Rapport CUDA - Matrice de convolution

Groupe : Thomas BERTHELOT, Simon CAVILLON, Antoine VACHER

Dans ce rapport, nous allons présenter les 4 filtres que nous avons implémentés en séquentiel, puis en CUDA simple et enfin en y ajoutant des optimisations.

Nous allons commencer par présenter les différents algorithmes, puis nous allons ensuite parler des différentes optimisations que nous avons réalisées en analysant les performances de celles-ci, pour enfin finir sur les problèmes que nous avons rencontrés durant notre développement et sur une conclusion de ce projet.

Sommaire :

I) Présentation des différents filtres réalisés

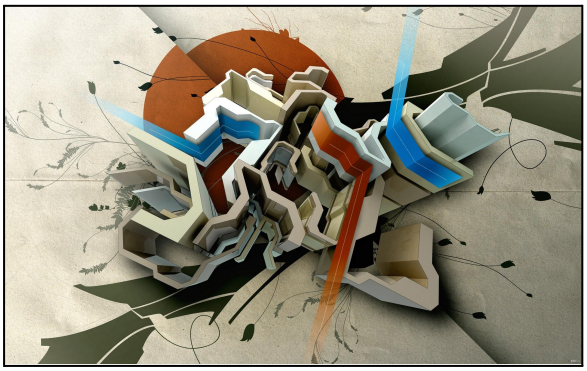
Nous avons développé et optimisé 4 filtres de convolution. La convolution est une opération de traitement d'images qui consiste à appliquer une opération à chaque pixel de l'image, cette opération prend aussi en compte le voisinage de ce pixel. Cette opération est souvent décrite par une matrice de coefficients qu'il faut appliquer à l'image entière.

Pour notre part, nous avons implémenté ces 4 filtres :

- Edge detection
- Sobel edge operator
- Laplacian of Gaussian
- Gaussian blur

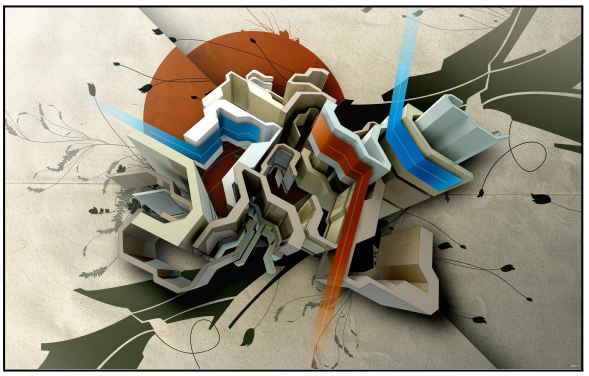
Voici pour chaque algorithme, un exemple d'exécution sur une image (image d'exemple utilisée durant les TP) ainsi que la matrice de convolution utilisée par l'algorithme (vu sur le site donnée en ressource du sujet https://www.aishack.in/tutorials/image-convolution-examples/) :

Edge detection :



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

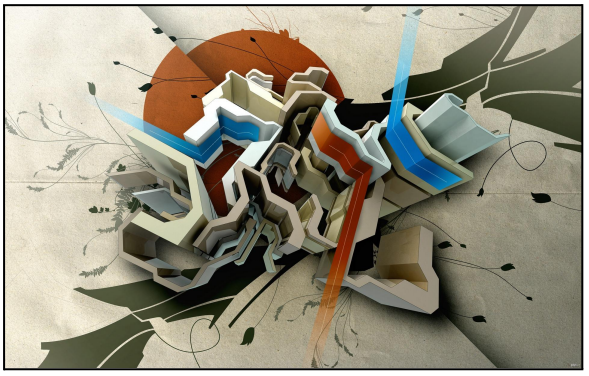
Sobel edge operator :



-1	-2	-1		-1	0	1
0	0	0		-2	0	2
1	2	1		-1	0	1

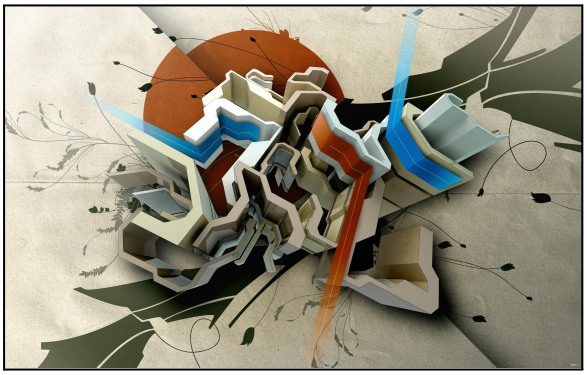
Matrice de convolution horizontale et verticale

