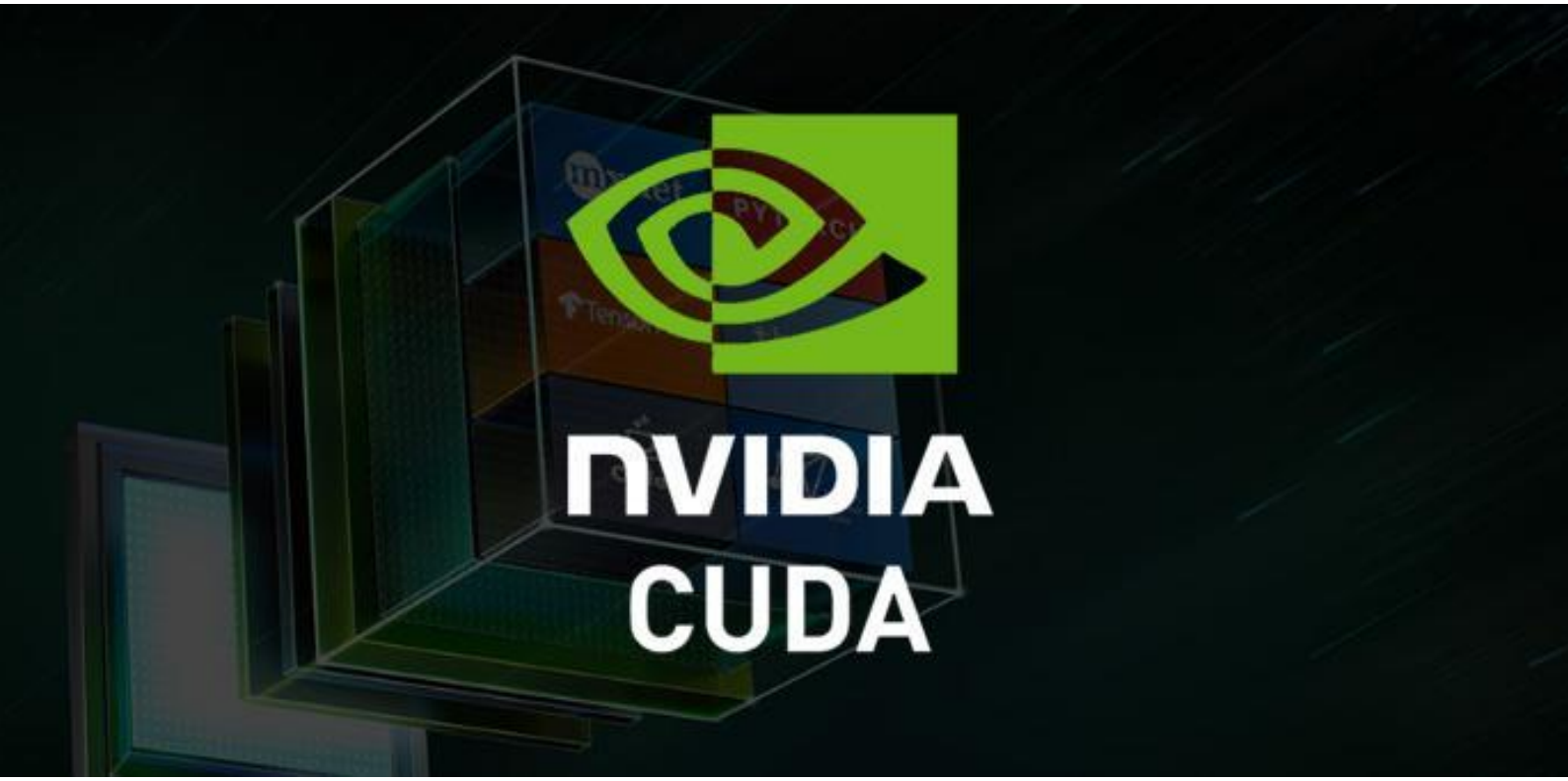


Programmation graphique – CUDA

SCHIVRE Nicolas

RIO Corentin

BADET Maxime

Introduction

Le but de ce projet était d'implémenter quatre filtres d'image différents via une programmation CPU et une programmation CUDA sur GPU NVIDIA ainsi que d'analyser les performances de ces implémentations selon différents paramètres. Ces filtres utilisent le principe de la convolution par lequel à partir d'un pixel on va modifier sa valeur selon la valeur de ses voisins. Les matrices de convolutions appliquent donc des poids aux valeurs des pixels des voisins ainsi que la valeur de base du pixel afin d'en donner une nouvelle. Parmi les filtres que nous avons décidé d'implémenter, deux d'entre eux ont en entrée une image en couleurs et les deux derniers utilisent une image en noir et blanc. Pour ces derniers il nous à fallu convertir les images en noir et blanc grâce à un filtre. Nous avons choisi le filtre grayscale pour lequel nous avions déjà une implémentation en programmation C++ (CPU) et CUDA (GPU).

Pour le choix des filtres, nous avons choisi d'implémenter le filtre de Sobel, le BoxBlur, le Laplace Operator et le Gaussian Blur.

Nous allons tout d'abord présenter le fonctionnement des différents filtres que nous avons énuméré ci-dessus, puis nous analyserons les données de performances que nous avons pu générer et ainsi déterminer qu'elle est l'utilisation optimale de ces filtres.

Nous utiliserons ces analyses afin de tirer des conclusions sur les performances et les optimisations en programmation sur carte graphique via CUDA.

Pour information, nous avons utilisé la carte graphique présente sur les ordinateurs de la salle E11 qui est une NVIDIA Quadro P400 avec un 256 CUDA coeurs et une interface mémoire de 64 bits. On suppose donc que cette carte graphique peut utiliser au maximum 256 blocs et 64 threads.

Note importante : La programmation et les tests de performances ont été réalisés en étroite collaboration avec le groupe de Baptiste LE GOUHINEC et Lucas LECHNER. En revanche, la partie analyse des données et l'écriture du rapport ont été effectuées de manière indépendante par chaque groupe.

