

Projet CUDA

Programmation graphique

Membres du groupe :

Erwan CARNEIRO & Yanis LAGAB & Maxime LEMOULT

2022-2023

I - Présentations des algorithmes implémentés	2
1 - Box blur	2
2- Sobel	2
3- Détection de lignes avec convolution	2
4 - Opérateur de Laplace	3
II - Réalisation des filtres	3
1 - Version basique (CPP et 2D)	3
2 - Version stream	4
3 - Versions memory shared	5
4 - Version memory shared et stream	5
5 - Version 1D	6
6 - Résultats	6
III - Observations des performances	8
1 - Filtre Sobel	8
2 - Filtre Blur	11
3 - Filtre Horizontal-line detection	12
4 - Filtre Opérateur de Laplace	14
IV - Bilan	16
V - Annexe	17
1 - Spécification de la machine	17
2 - Calcul du temps de Sobel sans prendre en compte l'écriture des données entre le GPU et le CPU	17
3 - Temps de la version finale de Sobel avec image in.jpg	17
4 - Temps de la version finale de Sobel avec image in.jpg	17
5 - Variation du temps en fonction de Radius pour Cuda Blur	18
6 - Temps de la version finale de Laplace avec image in.jpg	18
7 - Temps de la version finale de Laplace avec image in2.jpg	18
8 - Temps de la version finale horizontal line detection avec image in.jpg	18
9 - Temps de la version finale horizontal line detection avec image in2.jpg	19

I - Présentations des algorithmes implémentés

Pour ce projet nous devons implémentés 4 algorithmes selon nos choix.

1 - Box blur

Le flou de boîte est un filtre d'image dans lequel la valeur de chaque pixel d'une image est la résultante de la moyenne de ses pixels voisins.

Les flous de boîte sont populaires et fréquemment utilisés pour se rapprocher d'un flou gaussien.

Dans notre implémentation nous avons choisi d'avoir une variable radius que représente le rayon du filtre de flou qui est appliqué à l'image.

Plus le rayon du filtre est grand, plus le voisinage de chaque pixel est grand, plus le voisinage de chaque pixel s'étend. Cependant, cela peut également ralentir le calcul, car un grand nombre de pixels doivent être pris en compte pour chaque pixel de l'image de sortie.

2- Sobel

Le filtre de Sobel détecte les contours dans une image en calculant le gradient de l'intensité de chaque pixel. Ceci indique la direction de la plus forte variation du clair au sombre, ainsi que le taux de changement dans cette direction. On connaît alors les points de changement soudain de luminosité, correspondant probablement à des bords, ainsi que l'orientation de ces bords.

Le filtre de Sobel calcule une approximation assez inexacte du gradient d'intensité, mais cela suffit dans la grande majorité des cas de son utilisation.

Il peut être utilisé pour une variété de tâches, comme la détection de bordures, la segmentation d'image, l'extraction de caractéristiques et la reconnaissance de formes.

Le filtre utilise deux noyaux de convolution, l'un pour la détection horizontale des contours et l'autre pour la détection verticale des contours. Chacun est un filtre 3x3 qui calcule la différence de luminescence entre les pixels voisins comme suit.

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

3- Détection de lignes avec convolution

Il existe plusieurs techniques pour la détection de lignes, mais les détecteurs linéaires les plus populaires sont les techniques basées sur la transformation de Hough et la convolution.

L'algorithme le plus simple à implémenter pour la détection de lignes en CUDA est celui basé sur la différence de luminosité entre les pixels adjacents. Cet algorithme est facile à comprendre et à mettre en œuvre cependant, il n'est pas le plus efficace pour la détection de lignes, car il est sensible aux variations locales de luminosité et peut générer des faux positifs ou des faux négatifs. Pour obtenir une détection plus précise et plus robuste des

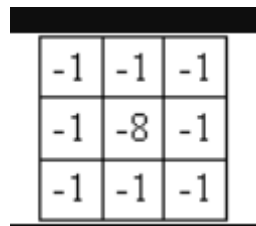
lignes, il faut utiliser des méthodes plus avancées telles que la convolution ou la transformation de Hough.

Ici nous avons donc choisi d'implémenter la détection de ligne par convolution en utilisant un noyau approprié basé sur Sobel.

4 - Opérateur de Laplace

Il permet de quantifier les variations dans toutes les directions autour d'un point. Il est souvent utilisé en traitement d'images pour détecter les contours et les bords d'une image.

En pratique, l'opérateur laplacien peut être calculé en prenant la somme des secondes dérivées partielles de la fonction par rapport à chacune des variables indépendantes. Dans le cas d'une image en deux dimensions, l'opérateur de Laplace peut être implémenté sous forme de convolution avec un noyau spécifique. Ce noyau peut être conçu pour mettre en évidence différents types de contours, tels que les bords forts ou les bords faibles.



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

Matrice qui applique l'opérateur de Laplace

II - Réalisation des filtres

Au début nous nous sommes partagé le travail comme suit :

- Box blur pour Erwan
- Horizontal line pour Yanis
- Laplace pour Maxime
- et Sobel sera développé en commun.

Nous avons quand même fait en sorte de garder un code similaire dans sa structure pour que le tout soit assez compréhensible. Les changements apparaissent vraiment dans les fonctions CUDA et les la partie du main appelant ce code.

1 - Version basique (CPP et 2D)

Nous avons d'abord réalisé pour chacun des filtres une version séquentielle C++ et une version Cuda sans optimisation particulière. Cela constituera déjà une base pour notre travail, nous permettant de l'adapter plus facilement en CUDA.

Notre version CUDA initiale fonctionne généralement de la même manière pour tous les filtres. La fonction CUDA prend en entrée une image en entrée et en sortie, les dimensions de l'image et parfois une option supplémentaire si celle-ci est un paramètre extérieur ayant une utilité dans le code. Il y a deux lignes de code pour obtenir les coordonnées 2D du thread dans la grille de threads CUDA, qui nous serviront à effectuer des calculs.