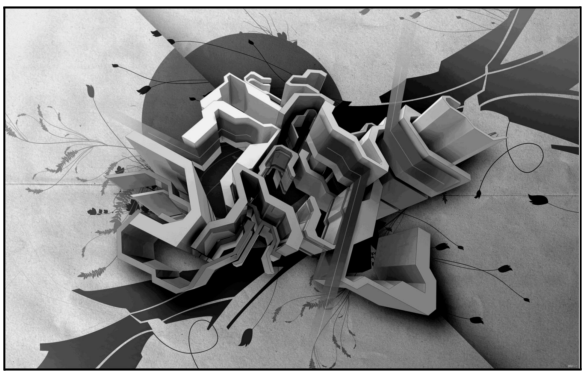
Laplacian of Gaussian :



0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

Gaussian blur :





0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

Pour ces 4 algorithmes, nous utilisons donc deux kernels, *grayscale* et *edges.* Le kernel grayscale permet d'appliquer un grayscale sur l'image donnée et le kernel edges va appliquer la matrice de convolution à l'image.
Le seul algorithme qui change un peu des autres est le gaussian blur puisque nous faisons une normalisation de la matrice de convolution avant de l'appliquer sur l'image. Pour le reste, l'exécution est la même pour ces 4 algorithmes.

II) Analyse des performances.

Matériel et données :

Afin d'avoir des résultats d'analyse plus précis, les temps d'exécution sont basés sur la moyenne de 150 itérations. Tous les résultats peuvent être retrouvés à ce lien.

Les valeurs ont été obtenues sur une machine décrite ci-dessous, sous Ubuntu 22.04:

```
CPU:
Model name:          AMD Ryzen 7 5800X 8-Core Processor
   CPU family:       25
   Model:            33
   Thread(s) per core:  2
   Core(s) per socket:  8
   Socket(s):        1
   CPU max MHz:      4850,1948
   CPU min MHz:      2200,0000

GPUs:
Number of devices: 1
Device Number: 0
   Device name: NVIDIA GeForce RTX 3060 Ti
   Memory Clock Rate (MHz): 6836
   Memory Bus Width (bits): 256
   Peak Memory Bandwidth (GB/s): 448.1
   Total global memory (Gbytes) 7.8
   Shared memory per block (Kbytes) 48.0
   minor-major: 6-8
   Warp-size: 32
   Concurrent kernels: yes
   Concurrent computation/communication: yes
   Max threads / block: 1024
   Max block dimension: 1024,1024,64
```

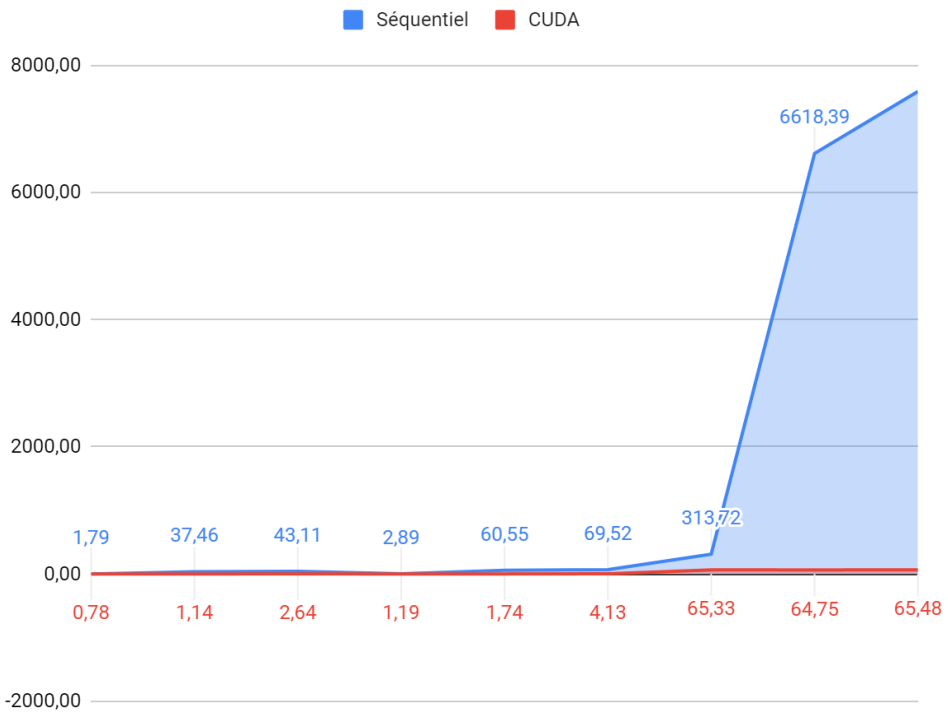
Description des algorithmes :

Comme il est possible de le constater dans les fichiers cpp, l'algorithme d'application de matrices de convolution repose toujours sur le même principe : un pixel va être traité indépendamment des autres (sauf cas particuliers tels que la mémoire partagée). Le but étant ici d'avoir un code générique qui permet de réussir à appliquer une matrice quelconque et de taille variable sur l'image. C'est avec cet état d'esprit que les scripts cpp sont construits.

