Projet CUDA 2023

Auteurs: Nabyl QUIGNON,
Mugilan RAGAVAN,
Térence CLASTRES.

I. Algorithmes implémentés

Pour ce projet, nous avons décidé d'implémenter les algorithmes suivant:

- Edge Detection
- Gaussian Blur
- Laplacian Operator
- Line Detection

Le filtre de convolution Edge Detection est utilisé pour détecter les contours ou les bords dans une image. Son Kernel est:

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

Le filtre gaussian blur est un filtre de convolution utilisé pour réduire le bruit et adoucir les détails d'une image. Son Kernel est :

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

Le filtre laplacian operator est un filtre de convolution utilisé pour détecter les lignes de l'image, contrairement aux autres il est très sensible au bruit. Son Kernel est:

0	-1	0
-1	4	-1
0	-1	0

The laplacian operator

Le filtre line detection est un filtre de convolution qui permet de détecter les lignes à 45 degrés. Son Kernel est:

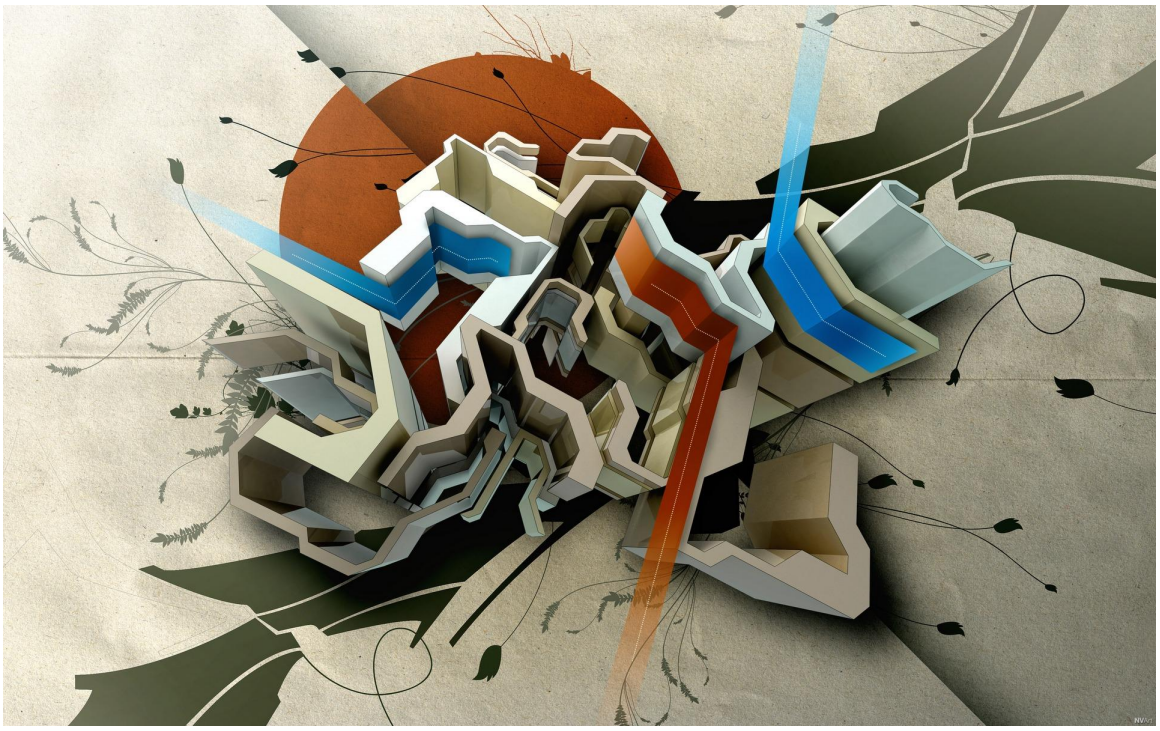
-1	-1	2
-1	2	-1
2	-1	-1

45 degree lines

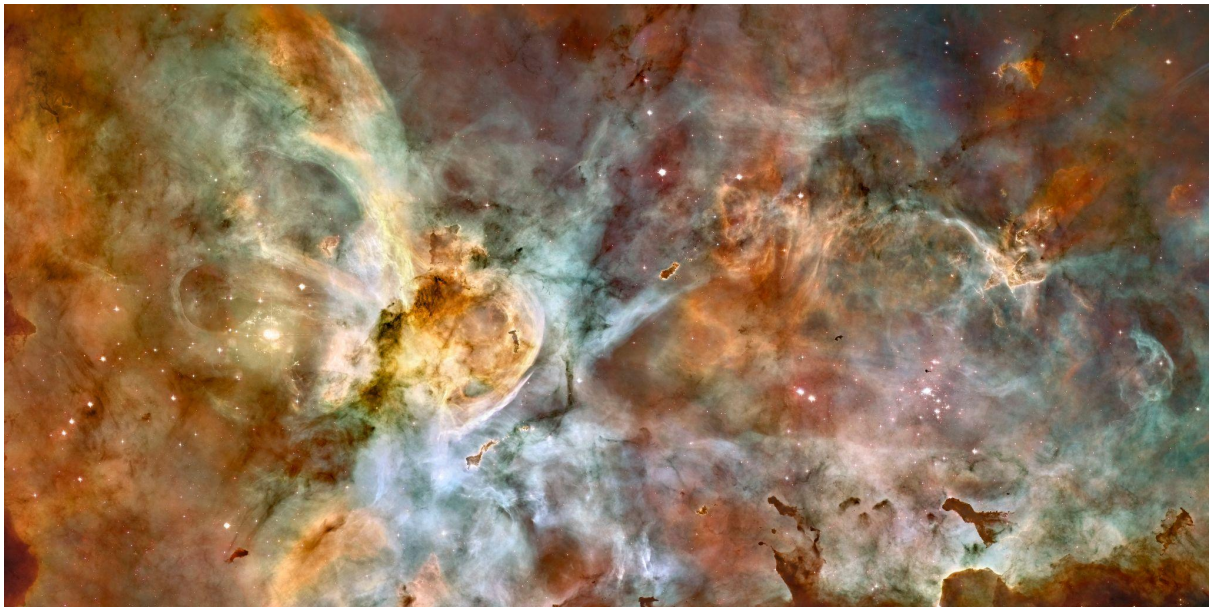
II. Images de référence

Nous avons exécuté les algorithmes sur les 2 images suivantes:

In.jpg:



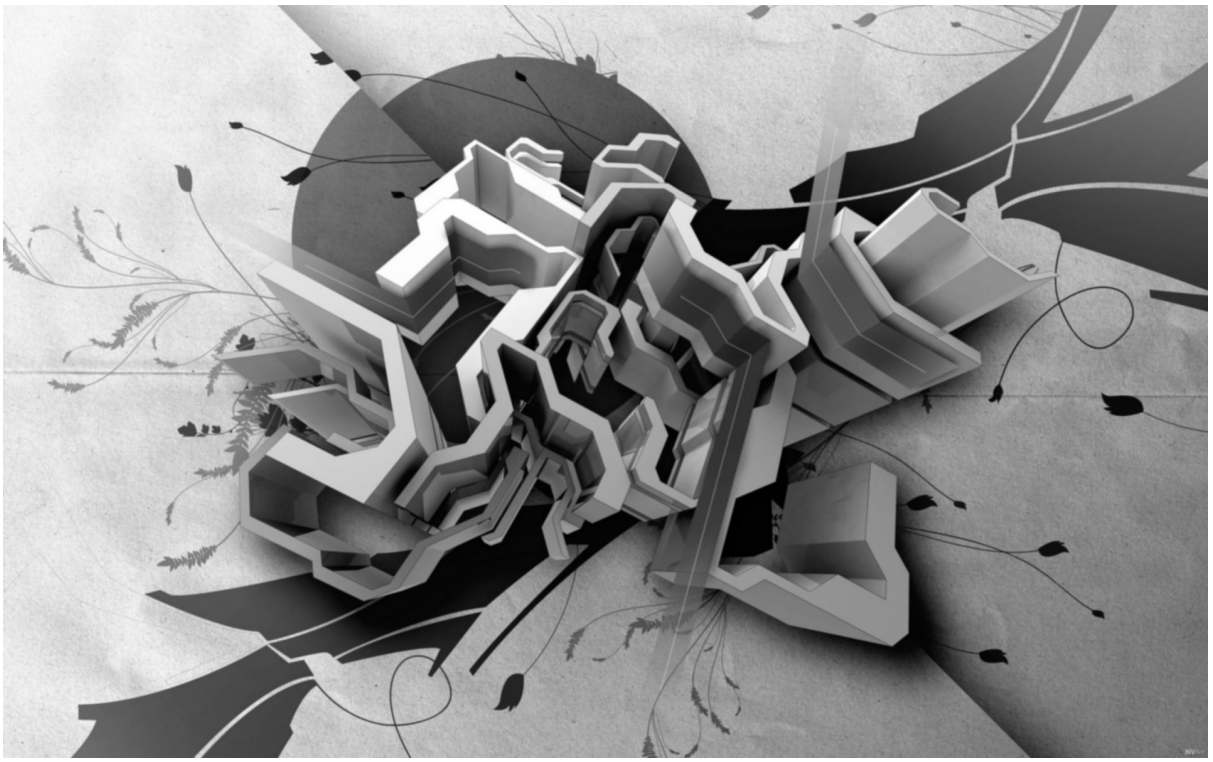
Espace.jpg:



III. Résultat de l’application des filtres

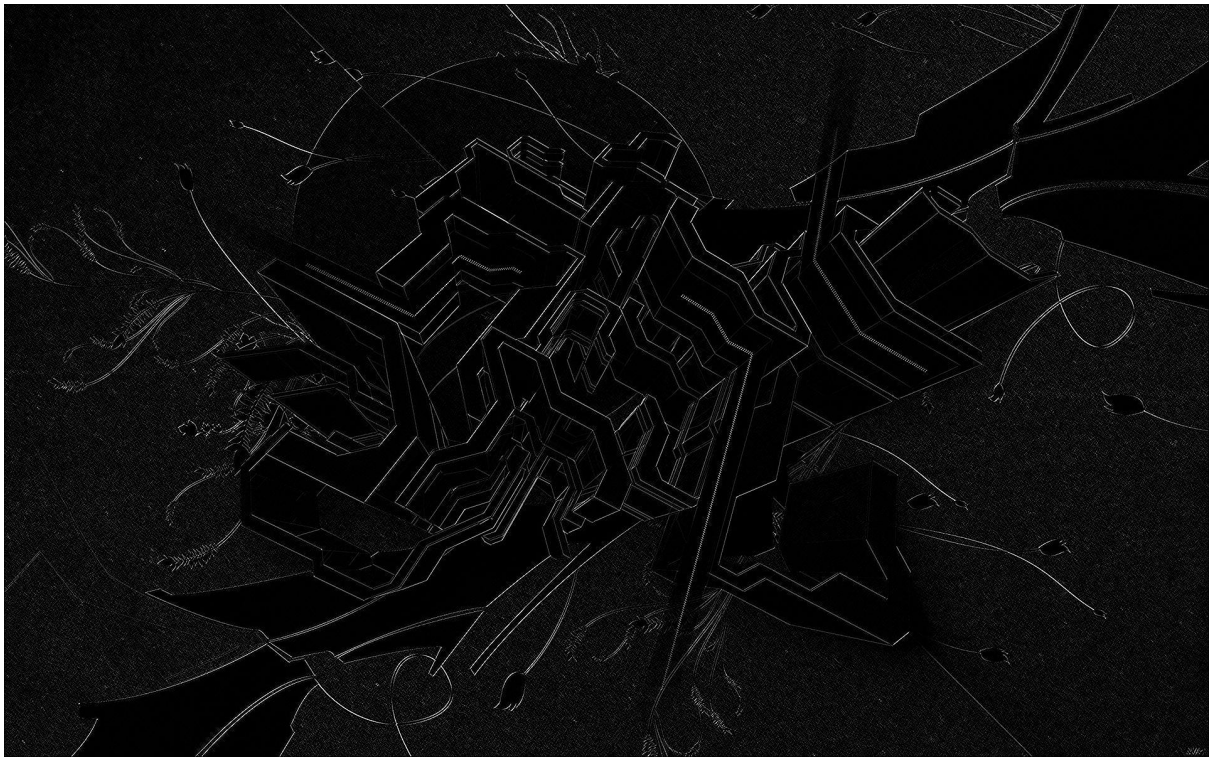
Edge Detection:

Gaussian Blur:



Laplacian Operator:

Line Detection:



IV. Optimisation et analyse des performances

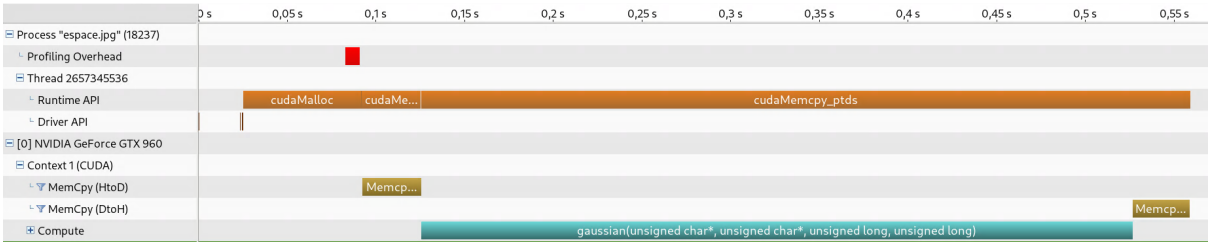
Pour les quatres différents filtres nous avons décidé d'utiliser comme optimisation:

- les streams: On découpe notre image en 2 segments horizontaux et on lance un Kernel cuda pour chaque partie sur deux streams différents.

- la mémoire partagée: Dans un premier temps, nous mettons l'image dans un registre partagé au niveau de la carte graphique afin d'éviter de récupérer l'image en dehors de la carte graphique. Dans un second temps nous appliquons le filtre à l'image.

Nous avons gardé comme taille de bloc 32x32, car nous n'avons pas observé d'amélioration assez pertinente des résultats en la changeant.

Voici de manière générale la trace d'une exécution :



On voit qu'il a de manière générale 4 phases :

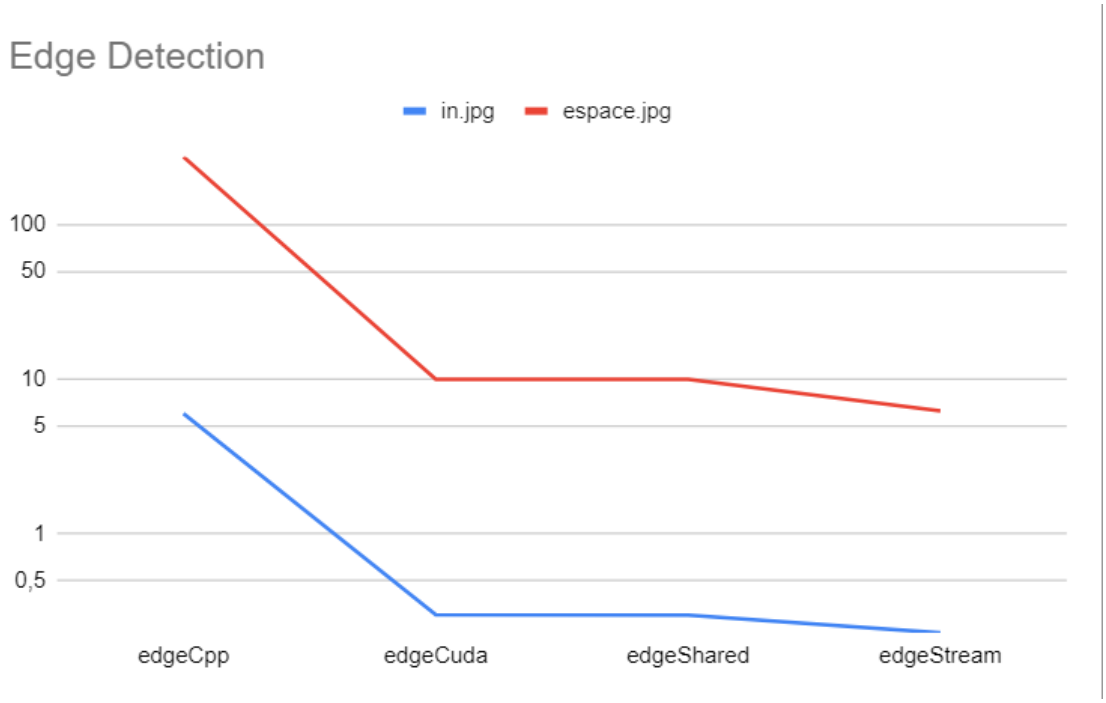
- Allocation et initialization de la mémoire
- Copie de la mémoire de l'hôte vers le GPU
- Lancement du kernel
- Copie du résultat du GPU vers l'hôte
- Libération de la mémoire