En soi, voici ce qu'on n'y trouve :

- L'utilisation de la fonction CUDA `cudaMalloc` pour allouer de la mémoire sur le GPU pour les images d'entrée et de sortie.
- L'utilisation de la fonction CUDA `cudaMemcpy` pour copier l'image d'entrée depuis le CPU vers le GPU.
- La définition de la taille des blocs et des grilles de threads à utiliser pour l'exécution du kernel, à savoir `blockSize` et `gridSize`.
- L'appel du kernel avec un schéma filtre `<<<gridSize, blockSize>>>(d_input, d_output, width, height)`
- L'utilisation de la fonction CUDA `cudaMemcpy` pour copier l'image de sortie depuis le GPU vers le CPU.
- Et la mesure du temps de calcul à l'aide des fonctions CUDA `cudaEventCreate`, `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventElapsedTime` et `cudaEventDestroy`.

Pour la suite, nous avons voulu améliorer les performances de notre version Cuda et plusieurs options s'offraient à nous. D'abord en utilisant les streams, puis avec une version utilisant la mémoire shared et enfin, faire une version qui utilise des streams et de la mémoire shared pour une version optimale.

2 - Version stream

L'intérêt principal des streams de CUDA est de permettre d'exécuter plusieurs opérations de manière simultanée, sans attendre que chacune soit terminée avant de passer à la suivante.

Voici comment on procède généralement :

- on copie la première moitié de l'image d'entrée sur le GPU avec le premier stream
- on copie la deuxième moitié de l'image d'entrée sur le GPU avec le deuxième stream
- ensuite, chaque stream va calculer le filtre avec sa moitié de l'image d'entrée
- on synchronise les streams avant l'envoi pour s'assurer que tout est bien calculé
- et enfin, le premier stream va s'occuper de charger l'image terminée sur le CPU

Cependant, il y a un léger problème.

Les opérations de copie de la première moitié de l'image d'entrée sur le GPU avec le premier stream et de copie de la deuxième moitié de l'image d'entrée sur le GPU avec le deuxième stream sont asynchrones. Cela signifie que les opérations de copie peuvent se produire dans n'importe quel ordre, ce qui peut causer des problèmes si une des opérations de copie n'est pas terminée avant que l'opération de filtrage correspondante ne commence. Pour remédier à cela, nous avons doublé la synchronisation des streams, désormais ils seront synchronisés avant et après le calcul.

Même si nous n'avons pas rencontré de problèmes majeurs sans cet ajout pendant les tests, cela pourrait arriver avec des tailles de blocs que nous n'avons pas testées, surtout avec les filtres utilisant des pixels voisins.

3 - Versions memory shared

Pour utiliser la mémoire partagée il est important de comprendre le principe du fonctionnement de CUDA car les principales modifications vont se passer dans la fonction de CUDA.

La mémoire partagée est globalement plus efficace que la mémoire globale pour deux raisons :

- La mémoire partagée est plus rapide à accéder que la mémoire globale, car elle est stockée localement sur le processeur et ne nécessite pas d'accès à distance. L'accès à la mémoire partagée peut être jusqu'à plusieurs ordres de grandeur plus rapide que l'accès à la mémoire globale.
- La latence de la mémoire globale est plus élevée que celle de la mémoire partagée. Lorsqu'un thread accède à la mémoire globale, il doit attendre que les données soient chargées depuis la mémoire globale, ce qui peut prendre plusieurs centaines de cycles de processeur. En revanche, la mémoire partagée est disponible instantanément pour les threads qui y ont accès.

En combinant ces avantages, la mémoire partagée peut offrir une amélioration significative des performances pour les applications qui nécessitent un accès fréquent et simultané à des données.

Voici le détail des ajouts nous permettant d'utiliser la mémoire partagée :

- Les variables *li* et *lj* représentent les coordonnées du thread dans le bloc.
- Les variables *w* et *h* représentent la taille du bloc en nombre de threads.
- Les variables *i* et *j* représentent les coordonnées globales du thread dans l'image.
- La variable *sh* est créée, c'est un tableau en mémoire partagée qui sert à stocker une partie de l'image.
- Ensuite, la valeur de la pixel situé à la position (*i,j*) dans l'image d'entrée est copiée dans la mémoire partagée, à la position (*li, lj*) du tableau *sh*.
- Enfin, la fonction `__syncthreads()` permet de synchroniser tous les threads du bloc pour qu'ils attendent que tous les threads aient terminé de copier les pixels dans la mémoire partagée avant de continuer. Cela permet d'éviter des incohérences de données en mémoire partagée et de garantir que les threads accèdent bien aux mêmes données.

4 - Version memory shared et stream

Le but de cette version est de combiner les versions shared et streams, censées être les plus performantes, pour évaluer si la combinaison des deux peut permettre des calculs encore plus rapides.

Le code n'est en soi qu'une version de shared du filtre dans laquelle on remplace la manière classique de traiter l'image par l'utilisation de deux streams pour exécuter plusieurs opérations de manière simultanée comme l'écriture entre le GPU et le CPU ainsi que l'application du filtre.

En fait, les calculs du filtre se font en shared pour chaque partie de l'image que gère un stream.

La version shared + stream de Sobel est un peu différente des autres. Lors des tests nous avons constaté rendu compte qu'une petite ligne noire horizontale apparaissait au centre de l'image et avons passé beaucoup de temps à développer une implémentation capable de résoudre ce problème.

5 - Version 1D

En plus des versions originales de nos algorithmes qui utilisent, nous avons décidé de créer des versions à une seule dimension

