Jean-Pacôme Delmas
Guillaume Troesch

Université d'Orléans
M1 IMIS

Rapport Programmation Graphique

Cuda : Convolutions

2019-2020

Ce rapport a été écrit dans le cadre du module Programmation Graphique pour le projet Cuda. Celui-ci a été réalisé par un groupe de deux étudiants. Le but de ce projet est de créer différents filtres de traitement d'image et différentes méthodes d'implémentations de ces filtres, pour ensuite étudier leurs performances. Les fichiers ont été adaptés pour qu'une exécution unique produise tous les résultats de tous les filtres.

Les filtres de ce projet sont des convolutions, c'est-à-dire que chaque pixel d'une image devra calculer une nouvelle valeur en fonction de son voisinage de pixels pour appliquer le filtre. Les filtres choisis sont "***Blur***", "***Sharpen***" et "***Edge_detect***". Le premier filtre consiste à rendre l'image plus floue, le second consiste à rendre l'image plus nette, et le troisième consiste à détecter les bords des formes dans l'image.

Deux combinaisons de filtres ont également été réalisées. Une première combinaison applique "***Blur***" puis "***Edge_detect***", la seconde applique les deux même filtres, mais dans un ordre inversé, c'est-à-dire "***Edge_detect***" puis "***Blur***". Nous nous sommes servis du lien https://beej.us/blog/data/convolution-image-processing/ mentionné dans le sujet, comme base pour l'implémentation de nos filtres.

Ces filtres ont été créés de six manières différentes pour tester leurs performances. Toutes ces implémentation ont été réalisées de manière à être lisibles, d'où l'utilisation de variables intermédiaires au sein d'un kernel. La grille de blocs utilisée est toujours de taille identique à la taille de l'image.

- **Implémentation séquentielle** : consiste à réaliser tous les calculs directement sur le CPU, sans utiliser le GPU, et sert de repère pour se rendre compte de l'efficacité du calcul sur GPU.

Les implémentations suivantes sont toutes réalisées avec des blocs en deux dimensions (32*32) puis en trois dimensions (17*20*3). En deux dimensions, le parcours de chaque indice i et j représente le traitement du pixel en position (i, j) dans l'image. Chaque thread du bloc fait donc 3 calculs, puisque chaque pixel contient 3 couleurs. En trois dimensions, le parcours dans un bloc avec chaque indice i, j, k représente la couleur k dans le pixel (i, j). Chaque thread du bloc réalise donc 1 calcul. La taille choisies des blocs tend à se rapprocher de largeurs et de longueurs égales, tout en respectant la limite de 1024 threads.

- **Implémentation simple avec kernels** : Utilisation des kernels dans le GPU divisée en blocs.
- **Implémentation avec kernels et mémoire shared** : Les résultats sont enregistrés dans une mémoire shared au sein d'un thread pour faciliter un accès ultérieur. Cette implémentation n'a de sens que pour les combinaisons de filtres.
- **Implémentation avec kernels et streams** : L'image est coupée en 2 afin de pouvoir commencer le calcul dès que la 1ère moitié est sur la carte, et envoyer la 2nde moitié pendant le calcul sur la 1ère. Pour pouvoir transmettre et calculer en même temps il faut utiliser les fonctions de copie asynchrones. De plus, il faut rajouter 1 ou 2 lignes de l'image aux streams pour éviter d'avoir une bordure noire au milieu de l'image.
- **Implémentation avec kernels avec mémoire shared et streams** : Même procédé que pour l'implémentation avec kernels et streams, mais on y ajoute un espace mémoire partagé afin d'améliorer la vitesse d'exécution lors d'une combinaison de filtres.

