RAPPORT MINI PROJET

IMAGE DOWNSIZE

ANALYSE DU CODE EXISTANT

Le code de **simple_decimation** consiste à ne garder qu'un pixel sur deux de l'image de base. Cette méthode n'est pas satisfaisante car on perd toute les données présentes dans le pixel supprimé. Le résultat est bien visible lorsqu'on décime *mire1024x768*, notamment sur les successions de bandes noire. En effet, comme on ne prend qu'un pixel sur deux, si à chaque pixel on tombe dans une bande noire, le résultat sera un carré noir au lieu des rayures de l'image de base.

AMELIORATION

On réalise une convolution sur les données avant de faire la décimation car cela permet de ne pas rejeter complètement la moitié des pixels.

Comme notre fréquence d'échantillonnage normalisée est 0.5 (on prend 1 pixel sur 2), on peut prendre un FIR passe bas qui coupe à la moitié de 0.5, soit 0.25 pour respecter Shannon.

En utilisant le script Scilab, on obtient les coefficients entiers suivants.

[*-3578, 69, 11139, 17508, 11139, 69, -3578*]

Et on obtient l'image suivante. On peut voir qu'au endroits où la décimation simple ne retenait qu'une partie des informations (les rayures noire et blanche par exemple), la décimation par convolution va tenir compte de toutes les informations de l'image de base et va, dans notre exemple de rayures, colorier l'image en gris.

On remarque dans le code que les coefficients du filtre peuvent être négatifs. Cela ne pose pas de problème en C++ du moment que l'espace mémoire est alloué et qu'on sait ce qu'il y a dedans. En pratique il n'y a pas de différence avec un coefficient positif, on va lire l'espace mémoire situé *n* cases avant la case mémoire du pointeur au lieu de lire *n* cases après.

Il faut cependant faire attention dans le cas d'une écriture à ce que la case mémoire accédée soit bien un espace alloué par le programme et que notre pointeur a le droit d'y accéder.

CLASSE ABSTRAITE

L'implémentation de base de la décimation passait par les fonctions **convolution_horizontal** et **convolution_vertical**. Or le C++ est un langage objet, et il serait donc plus logique de passer par une classe **Decimation** pour réaliser la décimation.

Nous créons la classe **Decimation** abstraite, afin d'utiliser le polymorphique sur les classes dérivées. Nous pourrons ainsi choisir le type de décimation de façon simple.

INSTRUCTIONS VECTORISEE

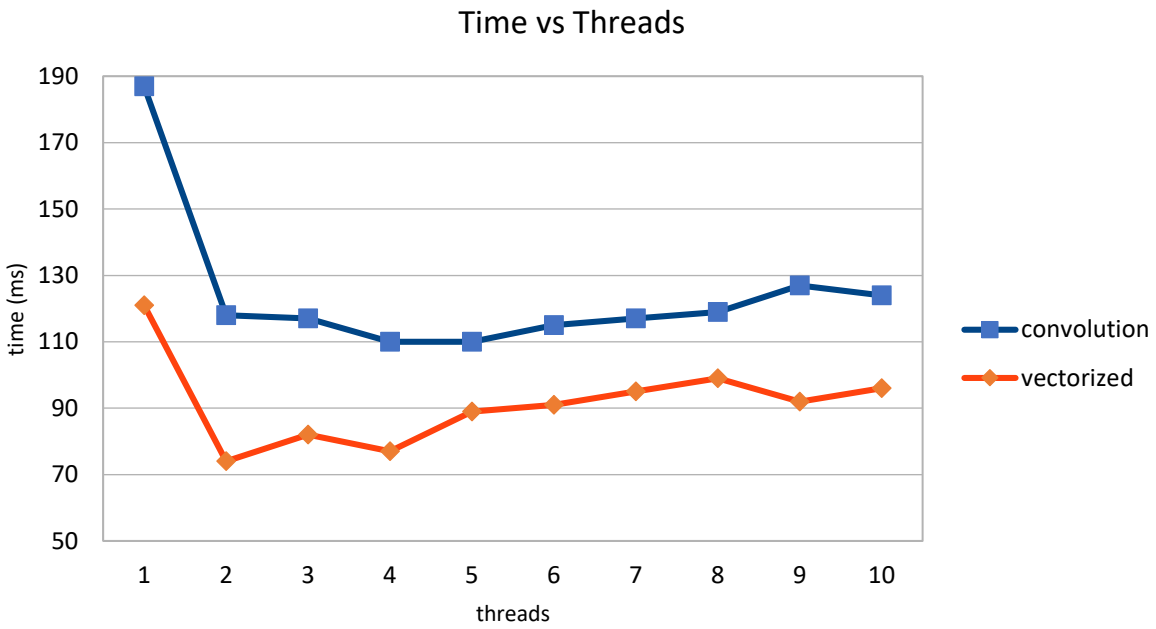
On utilise les instructions vectorisées d'Intel pour optimiser la décimation par convolution.