La seule différence ici est le calcul des coordonnées du thread :

```
// Calcul des coordonnées du thread en cours d'exécution
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

La version originale en 2D permet une utilisation plus efficace de la hiérarchie de mémoire du GPU.

Lorsqu'on utilise une grille de threads 1D, il faut effectuer des calculs pour déterminer l'indice global unique pour chaque thread. Ce calcul peut prendre du temps et ajouter une surcharge inutile au traitement. En outre, il peut être plus difficile d'accéder aux données de manière cohérente dans une grille 1D, car on accède aux éléments de manière séquentielle, ce qui peut entraîner des conflits de mémoire et des temps d'attente.

En revanche, une grille de threads 2D permet d'accéder plus facilement aux données de manière cohérente et efficace. En outre, elle permet souvent d'optimiser davantage l'utilisation des tampons de mémoire partagée en permettant aux threads voisins d'accéder à des éléments adjacents, ce qui réduit les conflits de mémoire et améliore les performances globales.

6 - Résultats

Voici les résultats obtenus sur les deux images suivante :



Image des planètes
dimension: 2048 X 1536

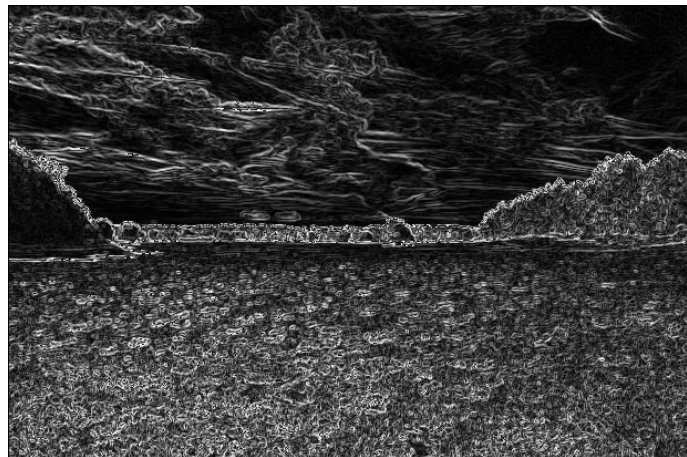


Image de la prairie
dimension: 615 x 410

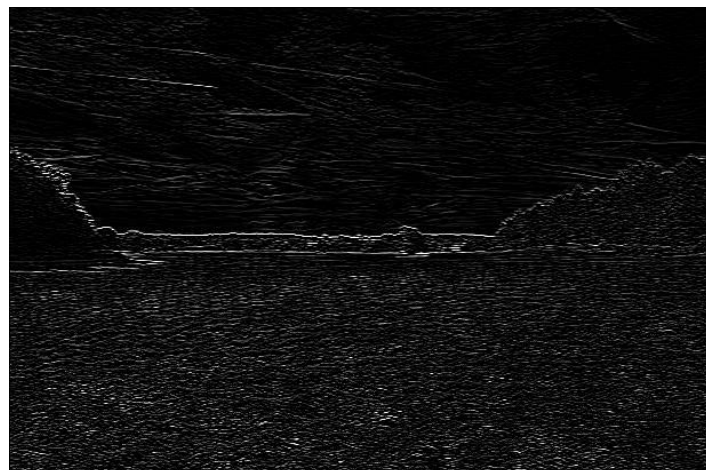
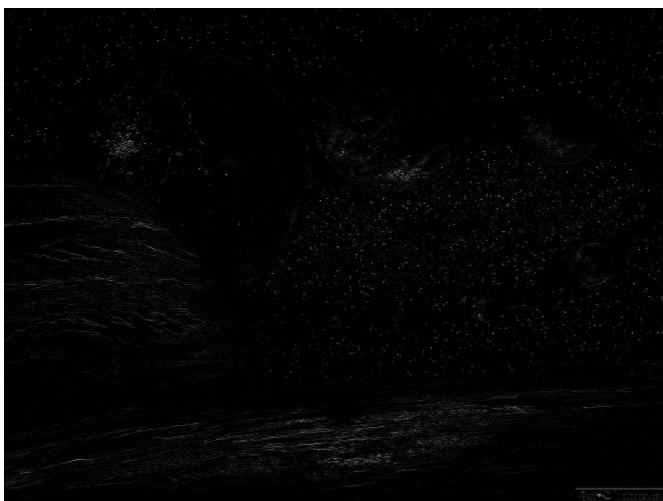
Filtre box blur :



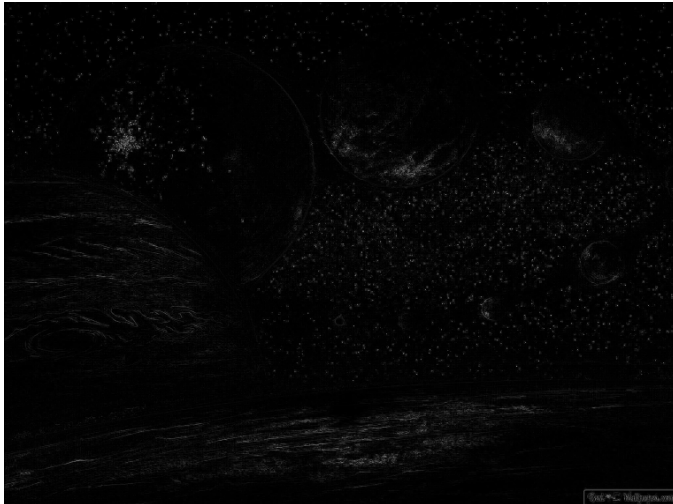
Filtre Sobel :



Filtre détection de ligne horizontale:



Filtre opérateur laplacien :



Ces deux images ayant des tailles bien différentes seront utilisées pour observer les performances.

III - Observations des performances

1 - Filtre Sobel

Déjà, nous avons pu observer une grande différence entre le modèle C++ de nos filtres et le modèle CUDA.

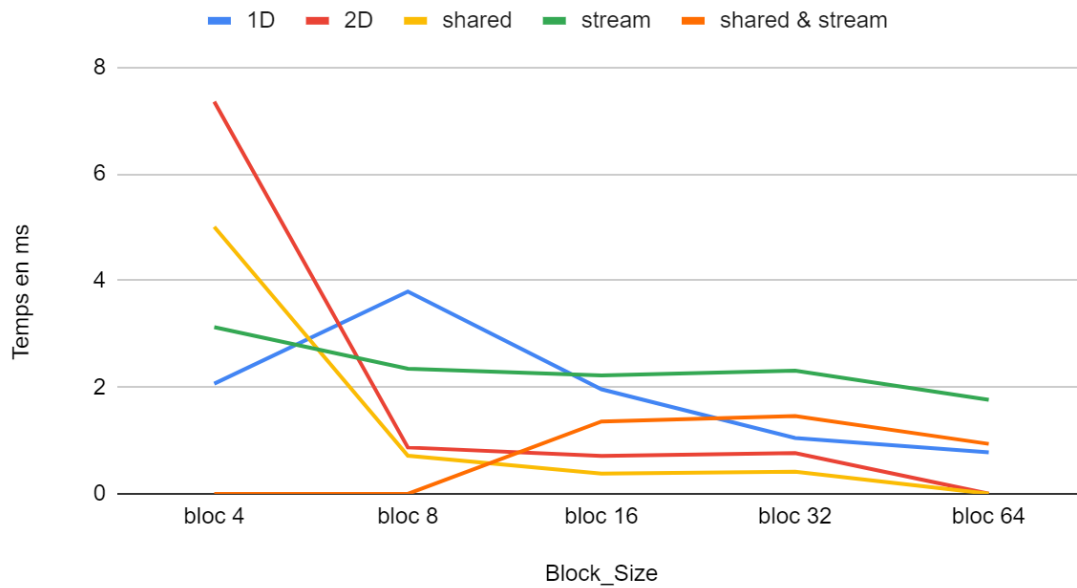
Pour le calcul de Sobel par exemple, en utilisant des blocs de taille 16 plutôt dans notre CUDA le code s'exécutant en 0,708992 ms contre 62 ms en C++ soit un temps 87 fois plus rapide avec CUDA et ceux pour tous les filtres.

Les versions CUDA sont majoritairement plus puissantes, cela est principalement dû au fait que CUDA divise la tâche entre plusieurs blocs de threads, chaque thread pouvant ainsi effectuer une opération sur un morceau de données différent, permettant d'exécuter de nombreuses opérations en parallèle.

Ici nous avons un bilan des performances sur différentes implémentations pour plusieurs tailles de blocs.

On peut néanmoins remarquer les meilleures performances quasi systématique de la version avec mémoire shared par rapport à la version simple en 2D. En effet on remarque même que l'écart se creuse lors de l'augmentation des dimensions.

Premier test cuda Sobel

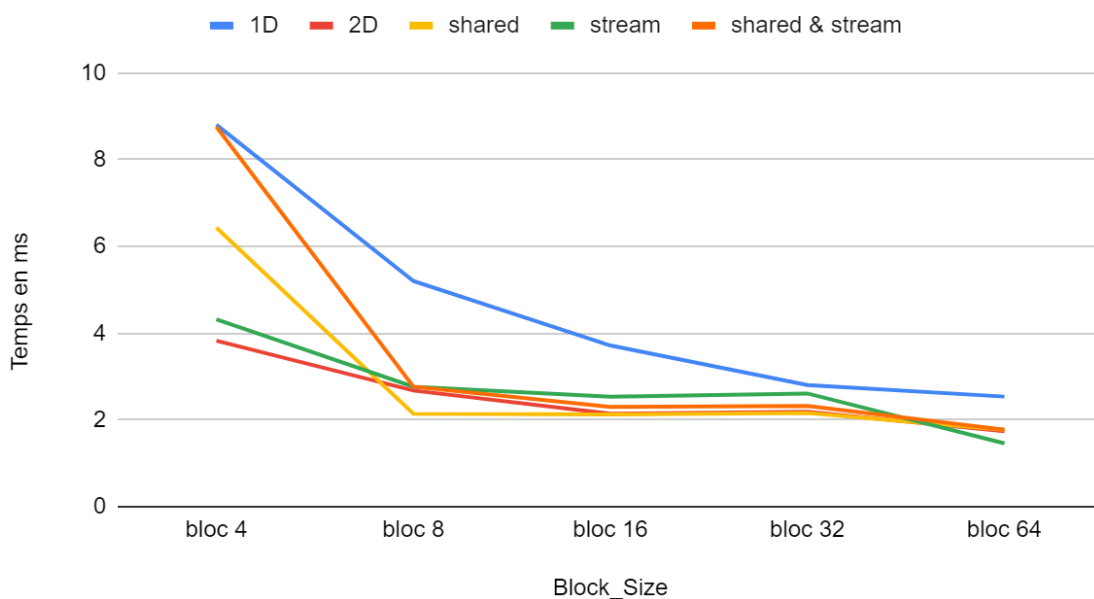