Tableau récapitulatif de nos expériences sur machine de l'université en millisecondes

Filtre Méthode	Blur	Sharpen	Edge_detect	Blur + Edge_detect	Edge_detect + Blur	Fichier
Séquentielle	94	74	88	184	182	main_seq.cu
Kernels (32*32)	0.87	0.46	0.45	1.23	1.22	main_kernel.cu
Kernels (17*20*3)	1.62	1.08	1.07	2.68	2.68	main_kernel.cu
Kernels + shared (32*32)	0.90	0.49	0.49	1.22	0.91	main_shared.cu
Kernels + shared (17*20*3)	1.59	1.01	0.98	2.42	1.83	main_shared.cu
Kernels + streams (32*32)	0.86	0.43	0.43	1.18	1.18	main_stream.cu
Kernels + streams (17*20*3)	1.58	0.97	0.94	2.52	2.52	main_stream.cu
Kernels + streams + shared (32*32)	0.87	0.45	0.44	1.21	1.21	main_shared_ stream.cu
Kernels + streams + shared (17*20*3)	1.55	0.91	0.89	2.42	2.41	main_shared_ stream.cu

Tableau récapitulatif de nos expériences sur machine personnelle en millisecondes (CPU i5-7200U, GPU GeForce 940MX).

Filtre Méthode	Blur	Sharpen	Edge_detect	Blur + Edge_detect	Edge_detect + Blur	Fichier
Séquentielle	116	98	117	262	246	main_seq.cu
Kernels (32*32)	2.87	1.50	1.50	4.37	4.38	main_kernel.cu
Kernels (17*20*3)	5.20	3.15	3.16	8.36	8.34	main_kernel.cu
Kernels + shared (32*32)	2.90	1.55	1.50	3.73	2.54	main_shared.cu
Kernels + shared (17*20*3)	5.27	3.19	3.21	6.89	5.49	main_shared.cu
Kernels + streams (32*32)	2.97	1.54	1.54	4.51	4.50	main_stream.cu
Kernels + streams (17*20*3)	5.27	3.18	3.20	8.43	8.42	main_stream.cu
Kernels + streams + shared (32*32)	2.96	1.54	1.54	4.50	4.49	main_shared_ stream.cu
Kernels + streams + shared (17*20*3)	5.19	3.12	3.13	8.32	8.30	main_shared_ stream.cu

Les résultats sont tous effectués sur l'image *in.jpg*, et sont réduits au centième pour faciliter l'interprétation.

A partir des tableaux des résultats ci-dessus, en comparant les résultats des méthodes séquentielles sur les deux machines, on peut constater que le CPU de la machine de l'université est entre 1.2 et 1.4 fois plus rapide que la machine personnelle. Les autres méthodes indiquent une différence de performances sur GPU de 3.2 à 3.6 fois plus rapides sur les machines de l'université.

Si on compare la rapidité d'exécution des filtres, on remarque que les filtres *Sharpen* et *Edge_detect* ont des temps d'exécution similaires et qui sont plus rapides que le filtre *Blur*. Cela est dû au fait qu'ils ne doivent accéder qu'à 4 pixels pixels voisins, alors que blur doit accéder à 8. *Blur* est lui-même plus rapide que les deux combinaisons de filtres qui ont des temps d'exécution similaires. La seule exception dans cette comparaison concerne la combinaison de filtres Edge_detect puis Blur qui avec des mémoires shared attends un temps d'exécution équivalent à Blur seul. Cette exception n'est cependant pas vérifiée avec des shared et des streams.

En comparaison des blocs, bien que les threads des blocs de 17*20*3 ne traitent qu'une seule couleur et sont donc plus rapides que les threads des blocs de 32*32 qui traitent 3 couleurs, leurs temps d'exécution est toujours 2 fois plus lent que les blocs de deux dimensions. Cela peut être dû au fait que 17*20*3 n'a que 1020 threads et n'est pas un multiple de 32, et n'est donc pas couvert par des warps entiers. La grille de blocs 32*32 ont également moins de blocs à traiter que la grille de blocs 17*20*3 puisqu'un bloc couvre 1024 pixels contre 340 pixels (17*20), soit 3 fois plus de pixels. Une autre hypothèse pourrait être que les GPU n'ont pas assez de warps actifs pour exécuter tous les threads simultanément, d'où le temps d'exécution différent.

