Programmation Graphique

Traitement d'image (notamment avec CUDA)

Brice JACQUESSON, Ahmet ONSEKIZ, Matthéo PAILLER

Introduction

Pour ce projet, nous avons dû implémenter plusieurs algorithmes de traitement d'images, parmi une sélection de divers algorithmes. Ces algorithmes utilisent des matrices de convolution, qui, en parcourant l'image, vont modifier le pixel traité mais aussi les pixels voisins de celui-ci. Plus la matrice de convolution sera grande, plus il y aura de pixel à traiter à chaque étape. C'est pour cela que nous avons choisi des algorithmes avec différentes tailles de matrices de convolution:

- L'algorithme box blur, avec une matrice de convolution de 3*3
- L'algorithme edge detection, avec une matrice de convolution de 3*3
- Le laplacian gauss, avec une matrice de convolution de 5*5
- Le gaussian blur, avec une matrice de convolution de 7*7

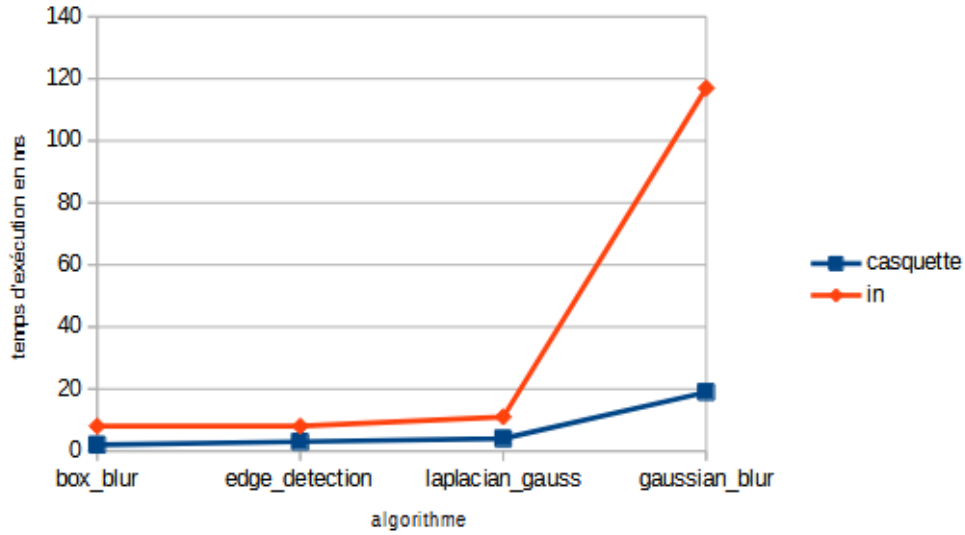
Nous allons, pour chaque algorithme, programmer différentes versions du filtre, pour comparer leur efficacité:

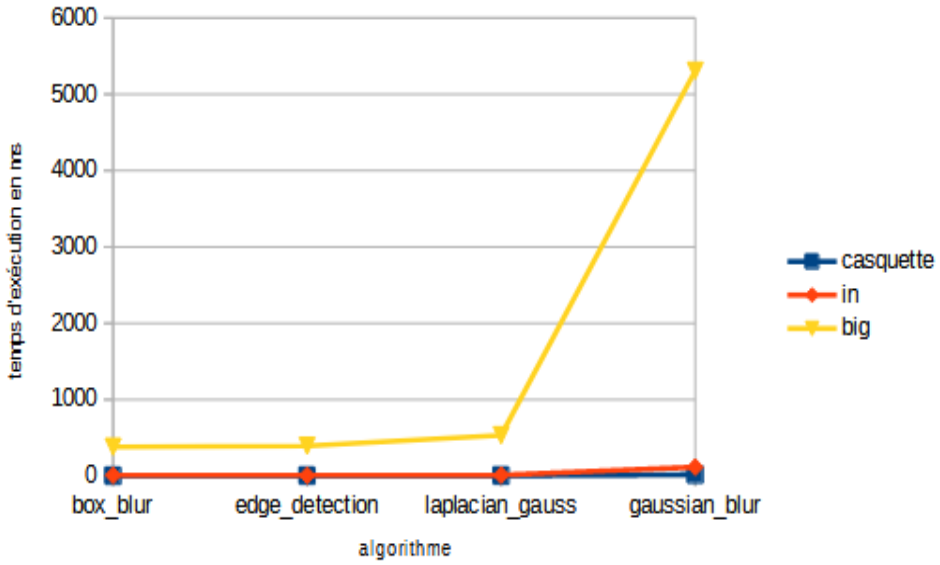
- Une version en C++
- Une versions en CUDA, en 2 étapes
- Une versions en CUDA, en 2 étapes avec mémoire partagée
- Une versions en CUDA, en une étape avec mémoire fusionnée
- Une versions en CUDA, avec des streams