On voit ici que le temps de calcul de stream est assez mauvais en général, suivi de la version une dimension et que les meilleures versions étaient les versions 2D qui se fait détrôner par la shared.

Cependant pour cette première version nous avons exclu du calcul le moment où l'on écrit les données sur le GPU et le CPU, nous ne calculons que le temps que met CUDA à obtenir le résultat final.

Or nous nous sommes concertés et avons conclu que le processus d'écriture entre le GPU et le CPU devait être pris en compte pour obtenir des mesures réels alors voici les résultats :

Test cuda Sobel



La différence entre les différentes implémentations est beaucoup moins évidente, nous en concluons donc que l'étape demandant le plus de travail, ou du moins de temps, sont les `cudaMemcpy` et que ceci doit être optimiser au maximum.

Cependant on observe des variations beaucoup plus réalistes.

Déjà les temps de calculs convergent lorsque le nombre de blocs augmente et d'ailleurs en fonction de la taille de ce bloc on observe que la meilleure solution n'est pas toujours la même.

Lorsque la taille du bloc est très petite, on va privilégier la version 2D basique du calcul qui est bien plus rapide que les autres.

La version `shared` est celle qui s'en sort le mieux le plus souvent.

La version `shared+stream` ne semble pas apporter une réelle plus value à ce niveau même si elle semble être une très bonne alternative quand la taille des blocs devient grand.

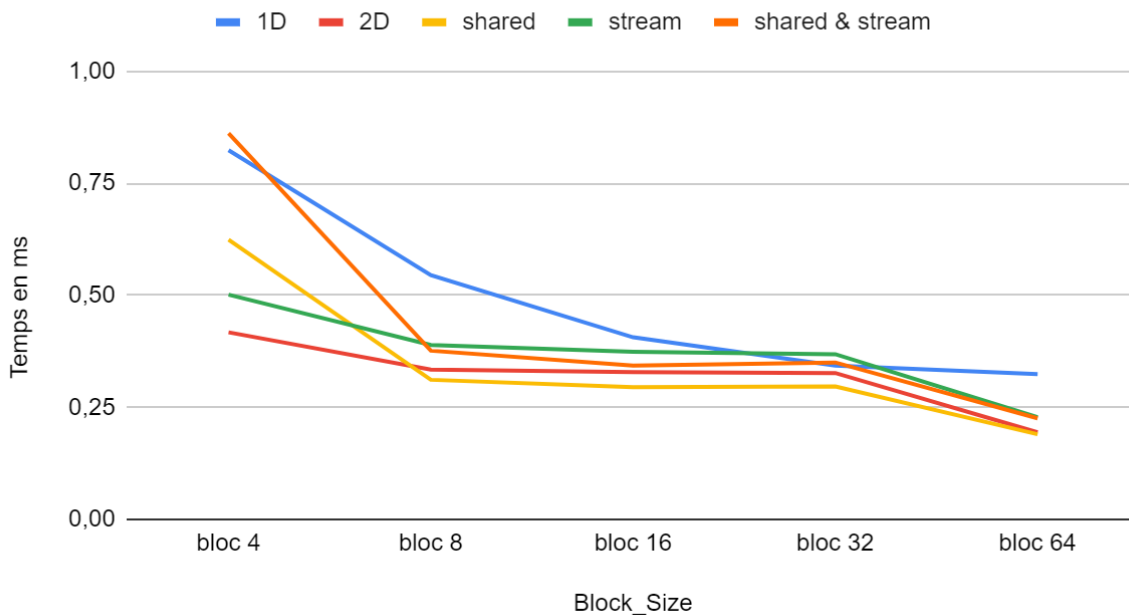
La version 1D est à la traîne par rapport aux autres.

Nous avons aussi testé l'implémentation sur une image plus petite que voici :



Là où la première faisait 2048 x 1536 pixels, celle ci n'en fait que 615 x 410, on peut voir alors une diminution globale du temps de calcul pour toutes les implémentations mais les observations faites précédemment sont les mêmes.

Test cuda Sobel sur in2.jpg



(le temps d'exécution de l'implémentation C++ est de 5 ms)

Un détail nous a alors frappé à ce moment : lorsque nous passons la taille des blocs de 32 à 64, nous n'obtenons qu'une image complètement noire en sortie.

Nous avons supposé que cela devait être dû à une limite matérielle que nous aurions dépassée. Nous avons donc voulu vérifier cela en regardant les spécifications de notre machine et supposons que si notre code utilise une quantité excessive de mémoire partagée par bloc, il peut atteindre la limite de 48 ko et provoquer ce comportement inattendu. Plus tard, en ajoutant la gestion des erreurs de CUDA, nous avons remarqué que lorsque le bloc était de 64 dans les versions 2D, cela relevait une de ces CUDA Error au niveau du Kernel qui nous dit "invalid configuration arguments".

