1 Structure du projet

Nous avons mis le code de notre projet dans un dossier appelé "code". A l'intérieur de ce dossier vous pouvez y retrouver :

- un dossier img où il y a l'image du château fournis avec le sujet et deux autres images de plus haute résolution : un fond d'écran South Park et une image de chèvre, modifiées par nos soins à l'aide de Photoshop.
- un dossier utils avec la lib stb, la classe image fournis avec le sujet, les chronos et le common fournis lors des tp, et notre classe HSV sur laquelle nous reviendrons.
- "codeCPU" dans lequel nous implémentons les 5 fonctions demandées pour la version CPU.
- "codeGPU" dans lequel nous implémentons 5 fonctions et 5 kernels demandés pour la version GPU.
- notre main qui permettra de tester notre programme en créant des nouvelles images dans le dossier img correspondantes à l'égalisation d'histogramme de l'image fournis avec le sujet ainsi que l'affichage dans la console des temps réalisés par nos fonctions de la version cpu et les kernels de la version gpu. Et dans lequel nous vous fournissons la ligne de commandes nécessaires à la compilation de notre projet. Elle se situe en haut de notre fichier main, en commentaire.

Pour simplifier le passage en argument des données des images en HSV, nous avons créé une classe HSV contenant 3 tableaux de float (H, S et V) et un int représentant la taille de ces tableaux (et représentant aussi le nombre de pixels de l'image). La classe dispose d'un constructeur ne demandant que le nombre de pixels de l'image afin de créer les tableaux à la bonne dimension et d'un destructeur, détruisant ces 3 tableaux. Pour faciliter l'accès et le remplissage de nos attributs, ceux-ci sont tous publics.

2 Code version CPU

Pour coder les fonctions rgb2hsv et hsv2rgb nous avons simplement été nous renseigner pour savoir comment passer de l'un à l'autre sur la page wikipedia HSL_and_HSV dont le lien est fournis dans le sujet et nous avons implémenté les fonctions comme nous les avons comprise.

Pour les fonctions de calcul d'histogramme et de fonction de répartition, la encore il n'y avait pas grand chose à faire. Pour le premier nous avons simplement créé un tableau de 256 cases (ce qui correspond au nombre de valeurs différente que peut prendre la composante V), où toutes les cases sont initialisées à 0 puis pour chaque pixels de l'image nous ajoutons 1 à l'endroit qui correspond à sa nuance en V dans notre tableaux. Et pour la fonction de répartition nous avons simplement créé là aussi un tableau de 256 cases et la valeur de chaque case correspond au nombre de pixels cumulés ayant la nuance de V correspondant à la case ou une nuance plus petite.

Et enfin pour la l'égalisation d'histogramme nous avons simplement implémenté la fonction T(xk)

trouvé sur le wikipedia français de l'égalisation d'histogramme. Le principe est simple. Pour chaque pixel de notre image, pour changer la valeur de sa composante V, on fait le rapport entre le nombre de nuances de V possible -1 et le nombre de pixel de notre image que l'on multiplie à la valeur de la fonction de répartition de notre pixel actuel.

3 Code version GPU

Puisque nous travaillons sur des images, la plupart de nos algorithmes sont assez facile à paralléliser, car chaques opérations sur un pixels peuvent souvent être faites indépendamment des autres. On peut se dire qu'un thread exécute un travail pour un pixel.

Et c'est ce que nous avons fait pour les fonctions rgb2hsv, hsv2rgb et pour l"égalisation. Nous avons repris notre code de la version cpu et nous l'avons parallélisé de la manière expliquée ci-dessus.

En ce qui concerne le calcul de l'histogramme nous avons réalisé le même traitement pour chaque pixels en parallèle cependant il a fallu que nous gérions les accès concurrents à la mémoire. En effet, les threads peuvent vouloir incrémenter la même case du tableau au même moment, donc nous avons utilisé un atomicAdd pour remédier à ça. Cependant la solution trouvée pour pallier le problème d'accès concurrent mémoire se ressent dans les performance.