Les différents filtres

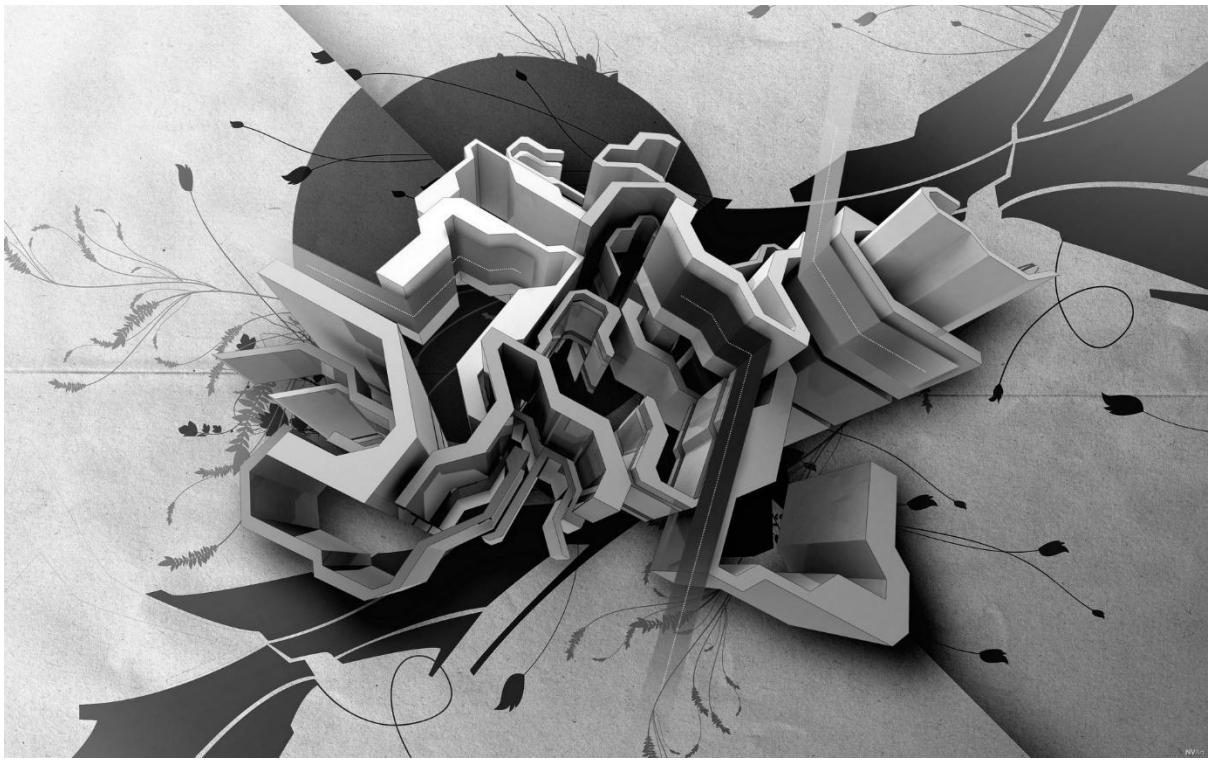
Grayscale

Ce filtre ne fait pas partie des filtres pour lesquels nous avons utilisé des matrices de convolution ni analysé les performances mais il nous semblait important d'en parler car il constitue une étape intermédiaire pour certains filtres.

En effet, ce filtre permet de convertir une image en couleurs vers une image en noir et blanc. Ce filtre est donc utile pour pouvoir utiliser certains filtres qui nécessite une image en noir et blanc sur une image en couleurs.

Pour convertir l'image en noir et blanc on applique un certains poids à chaque couleur du pixel pour avoir un rendu final gris. Chaque pixel est calculé de la manière suivante :

$$px = (307 * rouge + 604 * vert + 113 * bleu)/1024$$



Résultat obtenu après utilisation du filtre Grayscale

Sobel

Description :

Le filtre de Sobel est un filtre utilisé dans le traitement d'image pour la détection de contours (edges detection). Pour appliquer ce filtre, il faut au préalable utiliser le filtre grayscale qui permet de convertir l'image en couleur en noir et blanc et ce sera pareil pour tous les filtres suivants, nous utilisons une ou plusieurs matrices de convolution. Pour ce filtre on utilise deux matrices de convolution : une pour les approximations des dérivées horizontale et une pour les approximations des dérivées verticale (respectivement ci-dessous).

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Matrices de convolution utilisées par le filtre de Sobel

On applique donc ces deux matrices (par convolution) pour obtenir les approximations des dérivées horizontale et verticale que l'on va appeler *h* et *v*. On obtient donc la nouvelle valeur du pixel grâce à *h* et *v* de cette façon :

$$px = \sqrt{h^2 + v^2}$$

On fait juste attention à ce que les valeurs obtenues ne dépassent pas la valeur max pour un pixel noir et blanc (à savoir 65535). On met donc les pixels avec des valeurs plus hautes à cette valeur maximum.

Après l'application du filtre on voit donc bien que les contours sont plus apparents comme le démontre l'image ci-dessous :

Résultat après application du filtre de Sobel

Laplace Operator

Description :

Ce filtre, comme le précédent, est un filtre qui sert à la détection de contours. Il utilise lui aussi comme entrée, une image en noir et blanc et donc on doit appliquer un filtre de conversion d'une image en couleurs vers le noir et blanc avant de pouvoir utiliser le filtre. On va, de la même manière qu'avec le filtre précédent, appliquer le filtre Grayscale afin de faire cette conversion. Pour appliquer ce filtre, il existe plusieurs matrices de convolution, nous avons choisi la suivante :

-1	-1	-1
-1	8	-1
-1	-1	-1

Matrice de convolution utilisée pour le filtre de Laplace

Il existe différentes matrices de convolution pour ce filtre mais celle-ci accorde un poids plus important à la valeur d'origine par rapport à ceux des voisins. Après calcul via la matrice, on utilise un seuil afin d'affiner la détection. En effet, sans ce seuil le filtre ne permet pas vraiment de différencier les contours comme le montre l'image ci-dessous :

Résultat obtenu avec le filtre Laplacien sans seuil

Après l'ajout d'un seuil, on peut donc ajuster la précision de la détection (voir les photos ci-après) :

