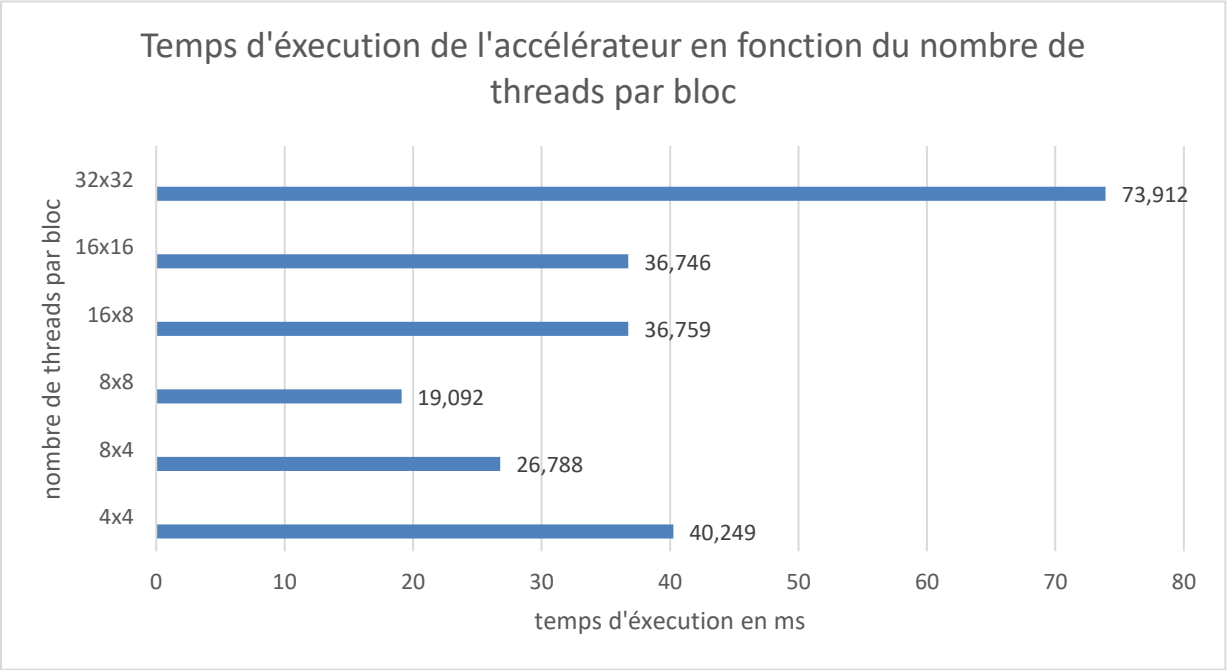
Puis une fois que nous avons bien compris le fonctionnement nous avons implémenté une solution qui traite les informations du RGB, (ficher conRgb.cu) ici rien de plus optimisé, nous multiplions certaines informations par 3 afin de traiter les couleurs. Dans le kernel, le filtre (étant le même qu'avant) doit être parcouru pour chaque couleur du pixel, et la somme des pixel*filtre se fait par couleur.

Jusqu'ici, toutes les données sont copiées sur la mémoire globale. Dans une optique d'optimisation nous avons placé le filtre dans la mémoire constante (le fichier conConst.cu). Et pour finir nous avons utilisé les streams pour copier l'image de manière asynchrone sur le kernel (le fichier conStream.cu).

**Performance**

Pour obtenir les performances nous utilisons la commande nvprof.