Pour les tests, nous pouvons donc regrouper en 3 catégories les matrices, 3x3, 5x5, 7x7. Qui seront elles aussi regroupées en fonction de la taille de l'image d'entrée afin d'avoir des résultats homogènes et valides.

Séquentiel vs. CUDA :

L'objectif ici va être de déterminer l'impact de l'efficacité de travail entre GPU et CPU, le même algorithme va alors être exécuté, d'un côté, le CPU va traiter chaque pixel à la suite, de l'autre le GPU va traiter chaque pixel en parallèle.



Temps d'exécution en ms. (le plus bas est le mieux)

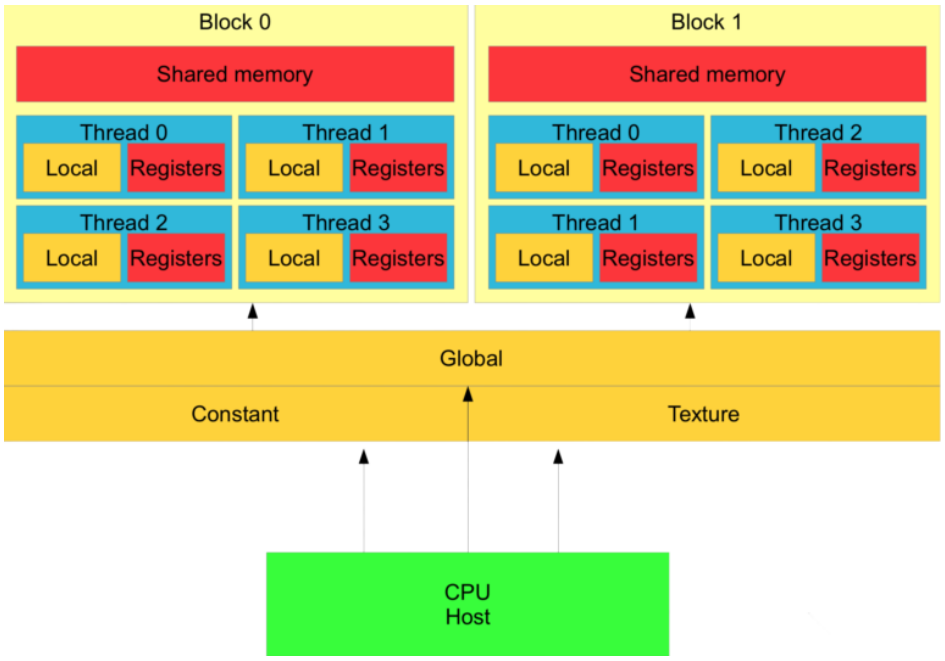
MATRICE	SEQ.	CUDA
Image: 768 x 768		
3x3	1,79	0,78
5x5	37,46	1,14
7x7	43,11	2,64
Image: 1280 x 721		
3x3	2,89	1,19
5x5	60,55	1,74
7x7	69,52	4,13
Image: 14400 x 7200		
3x3	313,72	65,33
5x5	6618,39	64,75
7x7	7593,04	65,48

Temps d'exécution en ms.

Comme il est possible de le constater, il y a un réel avantage à effectuer ces opérations avec CUDA et non pas en séquentiel, le fait est qu'il existe de très nombreux threads sur le GPU qui peuvent tous effectuer de très nombreuses convolutions en simultanées. Sur notre machine par exemple, nous avons un CPU à 16 threads, ce qui nous dit qu'il est possible de faire 16 pixels en simultanés. Ce n'est rien comparé à notre GPU qui, si on regarde la description technique, nous dit qu'il y a au minimum 1024 threads pour un unique bloc. Chaque thread CUDA faisant le même calcul qu'un thread CPU. De ce fait, nous allons **64 fois plus vite** avec l'utilisation de CUDA.