De plus, chaque filtre sera appliqué à une image, à laquelle on aura au préalable appliqué un filtre grayscale.
Pour cela, nous sommes partis des corrections des TPs, disponibles sur Célène.
Nous avons, enfin, fait les différents tests sur 3 images différentes:

- casquette (612*612)
- in (1920*1200)
- big (14400*7200)

Partie 1: les versions c++

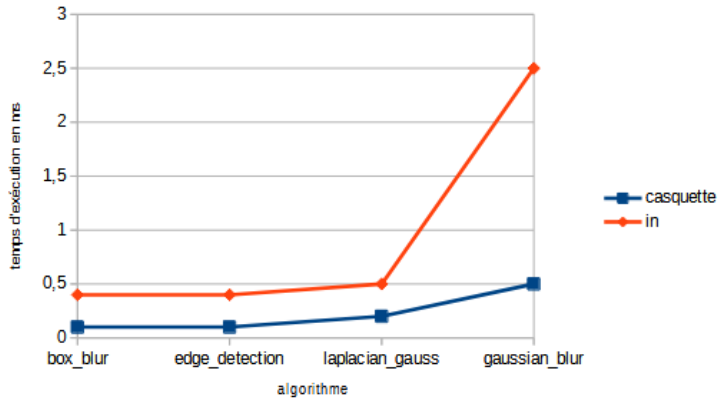


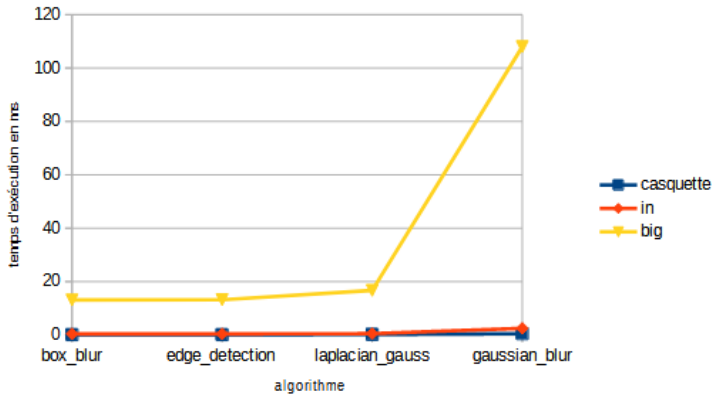


Pour ces versions C++, on a utilisé les librairies il.h pour récupérer l'image et la traiter. On voit bien que pour chaque image, plus la matrice de convolution est grande, plus le temps d'exécution est grand lui aussi. Cette affirmation est d'autant plus vraie que la taille de l'image augmente.