Les résultats suivants ont été obtenus en faisant la moyenne sur 10 exécution grâce à un script bash. Aussi la machine utilisée est celle disponible sur Guacamole avec la GTX 780.

Pour le filtre Edge Detection voici les résultats obtenus (L'échelle de l'axe des abscisse est logarithmique car la différence entre la version CPU et GPU est trop grande et on ne voit rien sur le graphique):

	in.jpg	espace.jpg
edgeCpp	6	275
edgeCuda	0,3	9,997
edgeShared	0,298	9,992
edgeStream	0,229	6,246

et voici le graphique lié à ce tableau:

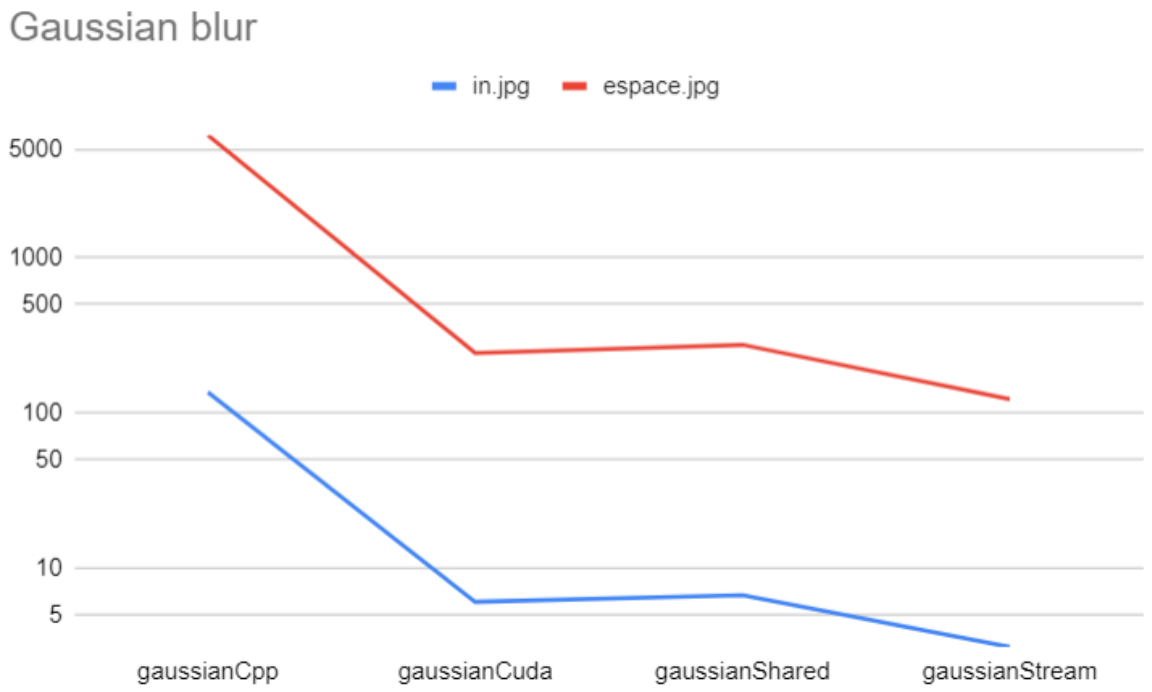


En comparant les résultats, on peut observer que la version C++ basique du filtre prend naturellement plus de temps que les versions utilisant la carte graphique.
On peut voir que les performances de la version utilisant la mémoire partagée sont quasiment identiques que la version cuda naïve.
La version utilisant les streams est quant à elle bien plus performante que les deux précédentes.