Résultat obtenu avec le filtre Laplacien avec un seuil à 64

Résultat obtenu avec le filtre Laplacien avec un seuil à 128

Résultat obtenu avec le filtre Laplacien avec un seuil à 256

BoxBlur

Description :

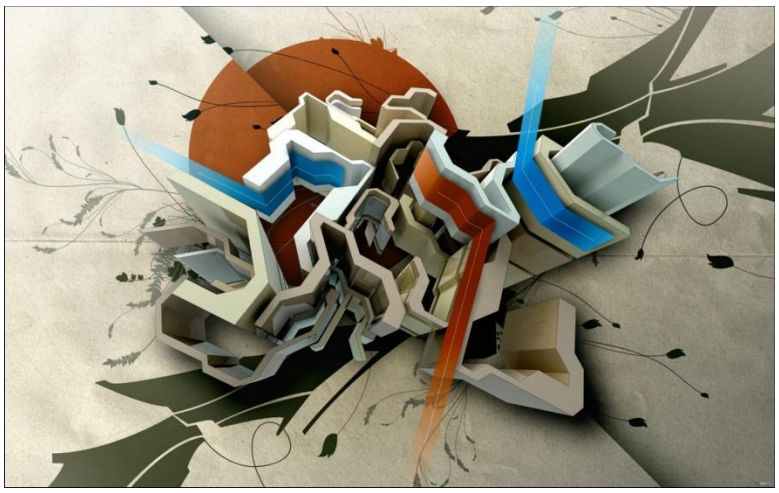
Ce filtre consiste à faire une moyenne des valeurs des pixels alentour afin d'en résulter en une image plus floue par rapport à l'image d'origine. La matrice de convolution utilisée pour ce filtre est donc la suivante :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Matrice de convolution du filtre BoxBlur

On accorde le même poids à chaque pixel alentour y compris le pixel d'origine. On divise ensuite le résultat obtenu via cette matrice par 9 afin d'avoir la moyenne qui est la valeur finale du pixel après application du filtre.

On obtient donc une image floutée comme ceci :



Résultat après application du filtre BoxBlur

Il est très difficile de remarquer les modifications du filtre sur une image de cette envergure (1920x1200) car comme la taille de la matrice de convolution est de 3x3 on ne fait qu'une moyenne des pixels que sur 1 pixel de distance. Pour plus d'efficacité du filtre il faudrait agrandir la matrice de convolution en 5x5 ou plus mais cela risque d'augmenter le nombre de pixels ou on ne peut pas calculer la moyenne car les voisins seraient inexistants (les pixels sur les bords de l'image).

Sur une image de plus petite taille on se rend plus compte de l'effet du filtre :

Résultat du filtre BoxBlur sur une image de dimension 50x35

On voit bien que les pixels ayant changé sont les pixels adjacents à un pixel d'une autre couleur uniquement.

Gaussian Blur

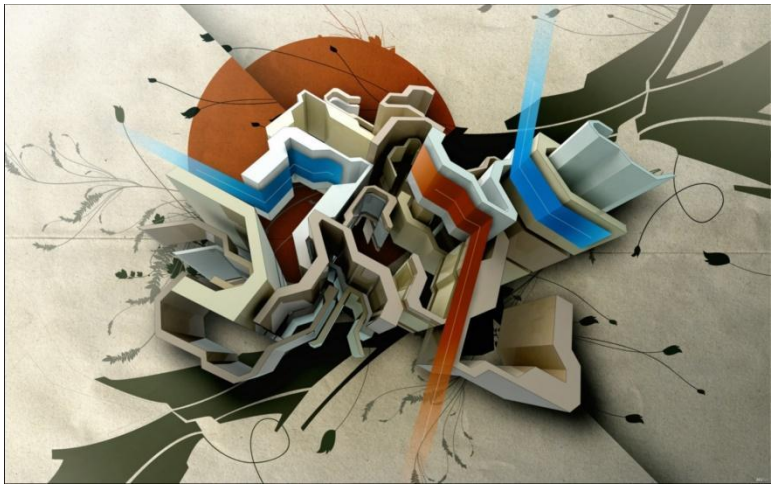
Description :

Ce filtre, à l'instar du Box Blur, sert à flouter une image en couleurs. Pour se faire, le filtre met un poids sur les valeurs des voisins (via la matrice de convolution) de telle sorte qu'une partie de la couleur des voisins se transmette à la valeur d'origine selon leur proximité. Pour implémenter ce filtre, nous avons choisi une matrice 5x5 pour changer par rapport au filtre BoxBlur. De ce choix, plus de pixels correspondant aux pixels des bords de l'image n'ont pu être calculés. Voici la matrice de convolution utilisée :

$$\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Matrice de convolution utilisée pour le filtre de Gauss

Pour avoir la nouvelle valeur du pixel, après avoir calculé la valeur via la matrice de convolution, il faut diviser le résultat par 256 qui est la somme des poids de la matrice afin d'obtenir une valeur de pixel possible pour une image.