Partie 2: les versions CUDA

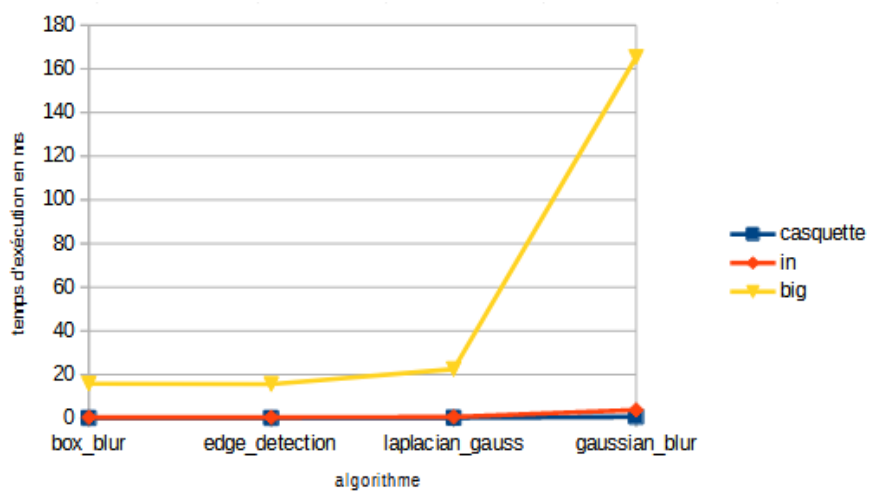
2.1: version 2 étapes

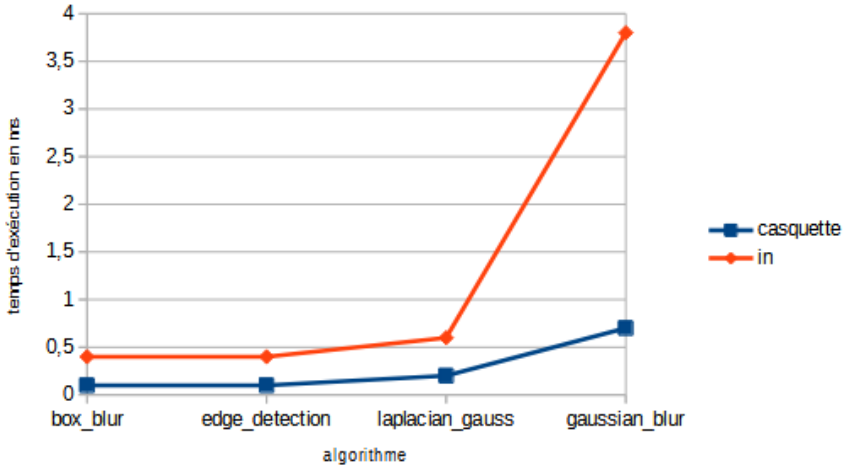




Pour cette version à 2 étapes, on applique d'abord un algorithme de grayscale pour obtenir une nouvelle "image". Suite à cela, on va appliquer à cette nouvelle image un algorithme pour le filtre qu'on implémente, qui va, pour chaque pixel, appliquer un convolution à celui-ci et son voisinage. La mémoire est ici globale, c'est-à-dire qu'on va utiliser toute la mémoire de la carte graphique, dans laquelle on va devoir allouer de l'espace pour pouvoir stocker nos données. On voit, encore une fois, que plus la matrice de convolution est grande, plus le temps d'exécution est grand. Le temps d'exécution est aussi grandement affecté par la taille de l'image.

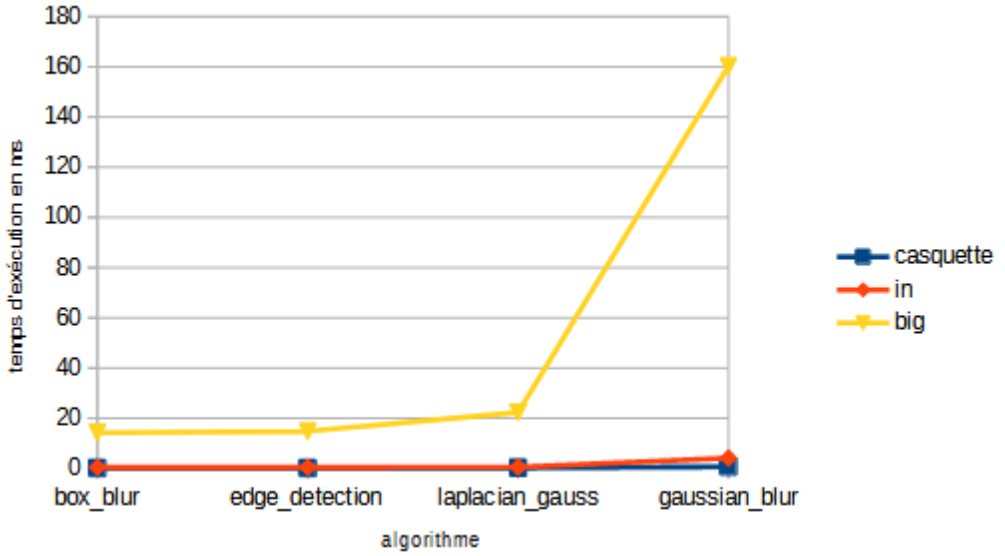
2.2: version 2 étapes avec mémoire partagée.