CUDA et optimisations (Shared Memory) :

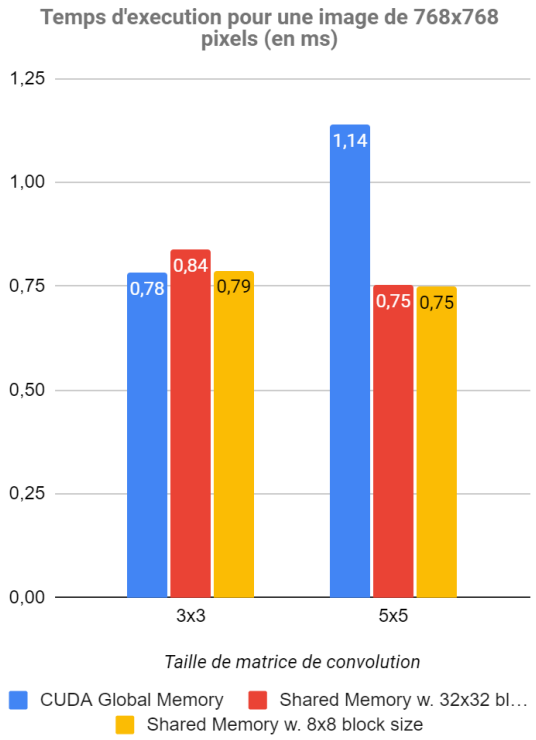
Les résultats obtenus au-dessus représentent le temps selon un algorithme simple qui répartit la charge de travail sur les différents blocs et différents threads. Néanmoins, la mémoire utilisée est la mémoire globale, il s'agit d'une des mémoires les plus lentes de la carte graphique, certes l'espace disponible est plus élevé mais le temps de transition des données est plus élevé. Afin de pallier ce problème, une première optimisation est possible, celle de la répartition de la mémoire.

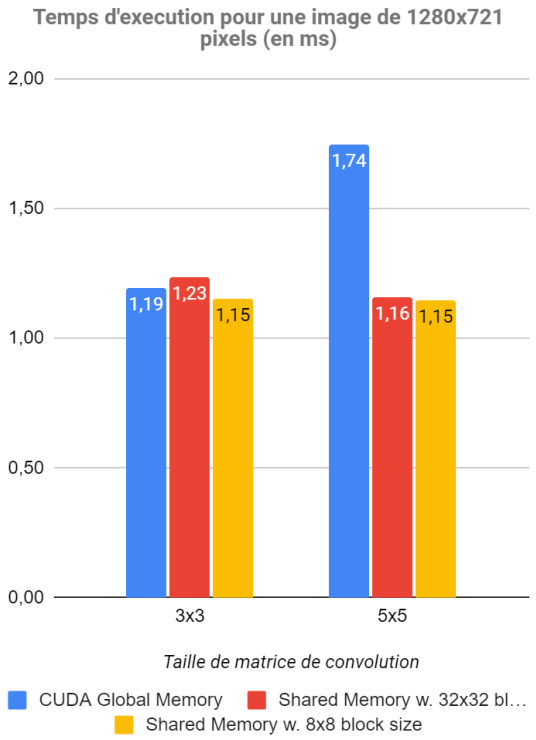


Source: http://thebeardsage.com/cuda-memory-hierarchy/

Nous pouvons voir sur le diagramme ci-dessus la représentation de la hiérarchie de mémoire sur un GPU, jusqu'à présent la mémoire utilisée est celle nommée **Global**, commune à tout le GPU.
Plus une mémoire est loin d'un thread, plus son parcours à une durée élevée. L'objectif ici est donc de limiter le plus possible les interactions entre les threads et les mémoires "distantes". La première optimisation est donc de distribuer la mémoire sur ce qui s'appelle le **Shared Memory** qui se situe à une distance de moins que la mémoire globale et est au sein même d'un bloc.

Regardons à présent quelques résultats avec et sans Shared Memory :





MATRICE	CUDA GLOBAL	SM. 32x32	SM. 8x8
Image: 768 x 768			
3x3	0,78	0,84	0,79
5x5	1,14	0,75	0,75
Image: 1280 x 721			
3x3	1,19	1,23	1,15
5x5	1,74	1,16	1,15

Comme il est possible de le voir, le temps d'exécution va plus dépendre de la dimension de la matrice de convolution que de la dimension de l'image, cela est expliqué par le fait que pour chaque pixel de l'image, il y a un total d'opérations de lectures en mémoire globale de :