Résultat obtenu après utilisation du filtre de Gauss

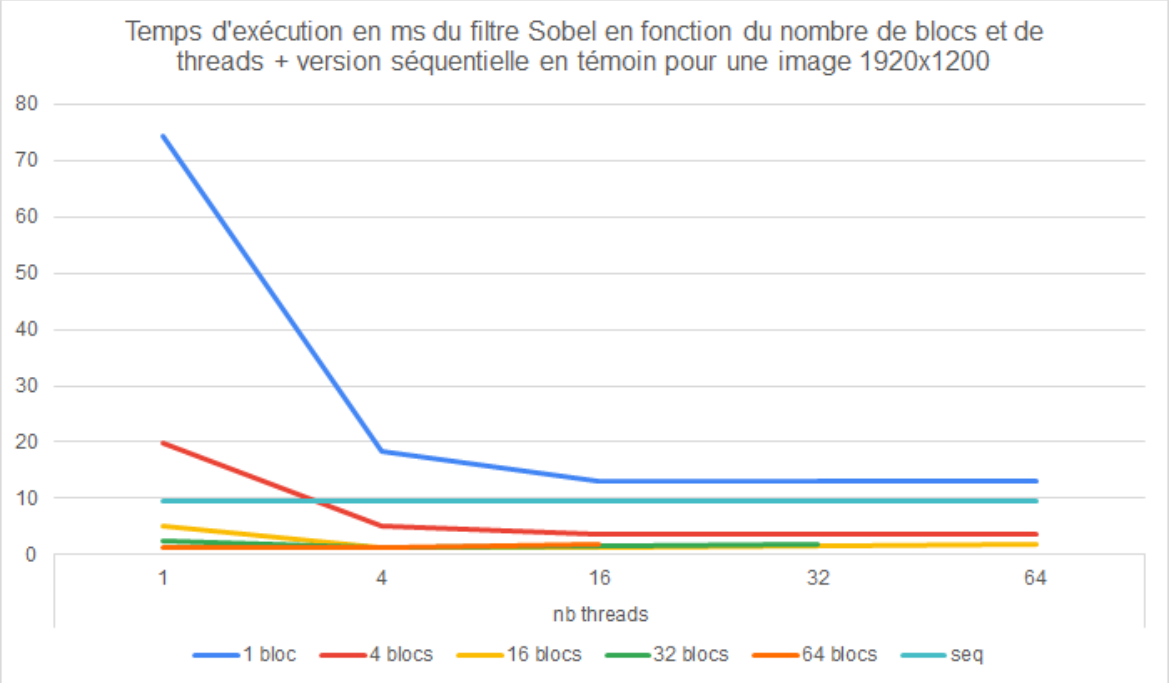
De la même manière que pour le filtre BoxBlur, on ne peut pas facilement distinguer le flou à l'œil nu sur cette image, car il se fait uniquement en prenant en compte les valeurs des voisins à 2 de distances. Pour augmenter l'efficacité de ce flou, il faudrait donc augmenter considérablement la matrice de convolution.