Pour le filtre Gaussian Blur voici les résultats obtenus:

	in.jpg	espace.jpg
gaussianCpp	136	6162,7
gaussianCuda	6,058	243,017
gaussianShared	6,685	275,077
gaussianStream	3,098	122,762

et voici le graphique lié à ce tableau:

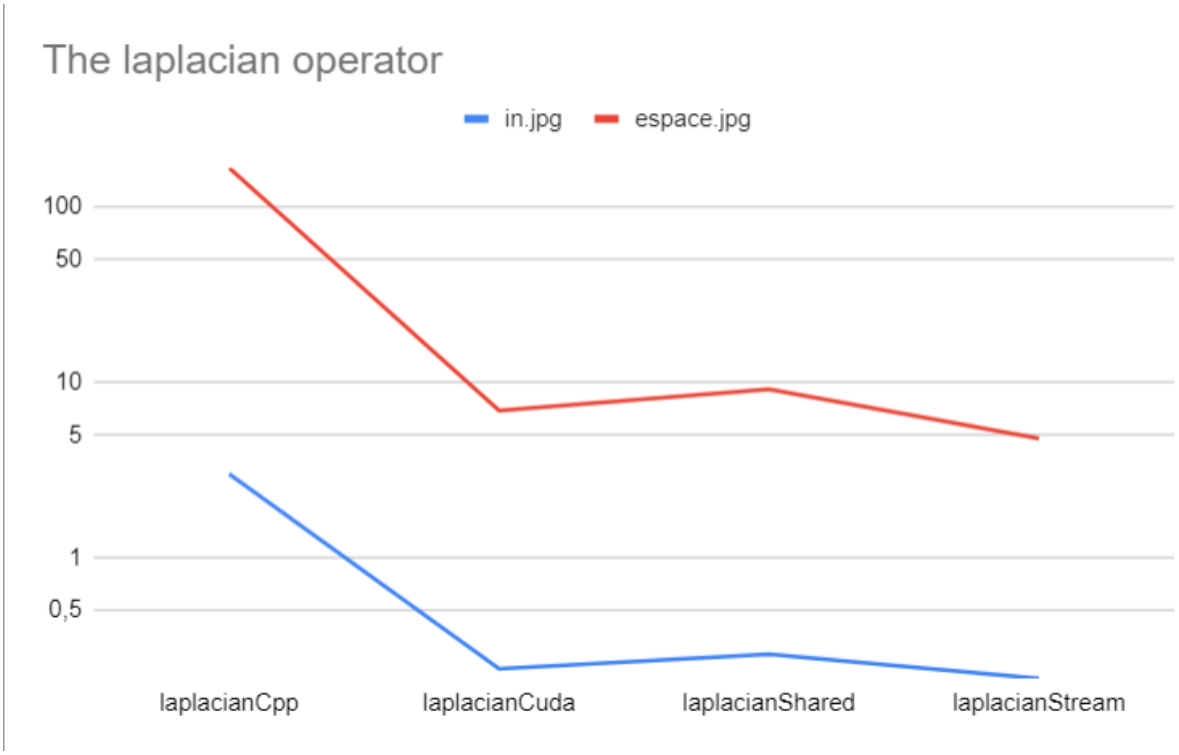


En comparant les résultats, on peut observer que la version C++ basique du filtre prend naturellement plus de temps que les versions utilisant la carte graphique.
On peut voir que les performances de la version utilisant la mémoire partagée sont quasiment identiques que la version cuda naïve.
La version utilisant les streams est quant à elle bien plus performante que les deux précédentes, l'amélioration entre la version naïve et avec mémoire partagé est beaucoup plus significative, ceci est dû au fait que le Kernel du filtre Gaussian est plus grand que celui de Edge Detection.