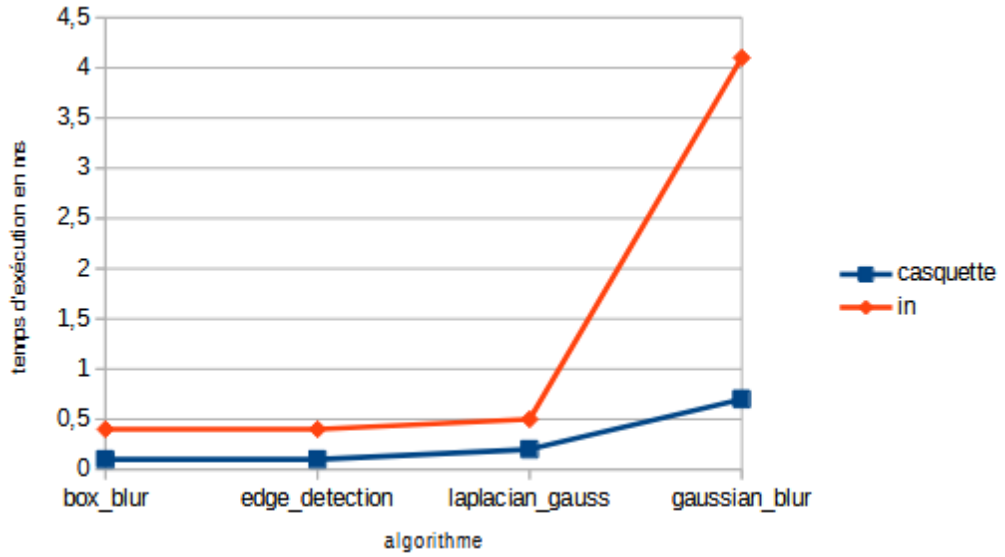




Pour cette version à 2 étapes, on va encore appliquer l'algorithme de grayscale pour obtenir une nouvelle “image”. Suite à cela, on va encore appliquer à cette nouvelle image un algorithme pour le filtre qu’on implémente, à la différence que cette fois-ci, la mémoire est partagée, c'est-à-dire que chaque bloc a un espace mémoire propre, qui sera donc (normalement) accessible plus vite. On voit, encore une fois, que plus la matrice de convolution est grande, plus le temps d’exécution est grand. Le temps d’exécution est aussi grandement affecté par la taille de l’image.