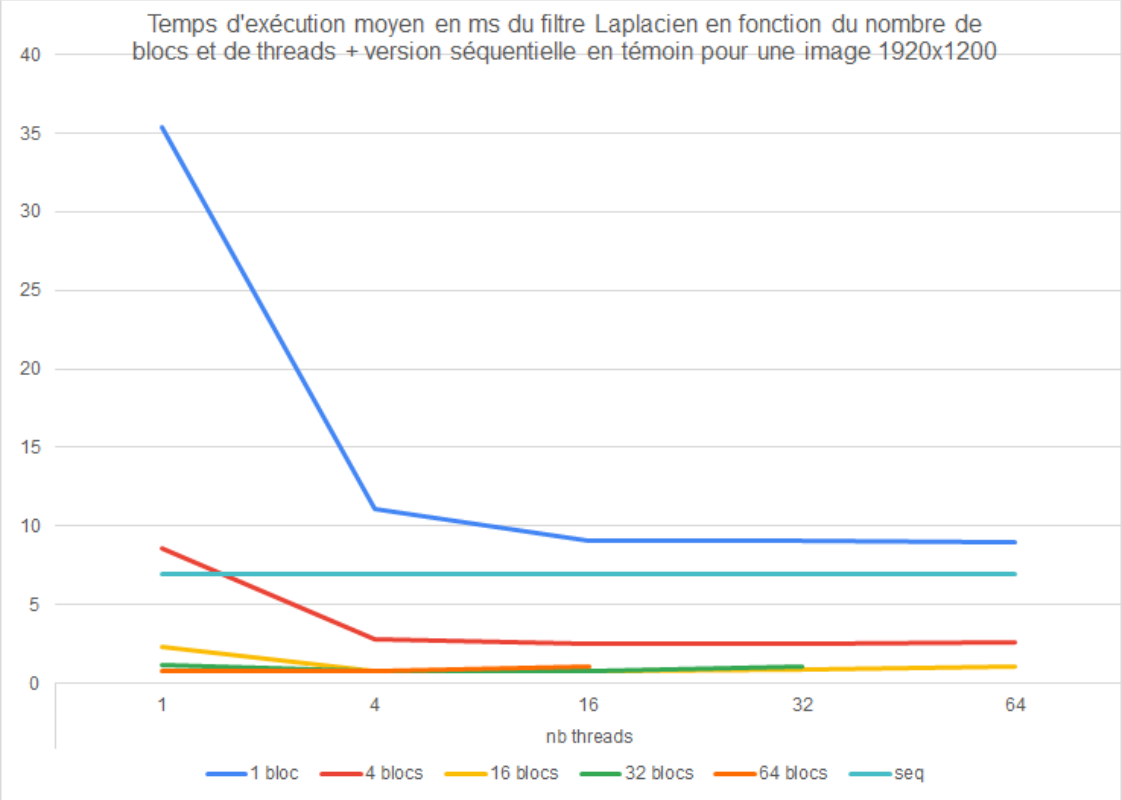
Analyse de performance

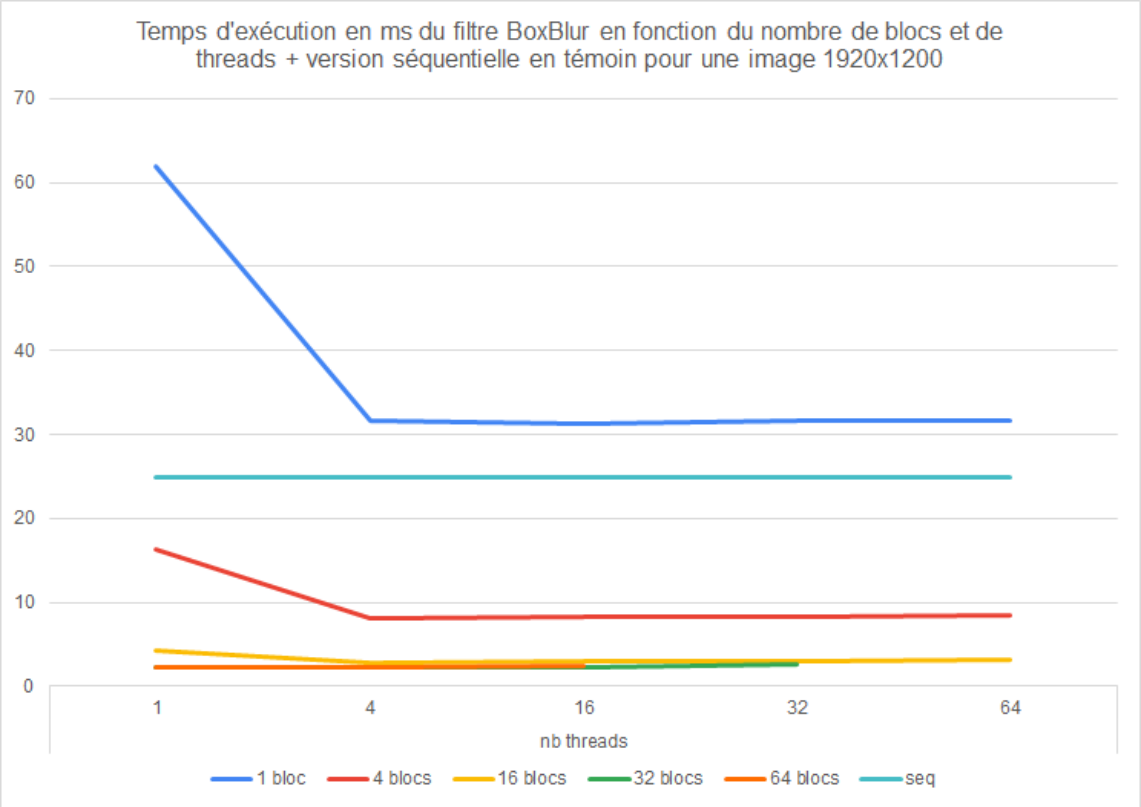
Pour générer ces analyses nous avons procédé de manière différente pour la version CPU (séquentielle) et pour la version GPU (CUDA). Pour la version CPU, nous avons lancé 10 fois le programme et nous avons fait la moyenne des 10 temps pour avoir notre valeur. Pour la version GPU, nous avons effectué 3 runs avec toutes les combinaisons possibles de nombre de blocs et de threads parmi 1, 4, 16, 32 et 64. Les tests ont été effectués sur 2 images de tailles différentes : une de 1920x1200 et une de 14400x7200.

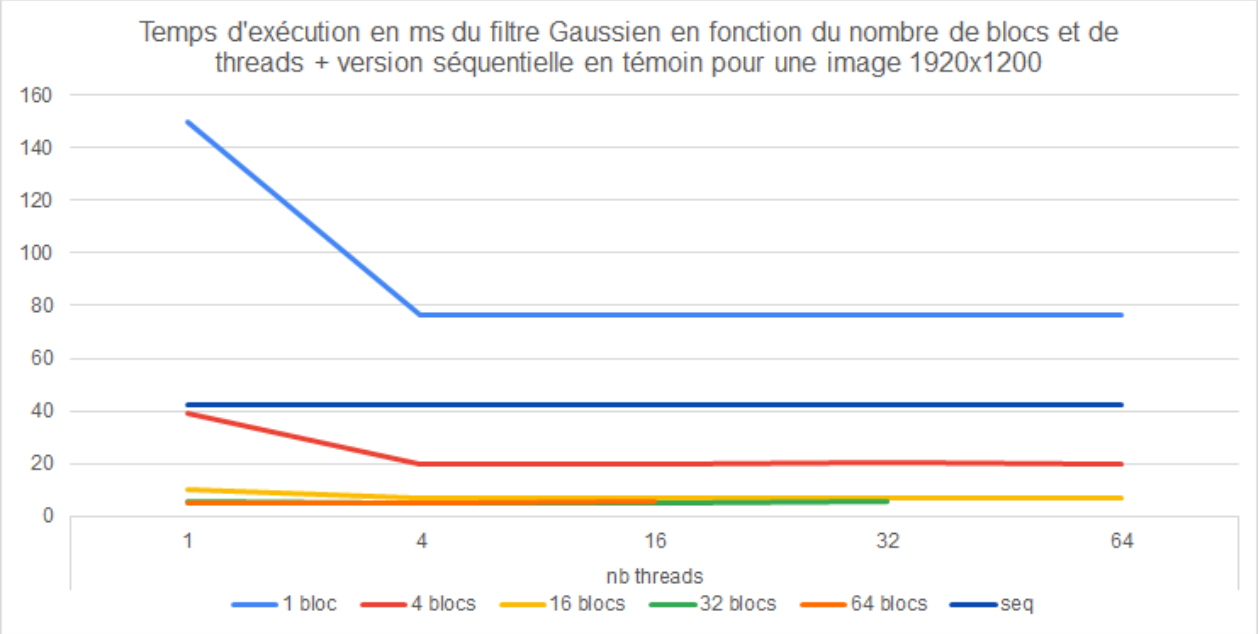
Vous pourrez trouver nos résultats sous forme de graphiques avec des analyses en poursuivant dans le rapport ou sous forme de données brutes qui sont à votre disposition dans le répertoire du projet. Parmi ces données, il arrive parfois qu'il y ait des N/A à la place des valeurs. Ceci s'explique par le fait que le filtre n'a pas fonctionné correctement sur l'image. En effet, nous obtenons une image noire avec un temps d'exécution très court (signe que le programme ne s'est pas déroulé entièrement).

Après avoir observé le temps d'exécution de tous les filtres, nous avons remarqué des tendances similaires entre les différents graphiques obtenus. Cela est compréhensible de part le fait que les filtres que nous avons implémentés utilisent le même procédé qui est celui de la matrice de convolution. Il est donc logique d'obtenir la même tendance dans les temps d'exécution.