Pour le filtre Laplacian Operator voici les résultats obtenus:

	in.jpg	espace.jpg
laplacianCpp	3	165,8
laplacianCuda	0,233	6,877
laplacianShared	0,282	9,107
laplacianStream	0,205	4,776

et voici le graphique lié à ce tableau:

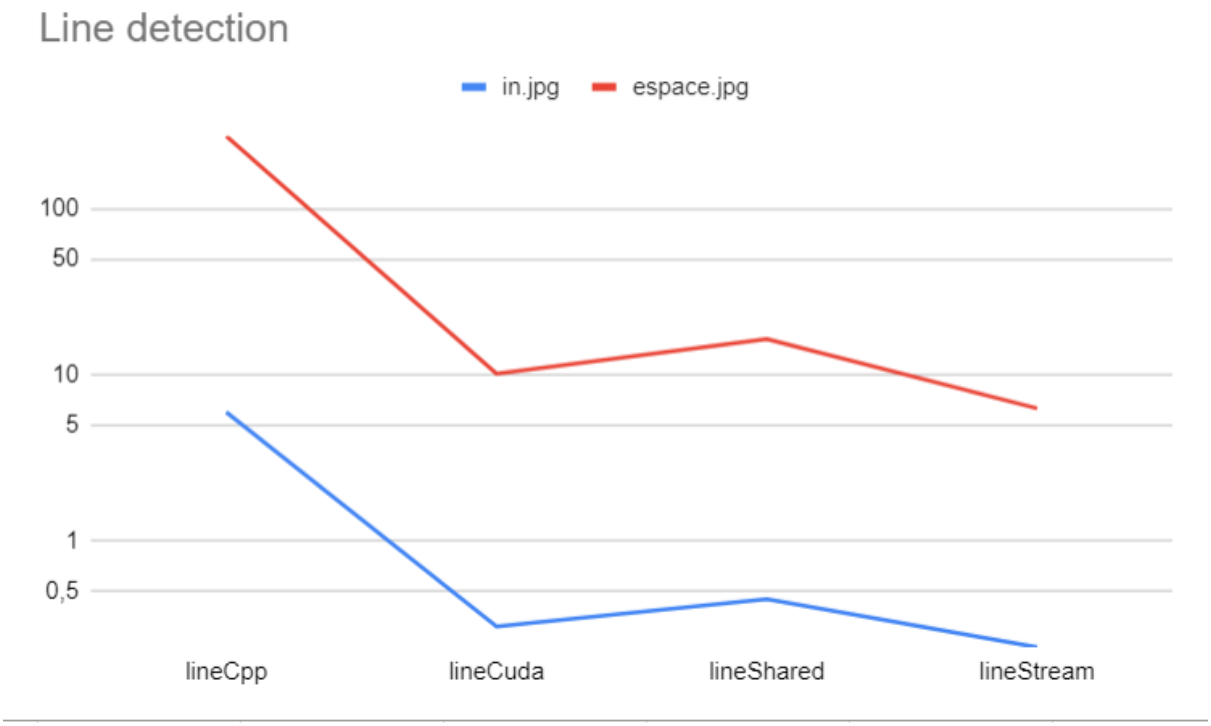


En comparant les résultats, on peut observer que la version C++ basique du filtre prend naturellement plus de temps que les versions utilisant la carte graphique.
On peut voir que les performances de la version utilisant la mémoire partagée sont pires que la version cuda naïve.
La version utilisant les streams est quant à elle bien plus performante que les deux précédentes.