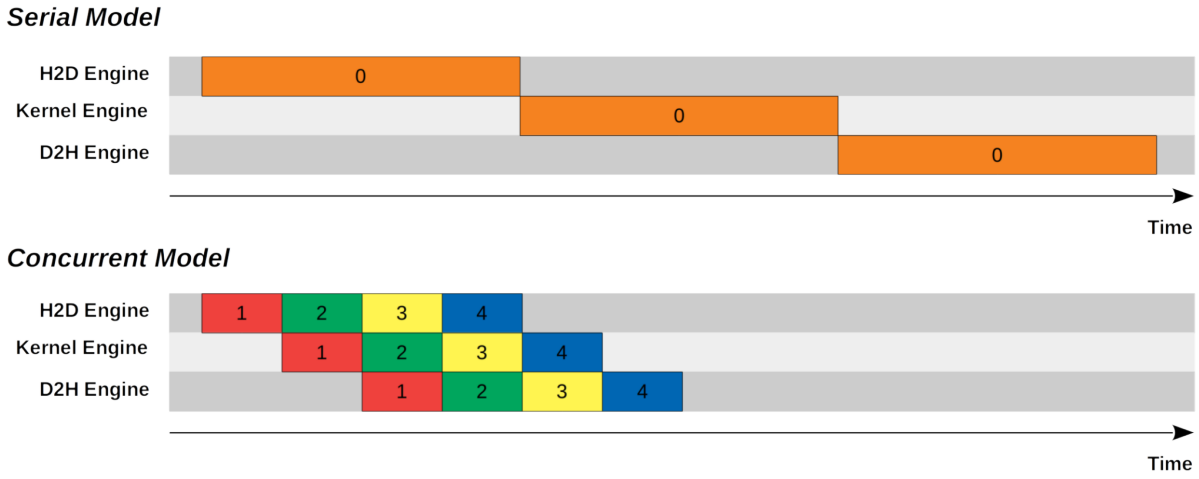
$$m \times n$$

où m est le nombre de lignes de la matrice de convolution et n son nombre de colonnes.

Cette première observation permet d'en conclure que la mémoire partagée permet d'avoir plus d'avantages sur un algorithme tel que Gaussian Blur qui dispose d'une matrice de taille 7x7, et qui profitera largement de la réduction de temps de récupération des données.

Une autre approche (Streams) :

Une autre manière d'optimiser le flux de données à traiter sur les GPU est de créer des Streams, ceux-ci permettent de subdiviser la charge de travail afin de faire une file en concurrence, une fois que la file à terminé sa charge de travail elle remplace donc le sous-groupe de données à son emplacement.



Source: https://leimao.github.io/blog/CUDA-Stream/

Le diagramme ci-dessus nous montre le déroulement du traitement de données par un GPU classique. D'une part, nous copions les données depuis le CPU vers le GPU (H2D), puis le kernel s'exécute avec les données (Kernel), et pour finir les données sont renvoyées vers le CPU (D2H).

En utilisant des streams il est possible de voir que tout est effectué en concurrence, c'est-à-dire que pour chaque stream, les données vont être envoyées, puis le kernel va être lancé et les données vont être récupérées. Cependant, pour les streams, lorsqu'un kernel va être lancé, pendant son exécution, une autre donnée va être copiée pour un autre kernel, cette opération en parallèle permet le travail en concurrence des données.

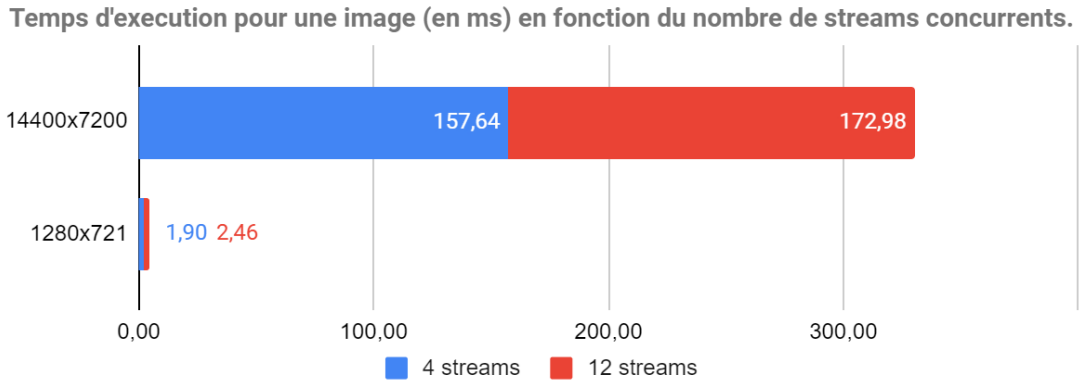
Il ne faut pas oublier que les streams doivent être initialisés par le CPU, et pour nous, c'est ce qui a fait que les streams ne sont pas une optimisation viable. De plus, il est impératif pour un résultat optimal propre de jouer avec les "frontières" des streams pour avoir une image correcte, sinon des lignes blanches se forment au niveau des limites des streams.