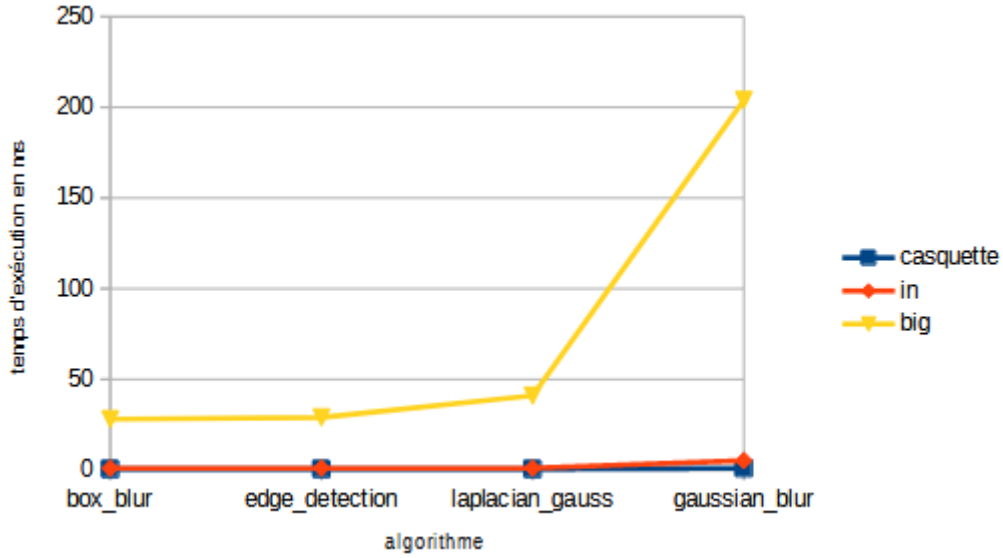
2.3: version fusionnée

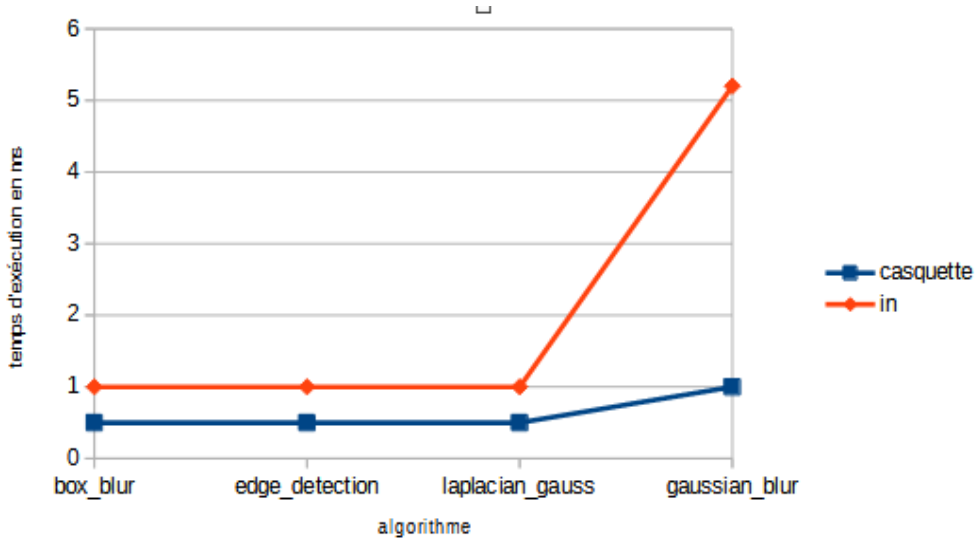




Pour cette version, on fusionne les deux algorithmes de grayscale et du filtre dans un seul kernel. On va utiliser encore une fois la mémoire partagée, mais on va synchroniser les threads pour que tout le grayscale soit calculé avant d'appliquer le filtre. On peut, comme pour les autres versions, constater que le temps d'exécution augmente selon la taille des matrices de convolutions, mais aussi selon la taille de l'image.

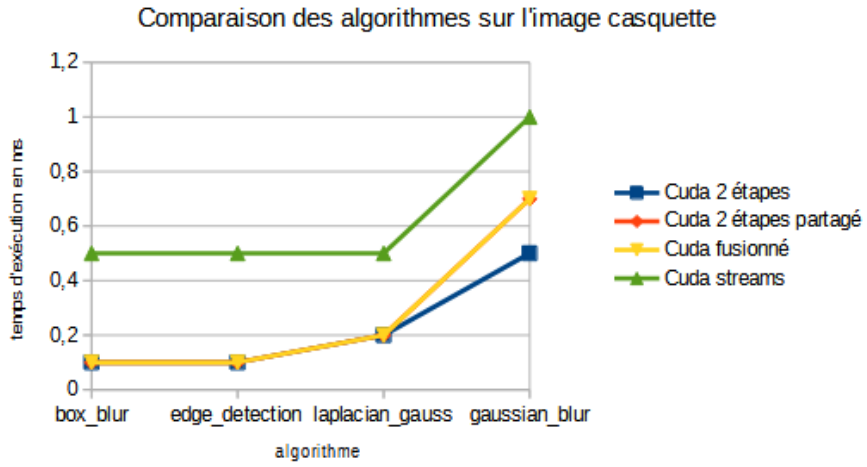
2.4: version avec streams



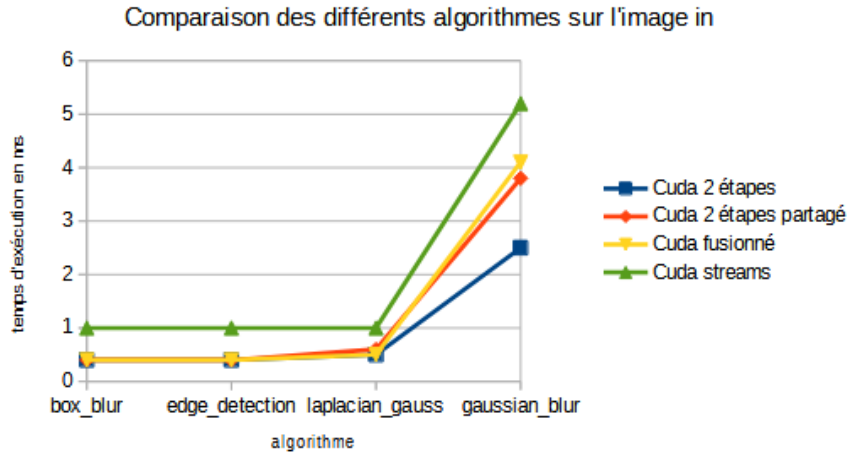


Dans cette version, on utilise des streams, en reprenant cependant l'idée de la version fusionnée. On a, dans notre cas, utilisé deux streams. On peut, comme pour toutes les autres méthodes, constater que le temps d'exécution augmente quand la taille de la matrice de convolution ou la taille de l'image augmentent. Ces streams permettent une meilleure gestion de la mémoire du GPU.