Pour le filtre Line Detection voici les résultats obtenus:

	in.jpg	espace.jpg
lineCpp	6	275
lineCuda	0,306	10,188
lineShared	0,449	16,52
lineStream	0,23	6,316

et voici le graphique lié à ce tableau:

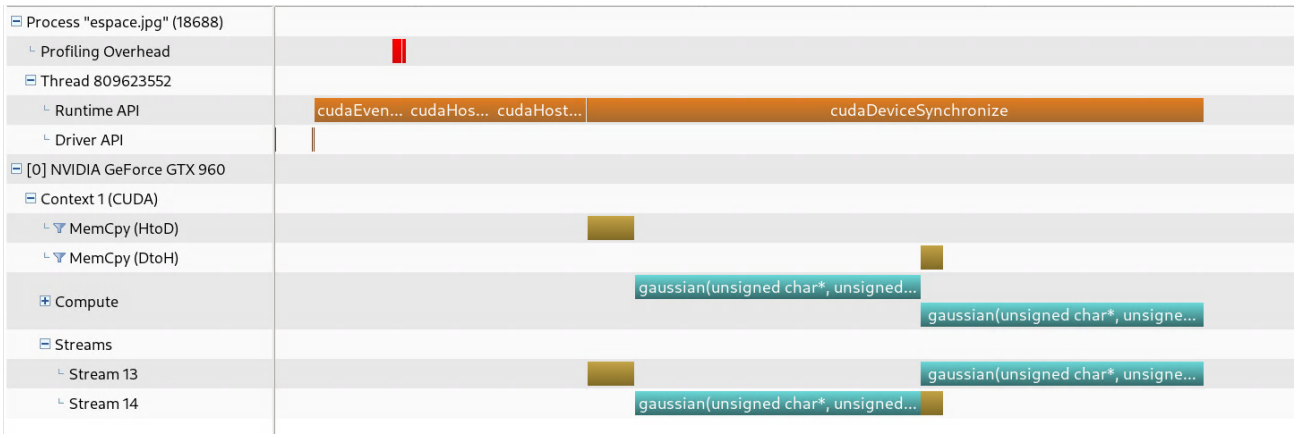


En comparant les résultats, on peut observer que la version C++ basique du filtre prend naturellement plus de temps que les versions utilisant la carte graphique.
On peut voir que les performances de la version utilisant la mémoire partagée sont pires que la version cuda naïve.
La version utilisant les streams est quant à elle bien plus performante que les deux précédentes.

V. Analyse des difficultés rencontrés et discussion

Nous avons trouvé le projet extrêmement répétitif, car l'implantation d'un nouveau filtre consistait seulement à reprendre l'ancien et modifier la partie Kernel (À part pour le Gaussian Blur qui était légèrement différent car le kernel avait une taille plus importante).

Nous avons aussi rencontré un problème lorsque nous avons voulu faire en sorte que les streams s'exécutent en parallèle, les Kernels ne se superposent pas (Même en utilisant l'option –default-stream per-thread) comme nous pouvons le voir sur l'image suivante (nous avons utilisé NVVP pour monitorer l'utilisation des streams):



au lieu d'avoir ce genre de graphique:

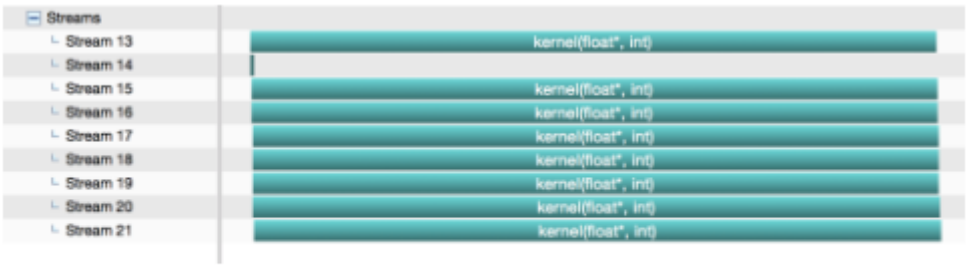


Figure 2: Multi-stream example using the new per-thread default stream option, which enables fully concurrent execution.

(https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/)

Aussi, nous trouvons étrange que les performances de la version utilisant la mémoire partagée soit pire dans certains cas que la version cuda naïve. Ceci peut être dû au fait que nous avons mal implémenté cette version (nous nous somme inspiré de la version faite en TP).

Il faut se référer au README pour l'exécution des différents algorithmes.
Il y'a un seul commit dans sur GitHub car nous avons utilisé BitBucket comme Git pour se projet.