Bien que les filtres simples n'ont pas de réel intérêt en mémoire shared, puisqu'ils copient d'abords leurs résultats dans la mémoire shared, pour y accéder immédiatement pour l'enregistrer dans l'image résultante, ils résultent en des temps identiques à 0.05 ms près aux kernels simples. On peut en conclure que l'accès aux mémoires shared est très peu coûteux. Un autre léger détail, on peut noter que les méthodes de filtre en stream_shared sont toujours plus rapides que la méthode stream uniquement (x1.02 à x1.06 plus rapide).

Si on considère des exécutions de méthodes différentes avec des blocs identiques, on peut observer deux comparaisons différentes sur les temps d'exécutions des deux machines. Globalement, sur la machine de l'université, les streams et stream_shared sont équivalents au kernel simple, alors que sur la machine personnelle, les streams et stream_shared sont toujours plus lents que les kernels (jusqu'à une perte de 3% en vitesse), mais ces différents peuvent être négligés car elles sont minimes.

En revanche, sur les filtres combinés, les shared sont toujours plus rapides. Cette observation est d'autant plus valable sur Edge_detect puis Blur que sur Blur puis Edge_Detect, puisque Blur demande d'accéder à plus de pixels voisins, jusqu'à 1.33 fois plus rapide comparé aux kernels simples sur la machine de l'université, et jusqu'à 1.72 fois plus rapide sur la machine personnelle. Cette observation est accentuée par le temps remarquable de seulement 2.54 ms sur la machine personnelle, indiquant un accès aux mémoires shared bien plus rapide sur cette machine.

Une autre implémentation avait été réalisée avec des kernels simples, mais avait des comportements étranges. L'idée était traiter chaque couleur dans un kernel, avec des blocs 32*32, et une grille en deux dimensions de taille longueur_image * (largeur_image * 3). Les comportements étranges de cette implémentation étaient visibles aux bordures de chaque ligne. Puisque la matrice utilisée en une dimensions pour représenter l'image est de la forme [r1, g1, b1, r2, g2, b2, ...], les indices (i, j) correspondent ici à chaque couleur de chaque pixel, et non plus à un pixel entier. Le comportement espéré ici est donc d'obtenir un indice j allant jusqu'à 3 fois la largeur de l'image, pourtant le comportement étudié menait à un parcours en obtenant trois fois des valeurs i et j identiques, résultant à des bordures sur la hauteur à chaque tiers de l'image. Après de nombreux essais et de tests pour résoudre ce problème, et en considérant que cette implémentation agissait de manière similaire à des blocs en trois dimensions puisque chaque kernel ne traite qu'une couleur, cette implémentation a finalement été abandonnée.

De plus, nous avons rencontré des problèmes avec les shared, notamment avec une bordure droite et basse qui n'était pas noire, mais affichait plutôt des pixels de couleurs aléatoires. On a résolu ce problème en gérant mieux le tableau extern __shared__ (les bordures étant initialisées à 0) car celui-ci générait des valeurs aléatoires sur les bordures plutôt que des valeurs nulles.

Nous avons aussi dû faire face à un second problème de bordures en utilisant les streams. Il apparaissait une bordure au milieu de l'image filtrée en raison du découpage de l'image en deux, on a résolu ce problème en augmentant d'une ligne ou 2 la taille des streams.

Pour conclure, pour appliquer un traitement à une image, cuda propose une programmation concurrente sur GPU bien plus efficace que sur CPU (de 60 à 200 fois plus rapides). En fonction des traitements à appliquer, s'il s'agit d'un filtre de convolution unique, une programmation par blocs en deux dimensions multiples de 32 avec des kernels simples ou des streams semblent le plus approprié. En revanche, s'il s'agit de plusieurs filtres à appliquer, les shared sont toujours plus efficaces pour réduire les accès aux mémoires entre CPU et GPU, et sont d'autant plus efficaces sur des machines avec un partage de mémoires plus développé.