Malgré que cette optimisation n'a pas été retenue pour nous, voici un jeu de données comparatifs :

MATRICE	SEQ.	CUDA	STREAM
Image: 768 x 768			
3x3	1,79	0,78	1,02
5x5	37,46	1,14	1,35
7x7	43,11	2,64	2,82
Image: 1280 x 721			
3x3	2,89	1,19	1,90
5x5	60,55	1,74	3,19
7x7	69,52	4,13	4,41
			Temps d'exécution en ms.

Il est possible de voir un graphique comparatif plus tard dans le rapport qui permettra de comparer toutes les données avec les streams. En attendant nous avons aussi exploré plusieurs possibilités en lien avec les streams. Notamment le nombre de streams, y à t-il une importance au nombre de streams et ce nombre fait-il une différence au sein de notre algorithme ?

Pour comparer et observer cette possible différence, nous avons fait quelques tests sur les images les plus grandes afin d'avoir un résultat plus "flagrant". Ce test consiste juste à changer le nombre de streams concurrent, de cette manière nous parvenons à vraiment isoler si un quelconque changement est provoqué par la quantité.

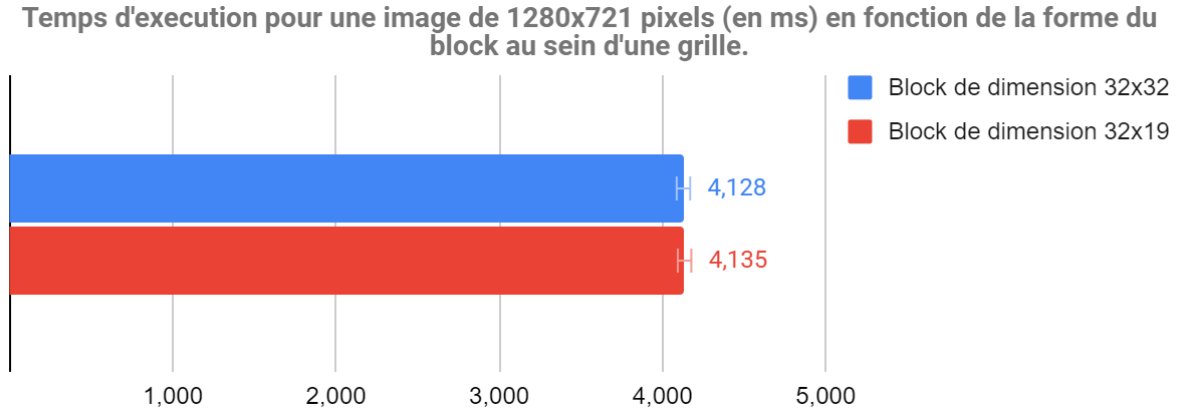


Nous pouvons voir que plus le nombre est élevé, plus le temps d'exécution est lui aussi élevé. De quoi cela peut venir ? Nous supposons qu'ici le temps supplémentaire est à cause de l'initialisation des streams sur CPU.
Le fait d'avoir plus de streams perd donc du sens puisqu'il nous fait perdre du temps et dans les deux cas, le temps d'exécution sera forcément plus élevé.

Note importante
Le fait d'avoir de nombreux streams peut avoir un intérêt, étant donné que cela permet de travailler sur un sous-groupe de données. Il est parfaitement envisageable d'allier CUDA Streams avec une librairie de parallélisation comme MPI afin de générer un stream par thread CPU, ce qui rendrait les streams très efficaces en théorie.

Une grille plus probante ? :

Nous avons essayé de voir si une taille de bloc allait faire varier l'efficacité de notre algorithme, puisque nous avons des images de différents types (carré et paysages) nous avons exploré l'hypothèse du ratio de la taille d'un bloc. Cette idée est venue en partant du postulat que nous pouvons essayer de faire une grille qui n'aura pas d'overflow (dépassement de l'image). Un bloc rectangulaire sera donc peut-être plus probant sur une image paysage et un bloc carré sur une image du même ratio, c'est-à-dire 1:1.



Après quelques observations, nous pouvons voir que ce n'est pas le cas, en indiquant même une marge d'erreur de 1 %, la différence entre un bloc carré et rectangulaire ne dépasse pas cette marge. Nous pouvons donc en conclure avec une assez grande certitude que la dimension du bloc n'influe pas sur l'efficacité de notre algorithme.

III) Les difficultés rencontrées

Avec les nombreuses itérations de notre code s'accompagnent des difficultés, la majorité est due à la gestion de la mémoire et les overflows. En effet, certains de nos algorithmes ne s'exécutent pas en fonction de la taille de la matrice ou alors de la taille de l'image. À cause de cela, nous n'avons pas pu tester toutes les optimisations sur tous les jeux de données, mais cela nous a quand même permis d'avoir assez de données afin de proposer des réponses viables face à notre algorithme et nos différentes optimisations.