Temps d'exécution moyen en ms du filtre Laplacien en fonction du nombre de blocs et de threads + version séquentielle en témoin pour une image 1920x1200
40
35
30
25
20
15
10
5
0
1
4
16
32
64
nb threads
1 bloc
4 blocs
16 blocs
32 blocs
64 blocs
seq


Temps d'exécution en ms du filtre BoxBlur en fonction du nombre de blocs et de threads + version séquentielle en témoin pour une image 1920x1200
70
60
50
40
30
20
10
0
1
4
16
32
64
nb threads
1 bloc
4 blocs
16 blocs
32 blocs
64 blocs
seq



Cependant, si on se penche sur l'analyse des courbes de performance en prenant le graphique concernant le filtre de Sobel, on remarque d'emblée que le temps d'exécution sur GPU pour 1 bloc et 1 thread est très élevé (environ 74 ms) pour une image relativement petite (1920x1200). Ce temps est d'ailleurs supérieur au temps de l'exécution en séquentiel. Cet écart peut notamment s'expliquer par le temps dépensé pour envoyer la matrice représentant l'image du CPU au GPU (allocation, copie d'espace mémoire...). De plus, l'utilisation d'un unique thread sur un seul bloc représente un comportement séquentiel. Ceci nous confirme donc que l'utilisation d'un seul bloc couplé à un seul thread est très peu performant.

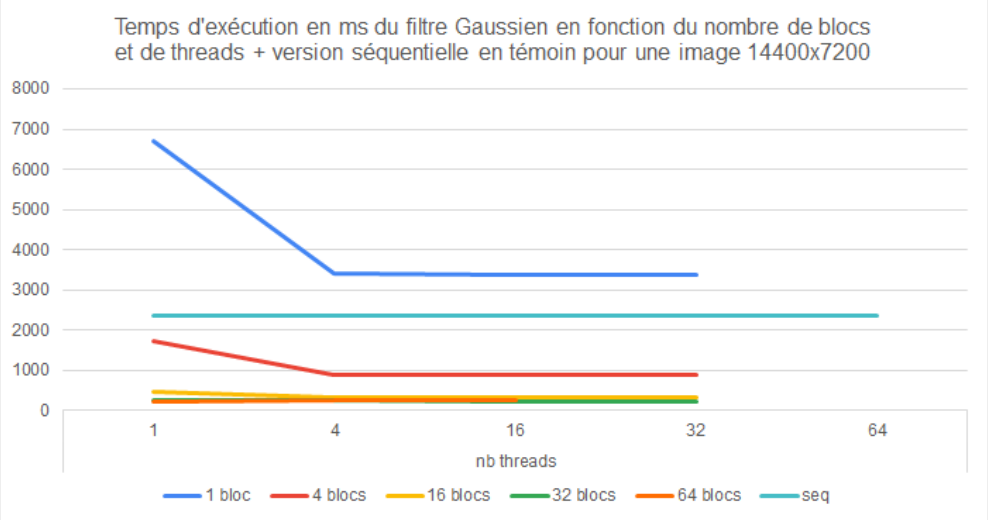
On observe néanmoins une baisse drastique du temps d'exécution (environ 4 fois plus rapide) lorsqu'on exécute les filtres avec 4 threads (environ 18 ms). Ceci s'explique logiquement parce que l'utilisation de 4 threads au lieu de 1 quadruple le nombre de pixels traités à chaque étape. Cependant, lorsque l'on utilise plus de 4 threads (16, 32 ou encore 64) la diminution du temps d'exécution est moins importante, nulle ou le temps d'exécution peut augmenter dans certains cas (comme pour 64 (threads par exemple). Le fait que la diminution du temps d'exécution est moins importante peut se justifier par le fait que les threads en programmation CUDA ne s'exécutent pas tous en même temps. On peut donc supposer que la carte graphique que nous utilisons n'exécute qu'environ 4 threads en simultané. Pour certains cas où le temps d'exécution augmente avec l'augmentation du nombre de threads, cela peut être dû au fait que la gestion des threads de manière interne par la carte graphique doit utiliser un certain nombre de cycles et donc ralentir l'exécution du programme. On ne peut cependant pas confirmer cette hypothèse car la gestion des threads par la carte graphique est une information que nous ne pouvons pas récupérer.

On peut aussi remarquer que pour un certain nombre de blocs et de threads, on n'a pas de valeurs. En effet, pour 32 blocs et 64 threads, 64 blocs et 32 threads et 64 blocs et 64 threads ont n'a pas de valeurs de temps car le filtre ne fonctionne pas. Cela est dû au fait que la carte graphique que nous utilisons ne permet pas la création d'autant de threads. On peut donc supposer que le nombre de threads maximum permis par la carte graphique est inférieur à de 32 * 64 = 2048 threads (car ce sont les valeurs maximum de blocs et threads pour lesquels nous avons plus de valeurs). On pourrait affiner le nombre de blocs et/ou de threads afin de trouver ce plafond avec plus de précision.

Nous avons aussi mené des tests de performance sur une image plus grande (14400x7200) afin de vérifier si ceci pouvait entraîner des changements au niveau des performances de nos programmes. Comme la résolution de l'image à traiter est plus élevée, le nombre de pixels l'est aussi. Ceci a pour conséquence de devoir traiter plus d'informations au niveau du GPU et donc d'augmenter le temps d'exécution du programme.

Cependant, si on compare les courbes pour nos deux images, on remarque que le temps sera plus long pour l'image avec la plus grosse résolution (car plus de pixels à traiter) mais il n'y aura pas de gain de performance. En effet, comme tous nos algorithmes utilisent la même méthode, ils ont la même complexité et donc par rapport à une même entrée ils auront un temps d'exécution proportionnel. Par exemple, pour le filtre Gaussien on passe d'un temps en séquentiel de 42 ms à 1900 ms (facteur 45) et si on regarde l'exécution pour 4 threads et 1 bloc on remarque qu'on passe d'un temps de 76 ms à 3417 ms (facteur 44) ce qui nous montre que le temps d'exécution est proportionnel en la taille de l'image.