Cela nous laisse finalement sous-entendre que soit nous avons atteint la limite de taille maximale de bloc utilisée en faisant 64x64, soit cela est bien dû à une quantité excessive de mémoire partagée utilisée.

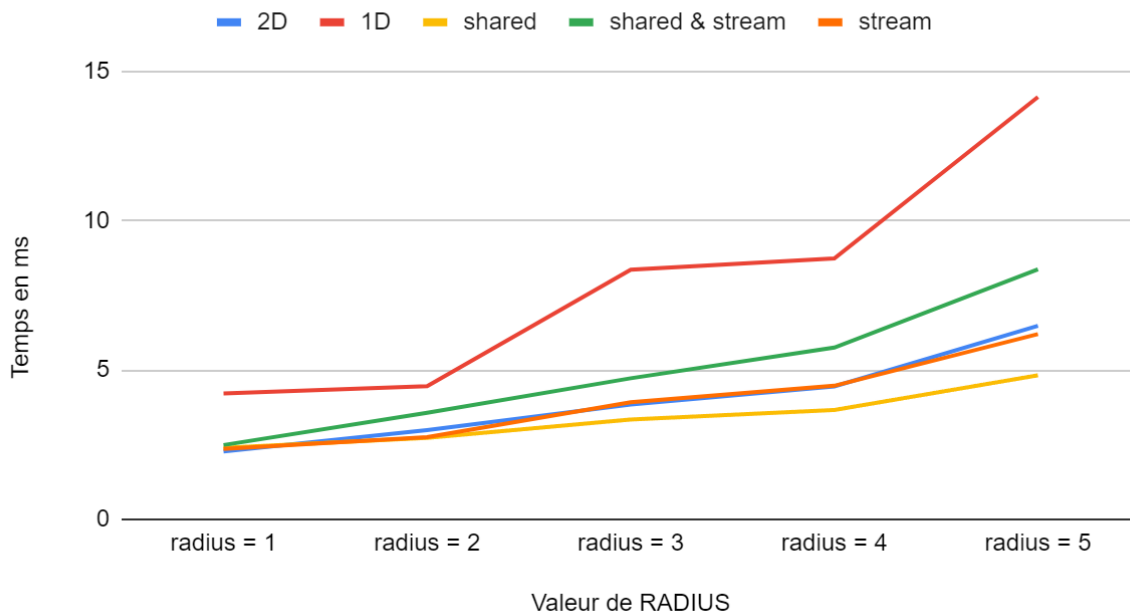
2 - Filtre Blur

Pour le blur, un détail intéressant à étudier est la taille du radius.

En effet, le radius définit le rayon du filtre de flou qui est appliqué à l'image. Plus il est grand, et plus il faudra consulter de pixels voisins pour obtenir un résultat, ce qui prolonge de manière exponentielle le temps de calcul.

Voici le résultat de quelques tests réalisés avec différentes tailles de radius sur l'image in.jpg (celle avec les planètes) :

Variation du temps en fonction de Radius pour Cuda Blur



Pour le C++, nous avons obtenu les temps : 137 ms, 344 ms, 651 ms, 1058 ms, 1592 ms. Toute cette partie du test a été effectuée avec une taille de bloc de 32, et si l'on peut observer directement que le temps augmente en même temps que le radius, la version la plus efficace pour le blur est la version shared.

La version 2D et stream sont très similaires mais on peut voir de manière globale que les différentes versions ne sont pas autant affectées par l'augmentation de cette valeur que d'autres.

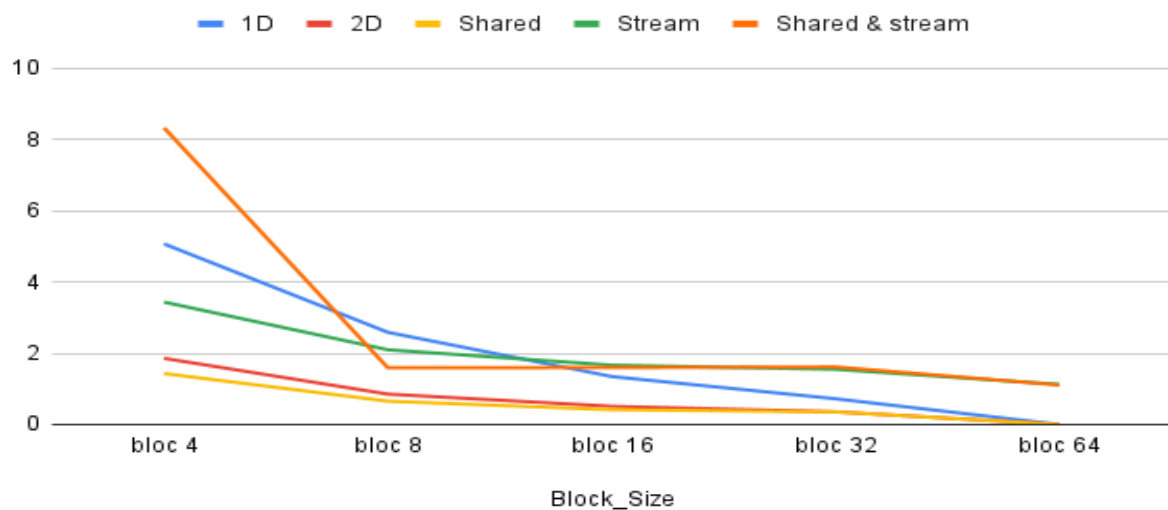
3 - Filtre Horizontal-line detection

Pour le filtre horizontal-line détection, nous avons pour la version C++:

- pour l'image planètes : 60 ms
- pour l'image prairie: 8ms

Maintenant examinons attentivement les résultats avec CUDA pour l'image de la planète avec un graphique:

test Cuda horizontal line detection in.jpg



Nous pouvons observer que Cuda est 7 fois plus rapide que la version C++, avec la version shared_stream en blocksize de 4 avec un temps de 8.23 ms

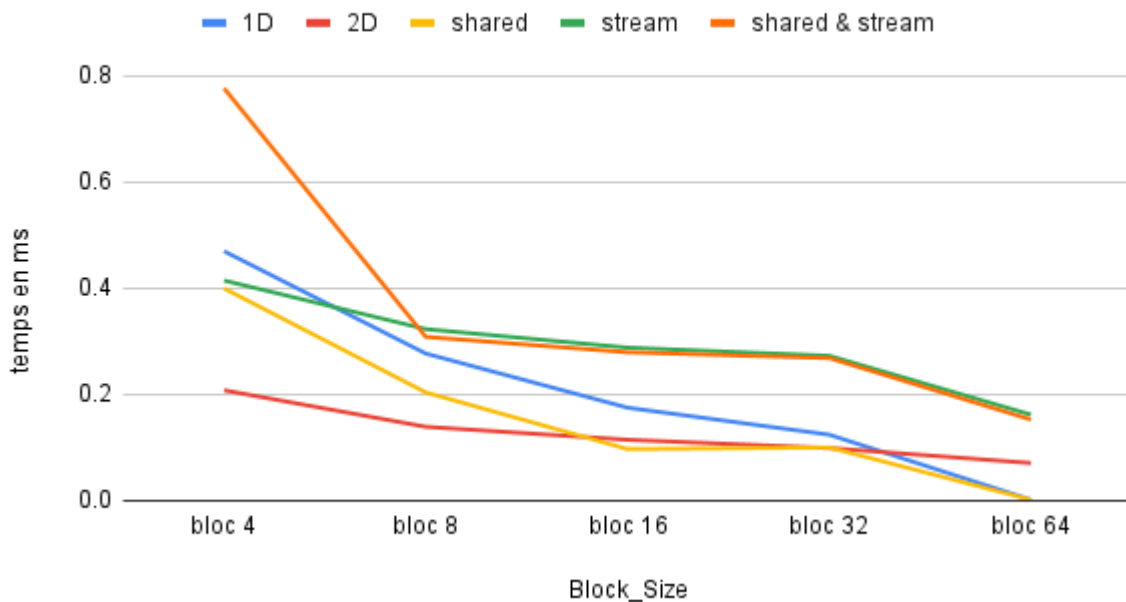
A mesure que la taille des blocs passe de 4 à 64, il existe une tendance à la diminution du temps d'exécution pour toutes les configurations. Cela signifie que les blocs plus grands fonctionnent mieux pour ces configurations CUDA(1D, 2D, shared, stream ...).

On peut observer du graph la variation des temps d'exécution dans différentes configurations (1D, 2D, shared, stream, shared et stream) pour chaque taille de bloc. Notamment, le processus Shared & Stream a toujours le temps d'exécution le plus élevé, ce qui suggère que l'utilisation combinée de la mémoire partagée et stream CUDA peut entraîner une surcharge supplémentaire dans ce scénario.

D'après les données présentées, il semble que pour les petites tailles de bloc (par exemple, 4 et 8), l'algorithme 2D fonctionne légèrement mieux par rapport aux autres algorithmes. Cependant, à mesure que la taille des blocs augmente, les différences entre les schémas deviennent moins importantes et certains schémas (par exemple, shared, stream) surpassent même le schéma 2D en termes de performances.