Pour la répartition, nous avons choisi de faire une réduction. Au tout début nous avons un thread pour chaque case de l'histogramme, chaque thread remplit sa valeur de l'histogramme dans celle du tableau de répartition correspondante. Puis nous commençons la réduction, à chaque étape nous doublons le nombre d'éléments que chaque thread gère soit à l'étape deux, le thread d'id 0 gère la case 0 et 1 et le thread d'id 1 les case 2 et 3. Puis chaque thread ajoute la valeur de la case précédente de la moitié du sous tableau qu'il gère à tous ses suivants (si un thread gère par exemple les cases d'indice 4,5,6 et 7 il ajoutera la valeur de la case d'indice 5 aux case 6 et 7). Et donc avec cette méthode à chaque étape la moitié des threads arrêtent de travailler.

4 Les performances

Tous les résultats présentés ci-dessous, ont été calculés sous Windows 10 sur un AMD Ryzen 7 1800X@3.6GHz et une Nvidia RTX3080@1710MHz. Le programme à été compilé sous nvcc avec le niveau d'optimisation O1.

Dans un premier temps, nous avons comparé différentes configurations pour chacun des kernels, afin de trouver de façon empirique la configuration la plus performante. D'après le tableau suivant, nous pouvons voir que la majorité de nos kernels les plus performants avec une grille de 512 blocs et des blocs de 1024 threads. On remarque également que la fonction RGB vers HSV est la plus efficace pour un plus grand nombre de blocs et un moins grand nombre de threads. Enfin la fonction qui elle aussi fonctionne sur 1024 threads mais seulement 128 blocs.

gridSize, blockSize	rgb2hsv	histo	repart	equalization	hsv2rgb
128, 128	4,20141	0,883712	0,231584	1,19494	1,93626
128, 256	4,21696	0,922624	0,348992	1,16317	1,91386
128,1024	4,09395	0,917504	0,232448	1,13254	1,83293
256,256	4,13485	0,862208	0,504832	1,47149	1,8729
1024,256	4,05418	1,22557	0,223232	1,23178	1,87498
1024,1024	4,14208	0,851968	0,236544	1,13971	1,8985
512,1024	4,12138	0,844672	0,220384	1,21958	1,82784

Table 1 – Comparaison des configuration des kernels pour Château.png

Maintenant que nous avons établi une configuration décente pour nos kernels, nous allons pouvoir comparer les performances des différents algorithmes dans leurs versions CPU et GPU. Pour cela nous utiliserons trois images : le château fournis dans le sujet (960x640), un fond d'écran South Park(1920x1080) et une chèvre (3840x2160).

CPU	rgb2hsv	histo	repart	equalization	hsv2rgb
960x640 - Château	9,9835	3,401	0,0001	1,1946	35,7133
1920x1080 - South Park	27,1459	9,2059	0,0001	3,9054	125,255
3840x2160 - Chèvre	177,002	45,2786	0,0002	16,9166	419,407
GPU	rgb2hsv	histo	repart	equalization	hsv2rgb
960x640 - Château	4,23014	0,85728	0,222208	1,22573	1,81862
1920x1080 - South Park	13,9477	2,17088	0,23456	2,99418	5,06982
3840x2160 - Chèvre	79,4264	6,51162	0,363296	10,0884	18,3378

Table 2 – Comparaison des performance CPU/GPU

Comme on peut le voir dans la Table 2, certaines fonctions sont moins efficientes dans leurs versions parallèles, notamment les fonctions de répartition et d'égalisation. Cependant dès que la taille du travail initial grandit (i.e. l'image), l'intérêt de la parallélisation des fonctions prend sens, notamment les fonctions hautement parallèles comme les conversions hsv et rgb car celles-ci peuvent assigner un thread par pixel et donc diviser le travail de façon bien plus efficiente, tandis que les fonctions comme le remplissage de l'histogramme, ou encore l'equalisation qui s'exécute plus vite dans leurs versions parallèle mais pas au point des fonction précédentes. Enfin la fonction de répartition ne peut pas aller plus vite que la version séquentielle à cause du coup en mémoire, en effet la combinaison d'une fonction qui se paralléliser très mal et d'un coup en IO grandissant avec la taille du travail produit un résultat qui est bien plus lent qu'une simple version séquentielle sans nécessité de réallouer de la mémoire.