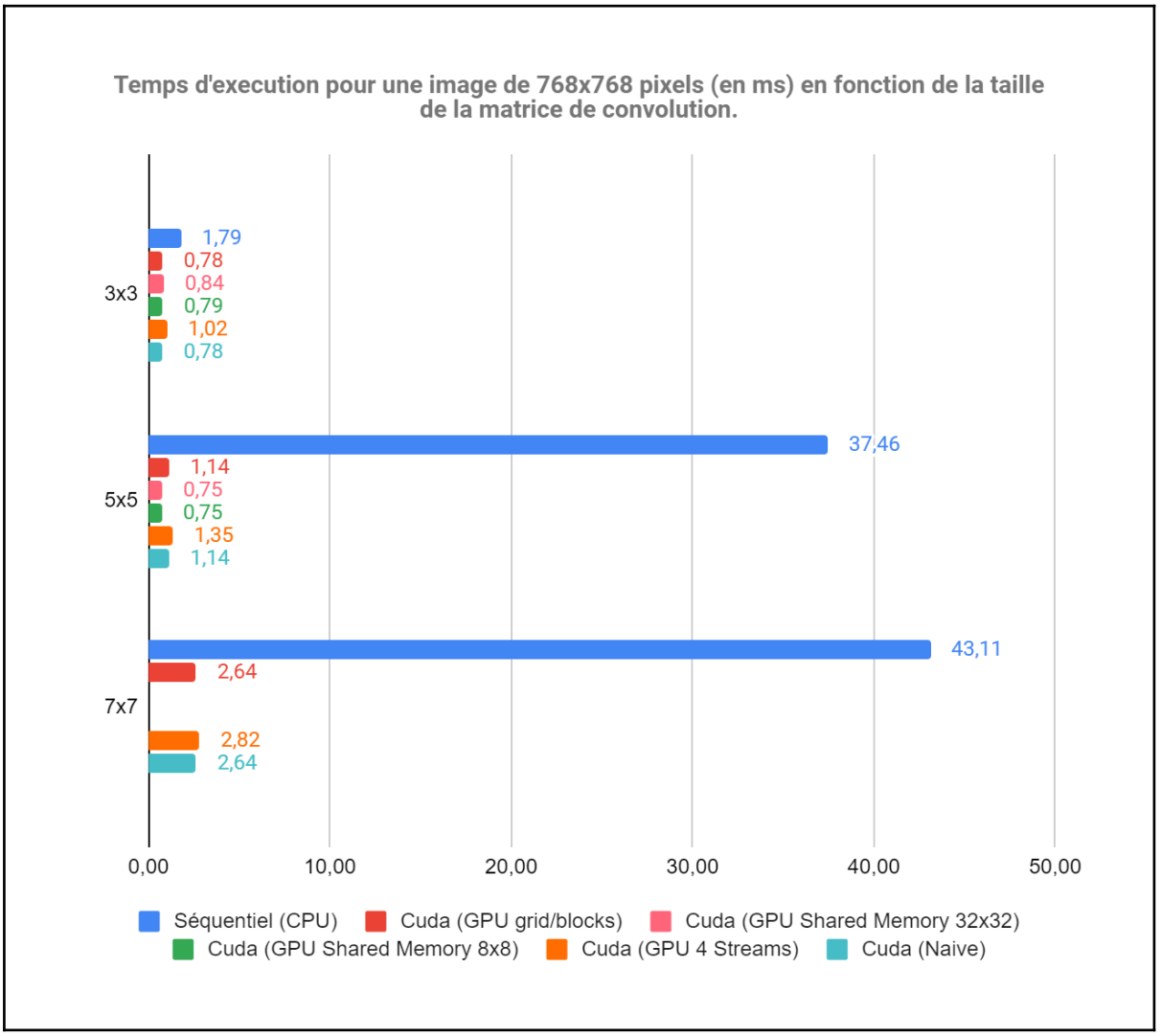
Il est possible de voir là où notre algorithme ne s'exécute pas sur le Spreadsheet des données, les colonnes mises en rouge représentent les données que nous n'avons pas pu récupérer pour cause d'exception dans notre code. Il est aussi possible de voir tous les jeux de données dans leur intégralité, séparés par catégorie, que ce soit par taille d'image ou par taille de matrice.