En comparant les temps d'exécution entre différentes tailles de bloc, nous pouvons observer que le taux de croissance diminue à mesure que la taille du bloc augmente. Par exemple, la réduction du temps de traitement de la taille de bloc 4 à 8 est importante, mais la réduction ultérieure devient plus petite pour les tailles de bloc plus grandes.

test Cuda horizontal line detection in2.jpg



Nous pouvons observer que Cuda est 12 fois plus rapide que la version CPP, avec la version `shared_stream` en blocksize de 4 avec un temps de 0.77648 ms

Comme la résolution de l'image est inférieure à celle de l'image d'origine, on peut constater qu'elle présente un temps d'exécution plus court.

Parmi les configurations, la configuration 1D affiche systématiquement le temps de traitement le plus élevé pour toutes les tailles de bloc. En revanche, la taille de 64 blocs du système Stream présente le temps d'exécution le plus faible. Les configurations Shared et Shared & Stream affichent également des performances améliorées par rapport à la configuration 1D mais ont généralement des temps d'exécution légèrement plus élevés par rapport à la configuration Stream.

différences de temps de traitement pour différentes tailles de bloc dans la même configuration. Par exemple, dans la configuration 1D, la consommation de temps diminue de manière significative à partir d'une taille de bloc de 4 à 8, mais la réduction ultérieure devient plus petite. De même, dans les configurations Shared et Shared & Stream, les temps d'exécution restent relativement stables avec des tailles de bloc différentes.

Vous pouvez retrouver les 2 tableaux qui ont permis de faire le graphe en Annexe.

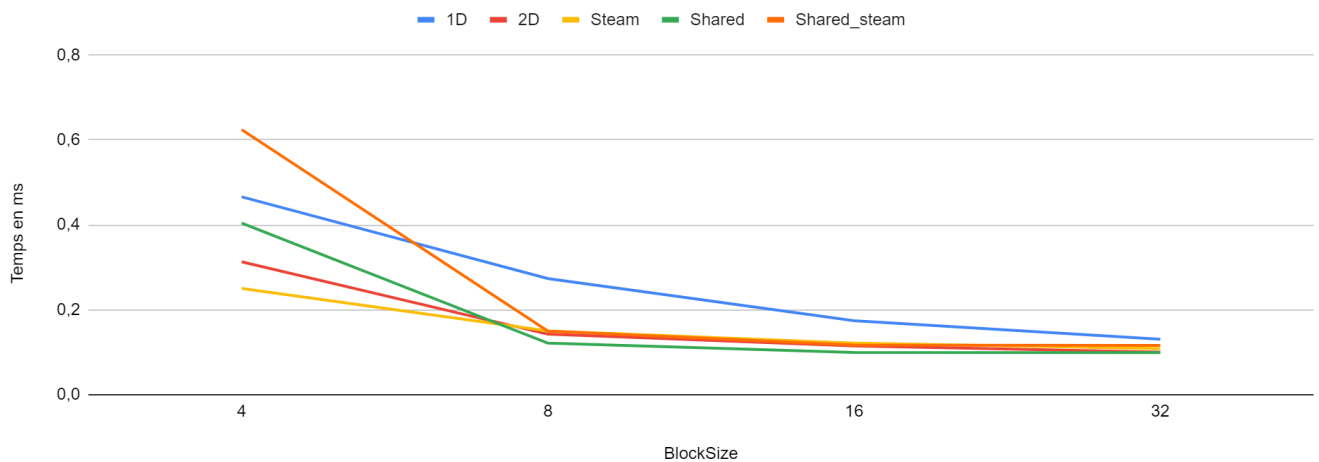
4 - Filtre Opérateur de Laplace

Pour le filtre Opérateur de Laplace, nous avons pour la version C++

- image prairie: 9 ms
- image des planètes: 114 ms

Maintenant regardons attentivement les résultats avec CUDA pour l'image de la prairie avec un graphique:

Performance filtre Laplace par rapport au différentes optimisations de l'image de la prairie



Nous pouvons observer que le temps pour l'ensemble des différentes optimisations se réduit avec l'augmentation des tailles des blocs. Lorsque nous sommes à 32 soit dim3 block (32) pour 1D et dim3 block (32,32) pour le reste on tourne aux alentours de 0.1 ms de manière générale.

Pour continuer, nous remarquons que pour des tailles de bloc de 32, la configuration 1D offre le temps le moins performant et la configuration shared est la meilleure pour un blocksize de 8 et 16.

Pour 2D et stream, nous remarquons que stream est meilleur pour les blocksize de 4, mais ils sont presque équivalents pour le reste soit 8,16 et 32.