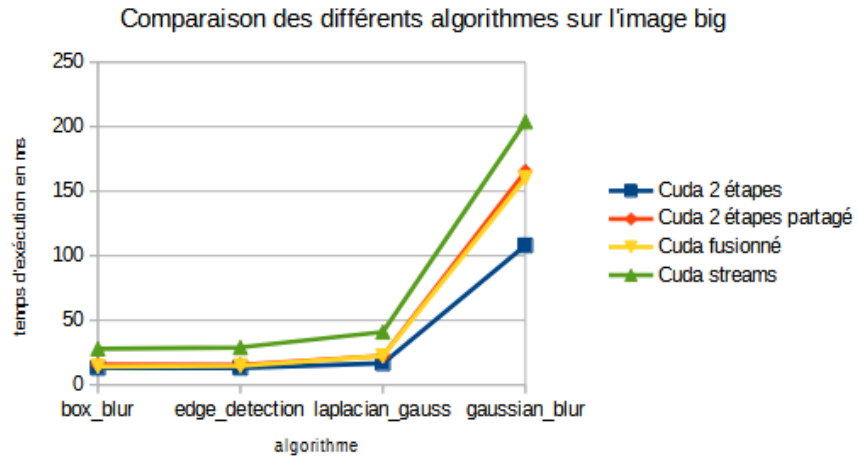
Partie 3: Discussions sur les résultats (temps d'exécution)



(1)



(2)



(3)

On va ici comparer l'efficacité des différentes versions pour chacune des images. La version c++ étant extrêmement lente par rapport aux versions CUDA, nous ne l'avons pas ajoutée dans les graphes de comparaison. Pour l'algorithme du flou gaussien sur la plus grosse image, on a même un temps de calcul de plus de 5 secondes !!!

On peut remarquer que le temps d'exécution des versions 2 étapes, 2 étapes en mémoire partagée, et fusionnée, est très similaire pour les algorithmes box blur, edge detection, et laplacian gauss, et croît très légèrement avec la taille de leurs matrices de convolution respectives. Cependant, ces 3 versions ont un temps d'exécution qui va varier un peu plus pour l'algorithme gaussian blur, qui lui utilise des matrices de convolution de 7*7. De ces 3 versions, la version 2 étapes est la plus rapide. Nous pensions, au contraire, qu'elle allait être la plus lente des 3, mais il s'est avéré que c'était tout le contraire. En comparant avec les résultats que nous avions obtenus au TP4 (et les résultats dans la correction de ce même TP), nous nous sommes aperçus que nos résultats ici n'étaient pas si surprenants, et que cette version à 2 étapes était bien la plus rapide. Cependant nous avons aussi remarqué aussi que cela dépendait de la taille des blocs choisie. En effet, si l'on choisit la taille (64,8), la version fusionnée devient la plus rapide des versions CUDA, et la version 2 étapes devient la plus lente.

Il en est de même pour les streams. En effet, on remarque, sur les graphes de comparaison, qu'à chaque fois la version avec streams est la plus lente de toutes les versions CUDA. Nous ne nous attendions pas non plus à ce résultat, et c'est pour cela que nous avons demandé à des camarades d'autres groupes si leur résultat était similaire, dont la plupart a obtenu le même genre de résultat. Nous avons par la suite essayé en doublant les streams, ce qui a légèrement réduit le temps d'exécution, mais ne l'a pas amélioré assez pour que la version avec streams dépasse les autres versions CUDA (mais elle reste tout de même plus rapide que la version c++, on ne peut pas lui enlever ça).

Pour conclure, on peut dire que toutes les versions CUDA sont largement plus optimales que la version c++, bien que les versions qu'on aurait pensé être des améliorations se sont avérées moins efficaces que prévu. Un axe d'amélioration possible serait de modifier la taille des blocs, afin d'adapter celle-ci pour chaque algorithme/filtre. Un autre axe d'amélioration serait d'améliorer la communication entre streams.

Partie 4: Annexe - Résultats des algorithmes sur nos images

Filtre Box Blur sur l'image casquette.jpg. (disponible sur le git.)

Filtre Gaussian Blur sur l'image casquette.jpg. (disponible sur le git.)

Filtre Laplacian Gauss sur l'image in.jpg. (disponible sur le git.)

Filtre Edge Detection sur l'image casquette.jpg. (disponible sur le git.)

D'autres images filtrées sont disponibles sur le git.