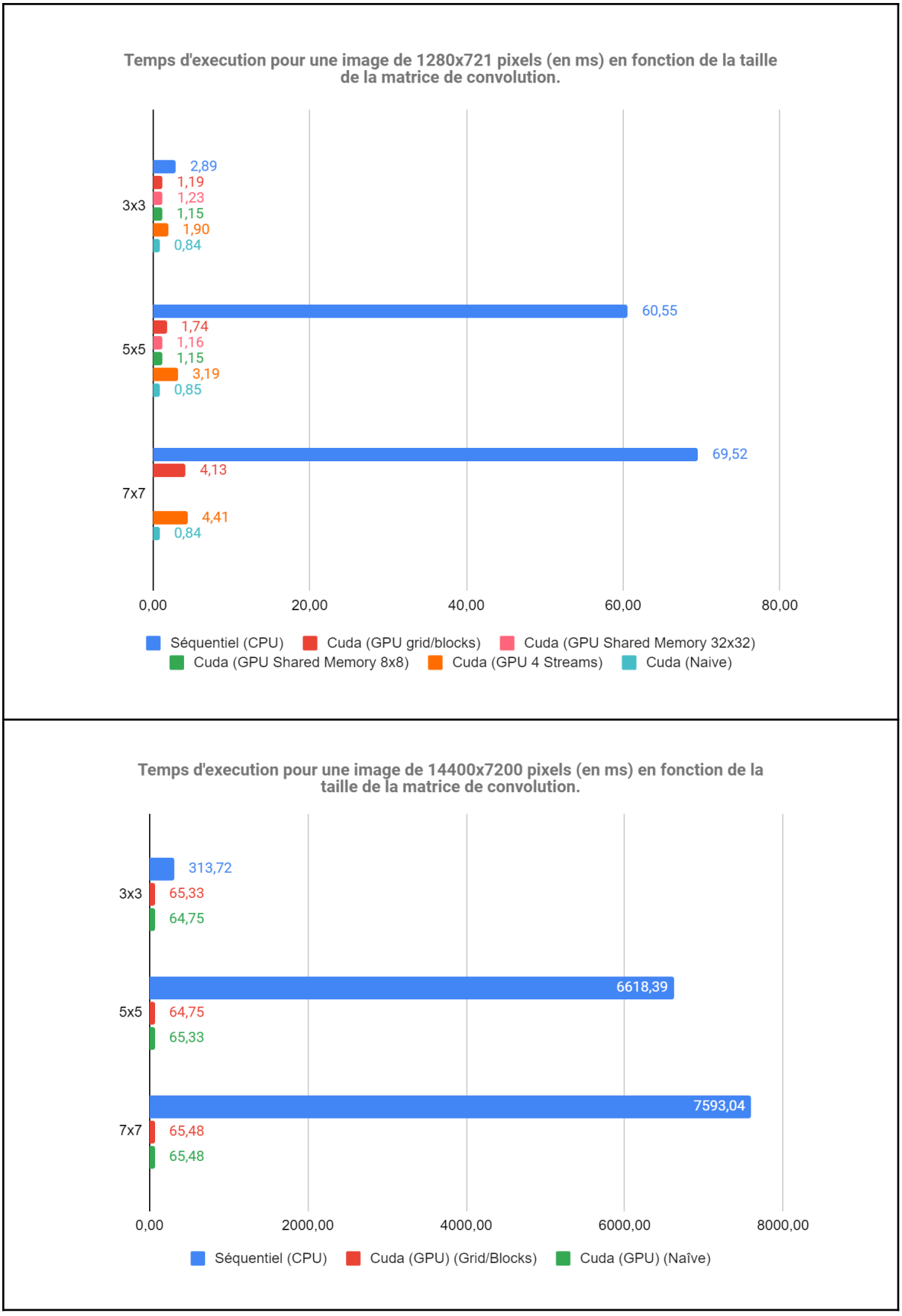
Une autre difficulté de ce projet était de pouvoir accéder à Guacamole sur le réseau universitaire, néanmoins à des fins de meilleure productivité et de facilité, nous avons effectué les calculs et modifications sur un de nos postes personnels sous Ubuntu 22.04.

IV) Conclusion et ensemble de données

Pour conclure, ce projet nous a permis de découvrir l'application de matrices de convolutions avec une librairie CUDA, librairie qui permet l'exploitation du GPU et de ses avantages en termes de performances. Nous avons aussi exploré et testé différentes optimisations de notre algorithme et nous avons pu voir l'impact des changements entre celles-ci.

Et pour terminer, voici quelques données supplémentaires montrant l'impact réel de chaque étape de notre processus mis cote à cote pour avoir un meilleur aperçu global.





Retrouvez [toutes les données des graphiques ici](#).