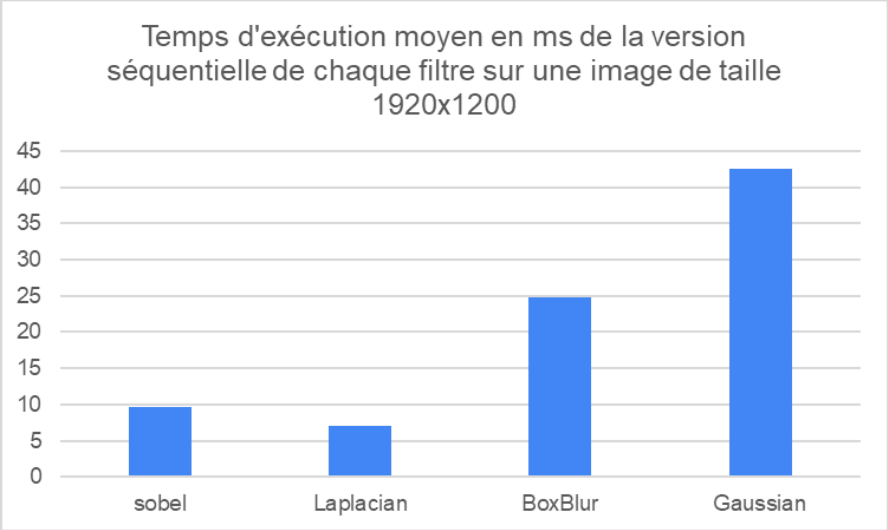
Lors des tests effectués nous avons rencontré une aberration avec le filtre Gaussien. En effet, sur la version GPU, nous obtenons une image noire avec un temps d'exécution très court. La seule particularité de ce filtre par rapport aux autres est l'utilisation d'une matrice de convolution de taille 5x5. Nous avons une hypothèse qui suggère que la carte graphique de la machine à l'université n'est pas capable d'appliquer des matrices de tailles trop élevées car elle serait limitée au niveau de ses performances.

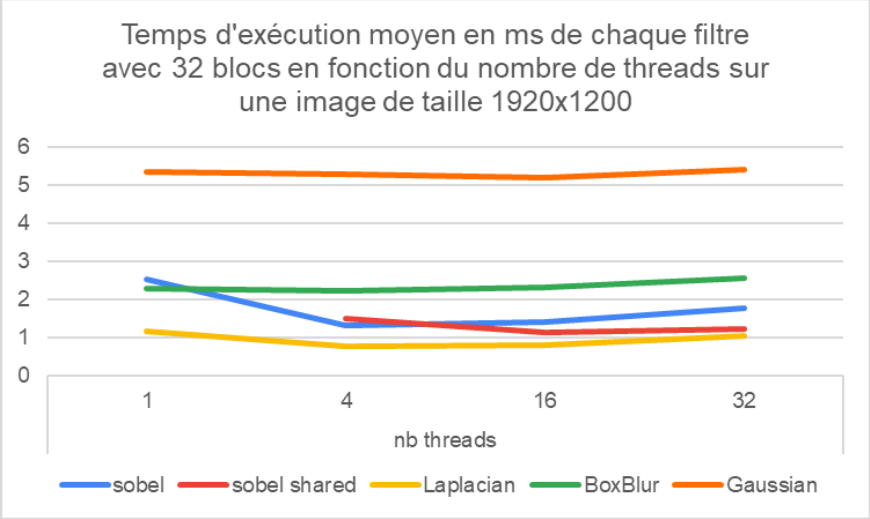


Si on s'intéresse maintenant aux différences de temps d'exécution entre les filtres, on remarque que le temps d'exécution du filtre Gaussian est nettement supérieur aux autres. Ceci s'explique par le fait que nous avons utilisé une matrice de convolution de taille 5x5 pour appliquer le flou Gaussien alors que nous avons utilisé une taille 3x3 pour les autres filtres.

On remarque aussi que le temps d'exécution pour appliquer le filtre Laplacien est plus court que Sobel car dans Sobel on utilise 2 matrices de convolution.

Le filtre BoxBlur s'applique sur une image en couleur, dont chaque pixel est composé de 3 composantes alors que les filtres Sobel et Laplacian s'appliquent sur une image en gris ce qui explique donc le fait que le filtre BoxBlur est 3 fois plus long à l'exécution que le filtre Laplacian (environ 7 ms pour le filtre Laplacian contre 25 ms pour le filtre BoxBlur).