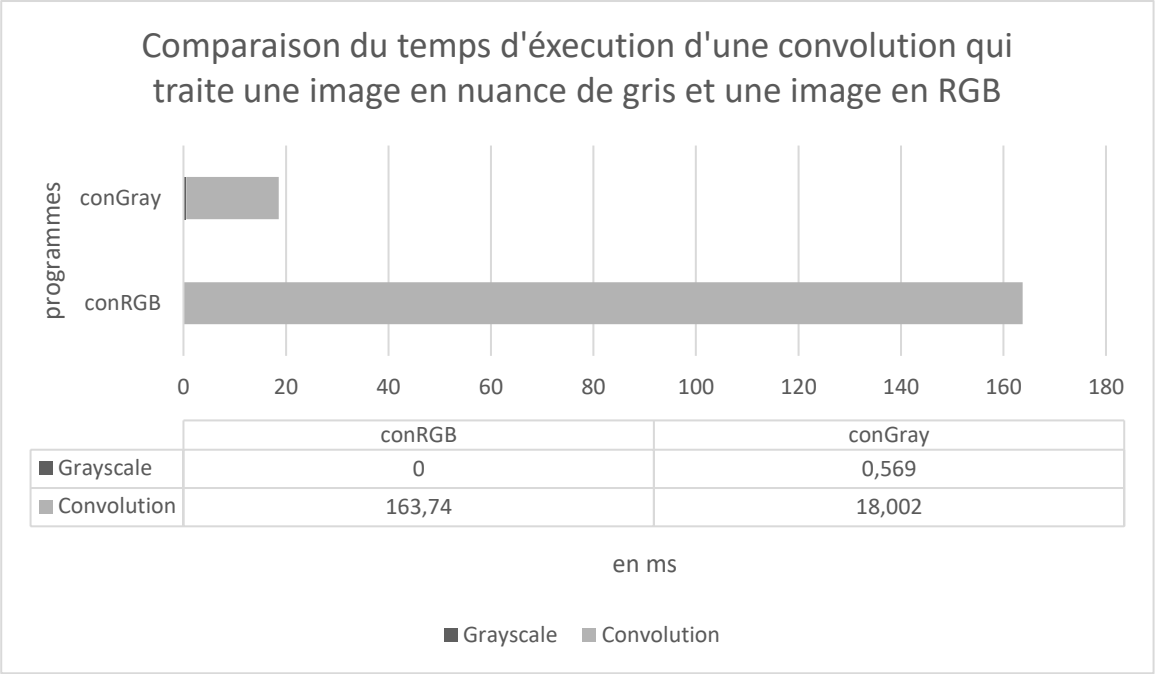
Dans un premier temps nous avons testé le nombre de threads par bloc avec le fichier conRgb, c'est-à-dire où toutes les données sont sur la mémoire globale.




*Les tests sont réalisés avec une image en résolution 3840x2160 et le filtre est de taille 3x3, le temps le plus court est le plus performant.*

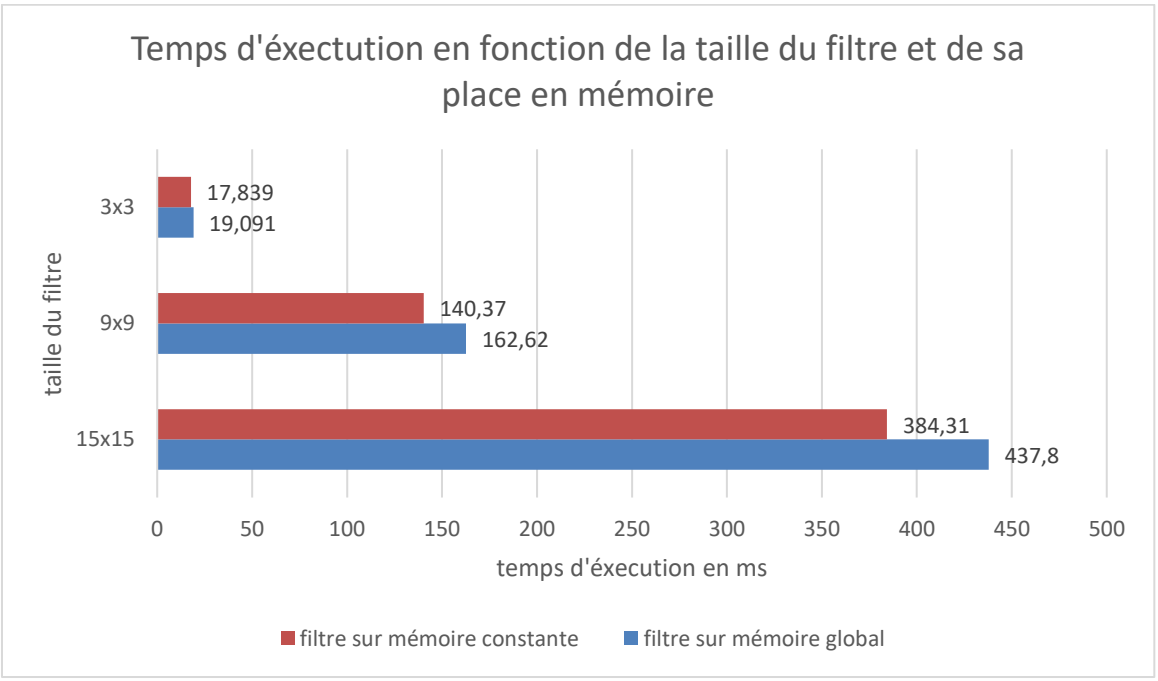
Nous observons que notre fonction est plus efficace dès que le bloc est de dimension (8,8).