L'idée est de se débarrasser de la boucle de calcul qui prend le plus de temps d'exécution. On peut voir ci-dessous le profil de la version convolution classique et la version vectorisée en fonction du nombre de threads. Le temps de calcul est diminué de 30% environ sur la version 1 thread.

On remarquera que la version vectorisée est imparfaite ; En effet, la boucle *for* n'a pas complètement disparue dans la partie verticale. Nous n'avons pas trouvé de solution pour charger les données sans

passer par une boucle, vu que les datas sont espacés de *stride* et ne sont pas consécutives dans la mémoire.

C'est pour cela que si on étudie le profil de chaque méthode avec l'outil *gprof*, la partie verticale prend plus de temps que la partie horizontale alors que cette dernière traite plus de données.




```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 58.36      0.07     0.07       30     2.33     2.33  VectorizedDecimation::vertical(unsigned short*, unsigned short*, un
t, unsigned int)
 41.69      0.12     0.05       30     1.67     1.67  VectorizedDecimation::horizontal(unsigned short*, unsigned short*,
int, unsigned int)
```

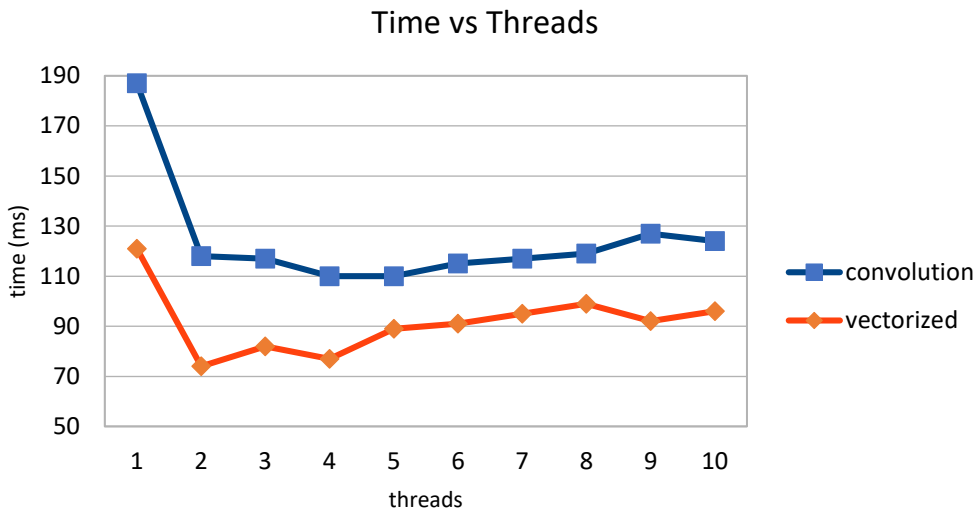
PARALLELISME

Pour la mise en place du parallélisme, nous allons découper notre image en bandes indépendantes. Nous pouvons paralléliser chaque composante, mais comme notre image est au format YUV420, notre vitesse d'exécution sera toujours limitée par le traitement de la composante Y (luminance).

En effet, les deux autres composantes sont sous-échantillonnées par 2 verticalement et horizontalement, leur temps de traitement est donc bien inférieur (environ 4 fois inférieur) au temps de traitement de la luminance. Il n'est donc pas efficace de paralléliser notre programme de cette façon.

Pour la partie verticale, nous découpons aussi notre programme en bandes, de la même façon qu'à l'horizontal.

On rappelle l'évolution du temps de calcul en fonction du nombre de threads.



En théorie notre programme devrait atteindre sa vitesse d'exécution maximale à 4 threads vu que notre machine tourne sur 4 cœurs, mais on remarque que pour le code vectorisé la version 2 threads semble en moyenne plus rapide. On peut voir ci-dessous les caractéristiques de la machine pour confirmer le nombre de cœurs.

```
romain@romain-X751LJ:~$ lscpu
Architecture :                       x86_64
Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
Boutisme :                           Little Endian
Address sizes:                       39 bits physical, 48 bits virtual
Processeur(s) :                      4
Liste de processeur(s) en ligne :    0-3
Thread(s) par cœur :                 2
Cœur(s) par socket :                 2
Socket(s) :                          1
Nœud(s) NUMA :                       1
Identifiant constructeur :           GenuineIntel
Famille de processeur :              6
Modèle :                             61
Nom de modèle :                      Intel(R) Core(TM) i3-5005U CPU @ 2.00GHz
Révision :                           4
```

Pour implémenter correctement le parallélisme, nous allons créer les fonctions **do_horizontal** et **do_vertical** dans la classe **Decimation**. Ces fonctions vont implémenter le multi-thread horizontal et vertical respectivement.

On sépare le parallélisme horizontal et vertical dans 2 fonctions pour assurer que la décimation verticale commence bien lorsque la décimation horizontale est finie.

En effet, si les deux décimations sont dans le même scope, la décimation verticale risque de commencer avant que la décimation horizontale finisse, ce qui va planter le programme.