Université de Bordeaux

Projet technologique
Traitement d'image sous android

Auteur :
Kamel Houchat

Résponsables :
Fabien Baldacci
Boris Mansencal
Anne Vialard

8 janvier 2020

TP 1

Togray && Tograys:

Dans la première version de la fonction, on a utiliser les deux méthodes getpixel() et setpixel() qui étaient beaucoup moins éfficace que getpixels() et setpixels(), et après l'implémentation d'un noyau en utilisant RenderScript on a constaté une amélioration en comparons le temps qu'elles prennent et l'utilisation de la mémoire.

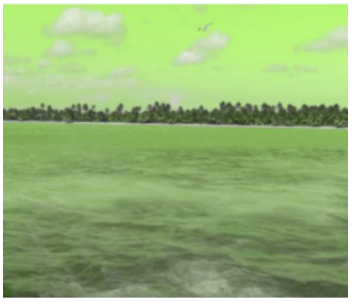
Je tiens à préciser que tous les testes sont effectué sur un téléphone SAMSUNG GALAXY A8 | Version android : 8

Pour le temps voici les résultats :

ToGray	0.721
ToGrays	0.033
ToGray (RenderScript)	0.016

TP 2

Colorize :

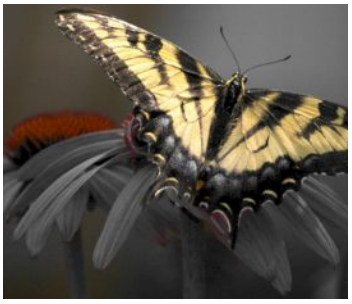


La Première version étant implémenter avec JAVA est clairement plus lente que la version faite avec un noyau RenderScript,le hue qui est généré aléatoirement.

Pour le temps voici les résultats :

Colorize	0.788
Colorize (RenderScript)	0.021

Keepcolor :



La Première version étant implémenter avec JAVA est clairement plus lente que la version faite avec un noyau RenderScript,le hue qu'on veut conserver est

passé en paramètre à la fonction qui génère le noyau, pour la version java on lui passe une couleur en RGB.

Pour le temps voici les résultats :

Keepcolor	0.988
Keepcolor (RenderScript)	0.023

TP 3

Contrast plus gray:

Dans la première version de la fonction, on n'a pas utilisé la table LUT , elle était beaucoup moins efficace que la deuxième (utilisation de la table LUT) , et après l'implémentation d'un noyau en utilisant RenderScript on a constaté une amélioration en comparons le temps qu'elles prennent et l'utilisation de la mémoire.

Pour le temps voici les résultats :

Sans lut	0.110
Avec lut	0.044
RenderScript	0.036

Contrast moins gray:

La première version est implémenté en java (avec lut) et après l'implémentation d'un noyau en utilisant RenderScript on a constaté une amélioration en comparons le temps qu'elles prennent et l'utilisation de la mémoire.

Pour le temps voici les résultats :

c-moins java	0.050
c-moin RenderScript	0.034

Contrast plus RGB:

La première version est implémenté en java et après l'implémentation d'un noyau en utilisant RenderScript on a constaté une amélioration en comparons le temps qu'elles prennent et l'utilisation de la mémoire.

Pour le temps voici les résultats :

contraste plus rgb (java)	0.060
contraste plus rgb (RS)	0.034

Contrast plus HSV:

La première version est implémenté en java et après l'implémentation d'un noyau en utilisant RenderScript on a constaté une amélioration en comparons le temps qu'elles prennent et l'utilisation de la mémoire.

l'histogramme est fait avec la plus grande valeur entre le h,v et s

Pour le temps voici les résultats :

contraste plus HSV (java)	0.065
contraste plus HSV (java)	0.033

Egalisation gray:

Le calcule de l'histogramme cumulé est l'opération la plus gourmande dans la réalisation de l'égalisation ! ,l'implémentation d'un noyau en utilisant RenderScript a amélioré la performance en comparons avec la version java.

Pour le temps voici les résultats :

Egalisation java	0.280
Egalisation RenderScript	0.033

Egalisation RGB:

Le calcule de l'histogramme cumulé est l'opération la plus gourmande dans la réalisation de l'égalisation ! ,l'implémentation d'un noyau en utilisant RenderScript a amélioré la performance en comparons avec la version java.

l'histogramme est fait sur le gris et appliqué aux trois canaux

Pour le temps voici les résultats :

java	0.300
RenderScript	0.034

TP 4

Filtre moyenneur:

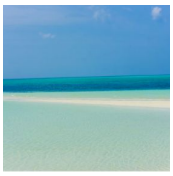
C’est une convolution 3X3 et les bordures de l’image sont mis à zéro.
la version RS est en cours d’implémentation

Pour le temps voici les résultats :

conc 3X3	0.156

.

Taille des images :



770x513

428x356

1000x560

2000x1332

256x256

615x410