On trouve assez pertinent de montrer la différence des performances entre un programme qui fait une convolution avec un image en RGB et un programme qui fait un Grayscale avant la convolution. Le temps d'exécution est grandement réduit avec un pré-traitement en Grayscale car on a plus qu'à traiter une seule information par pixel. A noter que la différence se creuse si l'image est plus grande et si le filtre est plus grand.




*Les tests sont réalisés avec une image en résolution 3840x2160 et le filtre est de taille 9x9, le temps le plus court est le plus performant.*

Ensuite nous avons cherché à placer notre filtre sur la mémoire constante du GPU, car cette mémoire a un temps d'accès plus rapide que la mémoire globale mais elle n'est pas très grande ce qui va limiter notre taille de filtre a 79x79 car 79x79x4(float) = 25KB, or la plupart des GPU actuel possède 32KB ou 64KB de mémoire constante. Ce qui reste une taille de filtre très honnête.




*Les tests sont réalisés avec une image en résolution 3840x2160, le temps le plus court est le plus performant.*

Nous remarquons que plus le filtre est grand plus notre *accélérateur* avec le filtre en mémoire constante est performant par rapport à celui qui a le filtre en mémoire globale. Ceci est logique du fait que le GPU va faire plus de demande d'accès au filtre et le temps d'accès à une donnée en mémoire constante est plus rapide.

Et nous avons fini par réaliser une version avec des streams pour envoyer les données de l'image de manière asynchrone, cette version n'est pas complète, en effet par souci de simplicité nous avons tous simplement coupés en 2 l'image en ajoutant les bordures nécessaires. Cela crée un problème visible avec des filtres supérieurs à 3x3. Au niveau des performances nous n'avons pas pu les tester. Vous nous avez répondu sur Discord de quelle manière on doit procéder, toutefois nos machines sous linux n'ont pas les configurations requises pour faire marcher le SDK Nvidia. On a essayé d'installer sur une machine plus performante Ubuntu 20.04 mais cette version n'est pas compatible avec le SDK Nvidia. C'est à ce moment-là qu'on laissé tomber pour se concentrer sur d'autres matières, car comme vous nous l'avez dit on peut passer des mois à améliorer les performances de notre programme mais il faut bien s'arrêter quelque part.

On aurait voulu faire un comparatif avec streams et sans streams selon la taille de l'image mais comme nous ne pouvons pas le visualiser nous allons juste faire un dernier test qui montre le temps d'exécution selon la taille de l'image.

Nous voilà à la fin de notre rapport, on regrette toutefois de ne pas avoir fait un programme qui utilise la shared memory qui aurait été très efficace mais surement toute aussi chronophage.