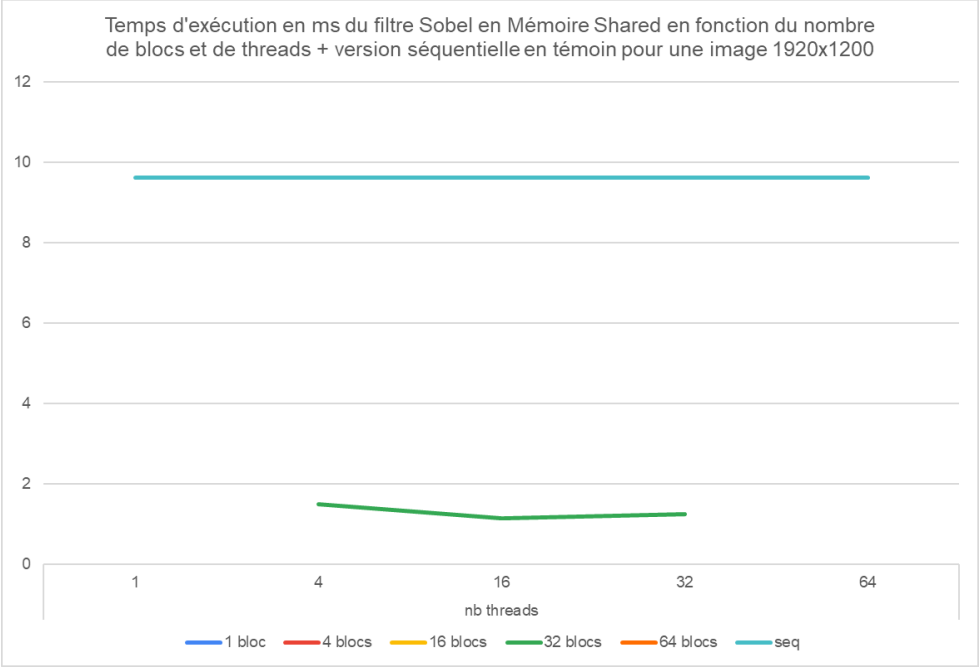


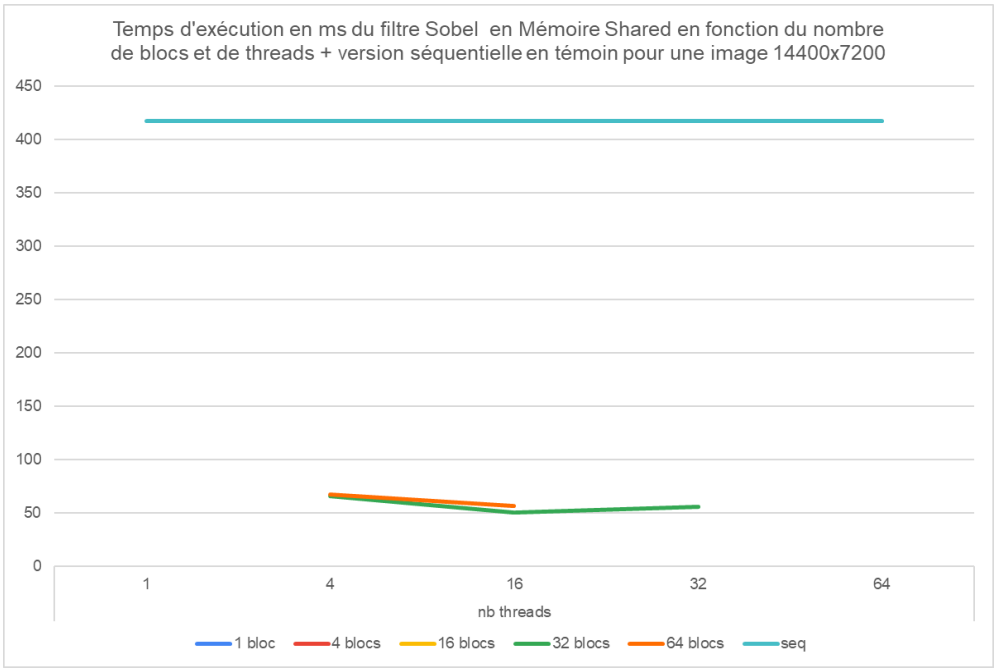


Pour le filtre de Sobel avec mémoire shared, on remarque que le temps d'exécution est légèrement en dessous du filtre Sobel classique lors de l'exécution sur 16 et 32 threads. Cela est compréhensible de par le principe de la mémoire shared qui permet de limiter l'accès à la mémoire globale. De ce fait, chaque bloc va copier la partie de la matrice image sur laquelle les threads du bloc ont besoin des valeurs. De ce fait, la récupération des valeurs dans la matrice se fait plus rapidement car cette dernière est beaucoup plus

petite. Cependant, on remarque que, pour 4 threads par bloc, le temps d'exécution reste plus long que la version sans mémoire shared. Cela peut s'expliquer par le fait que l'utilisation de la mémoire shared pour très peu de threads n'est pas très efficace. En effet, cela fait beaucoup d'opérations de copies. D'autant plus que les threads ont besoin des valeurs voisines, cela fait donc la copie de beaucoup de valeurs dites ghost. Au final, en prenant en compte l'énorme quantité de valeurs ghost copiées pour cette taille de matrice shared, la version shared fait un nombre d'opérations supérieur à la version de base.

Enfin, on remarque que pour la version shared, il y a nombre de compositions de blocs et threads pour lesquelles le filtre ne fonctionne pas. On suppose que cela est dû au fait qu'il faut un certain nombre de threads et blocs précis pour que l'utilisation d'un kernel CUDA shared soit possible.





Conclusion

Ce projet à servi de première approche à la programmation sur carte graphique NVIDIA via CUDA. Cela nous a permis de mieux comprendre les enjeux et le fonctionnement de la programmation sur GPU ainsi que l'importance du parallélisme dans cette approche. De plus, avec ce projet, nous avons pu nous initier au traitement d'images avec l'application de plusieurs filtres via des matrices de convolutions. La partie qui nous a le plus posé de problèmes pour l'implémentation des filtres en version CUDA est la gestion des tailles des matrices entre les matrices d'image en couleurs qui sont 3 fois plus grande que celle en noir et blanc.

Ensuite, l'analyse de performance qui était encore nouvelle pour nous (depuis le module de programmation parallèle du semestre passé) est devenue moins obscure car on s'y est exercé par le biais de ce projet.

Enfin, dans nos choix de filtre et leur implémentation, nous n'avons pas pu nous exercer à toutes les pratiques proposées par CUDA. Nous avons pu nous essayer à l'optimisation de nombre de blocs et threads ainsi qu'à l'utilisation de mémoire partagée (mémoire shared), mais nous n'avons malheureusement pas fait un filtre qui utilise les streams. On pense que l'utilisation des streams n'aurait pas pu augmenter la performance car les streams servent à paralléliser les étapes Host to Device, Kernel et Device to Host. Cependant, comme nos filtre ne nécessitait pas de traitement Host to Device et Device to Host mis à part la copie de l'image sur la carte graphique cela ne nous aurait pas fait gagner beaucoup en temps d'exécution, et, cela aurait même pu nous faire perdre du temps de par la maintenance des streams.

Lors de ce projet, nous avons rencontré quelques difficultés pour lancer le code sur machine. En effet, certains n'étaient pas en capacité de compiler le projet Cuda et ne pouvaient pas se déplacer à la fac. Certains ont donc dû réaliser le projet à en se connectant sur les machines à distance via Guacamole pendant que d'autres étaient sur place à la fac. Malgré les problèmes de fluidité des ordinateurs sur Guacamole, nous avons tout de même réussi à échanger et à nous coordonner pour réaliser les filtres et effectuer les tests sur machine.

Ce projet reste néanmoins un projet qui à apporté beaucoup sur notre vision de la programmation graphique. En effet, la programmation sur GPU est très utilisée dans de nombreux domaines de nos jours comme dans le deep learning et les jeux vidéos qui sont des domaines ou l'utilisation de matrices est essentiel et c'est exactement ce en quoi la programmation sur GPU excelle de par la conception d'une carte graphique. Enfin, cela nous à aussi appris à rendre important l'optimisation d'un programme ce qui est de plus en plus important dans la société actuelle où tout doit être optimisé.