On pourrait croire que c'est shared_stream qui serait le plus performant car il combine les optimisations de shared et de stream, mais non ! On remarque que c'est lui qui est le plus lent dans son exécution avec le blocksize de 4 mais il se rapproche de stream et de la version 2D quand la taille des blocs passe à 8,16 et 32 même si c'est lui qui a les résultats les moins bons entre les 3. Ainsi on remarque une chute entre les tailles de blocs de 4 et 8 pour shared_stream.

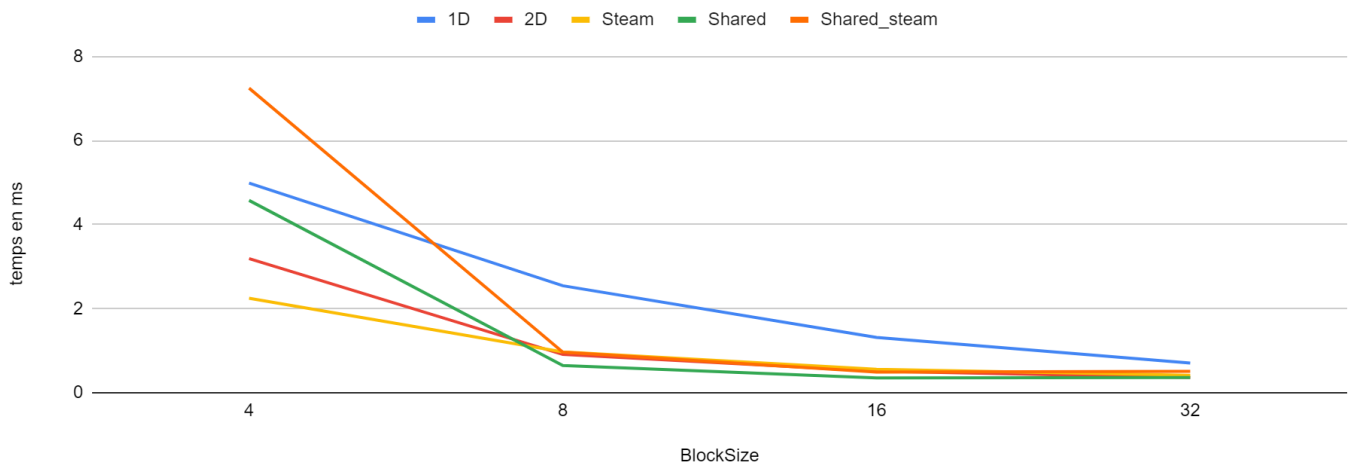
Pour conclure les performances sur l'image de la prairie, la version CUDA en 1D est la moins performante tandis que la meilleure reste la shared.

Avec ce filtre et la configuration du serveur actuel, la version combinant shared et stream n'est pas la plus adaptée même si c'est la combinaison de deux optimisations. Mais peut-être qu'avec des images plus grandes et avec une configuration matérielle différente celui-ci aurait pu être le plus performant.

Mais nous pouvons aussi voir que l'exécution Cuda est plus rapide que celle en C++. Cuda pour l'image de la prairie est 15 fois plus rapide avec shared_stream en blocksize de 4 que la version C++.

Maintenant regardons les performances pour l'image avec les planètes:

Performance Filtre Laplace par rapport aux différentes optimisations pour l'image des planètes



Nous pouvons conclure la même chose pour les différentes optimisations en termes de performance et que Cuda est 16 fois plus rapide que C++ avec sa version `shared_stream` en blocksize de 4.

Vous pouvez retrouver les 2 tableaux qui ont permis de faire le graphe en Annexe.

Pour terminer, nous voyons qu'il y a un impact sur les performances lorsque l'image est plus grande. On remarque qu'en `shared_stream` l'image de la prairie est 11 fois plus rapide avec les blocksize de 4 que l'image avec les planètes qui sont les temps d'exécutions les plus mauvaises pour les 2 images. Puis en `shared` avec un blocksize de 32 qui est le temps d'exécution le plus rapide pour les deux image, on remarque que l'image de la prairie est 7 fois plus rapide que l'image avec les planètes.

IV - Bilan

Pour avoir tester les différentes versions des filtres sur différentes images, la version `shared` semble être la plus performante de manière générale même si c'est cette dernière qui nous aura demandé le plus de travail pour l'implémentation. De plus, nous pouvons ajouter que la taille de l'image a un impact sur les performances.

Et la version 1D semble toujours être le mauvais choix quand il s'agit d'utiliser CUDA.

En tout cas, CUDA est un outil puissant et cela se voit bien lorsque l'on compare les temps entre le C++ et n'importe quelle implémentation.

V - Annexe

1 - Spécification de la machine

```
erwan.carneiro@haswell-cuda:~/ProjetCuda2023$  
Number of devices: 1  
Device Number: 0  
Device name: GeForce GTX 780  
Memory Clock Rate (MHz): 2933  
Memory Bus Width (bits): 384  
Peak Memory Bandwidth (GB/s): 288.4  
Total global memory (Gbytes) 3.0  
Shared memory per block (Kbytes) 48.0  
minor-major: 5-3  
Warp-size: 32  
Concurrent kernels: yes  
Concurrent computation/communication: yes
```

2 - Calcul du temps de Sobel sans prendre en compte l'écriture des données entre le GPU et le CPU

SOBEL	bloc 4	bloc 8	bloc 16	bloc 32	bloc 64
1D	2,06605	3,79485	1,96038	1,04643	0,776448
2D	7,35722	0,869152	0,708992	0,76288	0,00176
shared	5,0071	0,713216	0,379328	0,412928	0,001792
stream	3,12512	2,34493	2,21901	2,31024	1,76416
shared & stream	0	0	1,35619	1,45738	0,937152

3 - Temps de la version finale de Sobel avec image in.jpg

SOBEL	bloc 4	bloc 8	bloc 16	bloc 32	bloc 64
1D	8,80429	5,20163	3,71562	2,80467	2,53792
2D	3,82509	2,67738	2,14253	2,18899	1,74125
shared	6,43299	2,14349	2,12534	2,15952	1,77587
stream	4,32106	2,76499	2,53722	2,61014	1,45888
shared & stream	8,75846	2,76672	2,3001	2,324	1,77619

4 - Temps de la version finale de Sobel avec image in.jpg

SOBEL	bloc 4	bloc 8	bloc 16	bloc 32	bloc 64
1D	0,823616	0,545056	0,406304	0,34336	0,32416
2D	0,417376	0,334112	0,3288	0,326528	0,194176

shared	0,623936	0,311456	0,294752	0,29664	0,189856
stream	0,501312	0,38896	0,374176	0,368416	0,227744
shared & stream	0,861472	0,376416	0,342944	0,349984	0,225536

5 - Variation du temps en fonction de Radius pour Cuda Blur

Valeur de RADIUS	radius = 1	radius = 2	radius = 3	radius = 4	radius = 5
2D	2,2735	2,98464	3,84547	4,45267	6,47974
1D	4,21744	4,45642	8,35843	8,73242	14,1372
shared	2,39485	2,72758	3,34205	3,66099	4,8217
shared & stream	2,4872	3,56899	4,72243	5,74867	8,36605
stream	2,36096	2,7495	3,92234	4,47238	6,20109

6 - Temps de la version finale de Laplace avec image in.jpg

laplace planete	4	8	16	32
1D	4,98826	2,54544	1,31539	0,7064
2D	3,18989	0,910432	0,51472	0,359392
Steam	2,24624	0,96912	0,55968	0,414592
Shared	4,57568	0,648032	0,351392	0,35984
Shared_steam	7,2465	0,967168	0,489792	0,507328

7 - Temps de la version finale de Laplace avec image in2.jpg

laplace prairie	4	8	16	32
1D	0,465984	0,273632	0,174176	0,130752
2D	0,312896	0,1424	0,114912	0,100416
Steam	0,25024	0,150176	0,121504	0,10864
Shared	0,40416	0,121376	0,099424	0,099168
Shared_steam	0,624384	0,149792	0,116864	0,116032

8 - Temps de la version finale horizontal line detection avec image in.jpg

Block_Size	4	8	16	32	64
1D	5.07034	2.58819	1.34646	0.722816	0.001792
2D	1.85808	0.853504	0.510848	0.356416	0.00176
Shared	1.43094	0.65492	0.415904	0.354944	0.00176
Stream	3.4367	2.09779	1.66598	1.54941	1.12557
Shared & stream	8.32893	1.59171	1.59942	1.61616	1.10854

9 - Temps de la version finale horizontal line detection avec image in2.jpg

Block_Size	4	8	16	32	64
1D	0.46976	0.276992	0.175104	0.124256	0.001632
2D	0.207904	0.139136	0.11472	0.09904	0.07088
shared	0.399648	0.20384	0.097152	0.10016	0.00176
stream	0.41456	0.32304	0.288192	0.2728	0.161952
shared & stream	0.77648	0.308448	0.279488	0.268544	0.152736