POLYTECH NICE-SOPHIA

Projet “Filtrage Numérique”

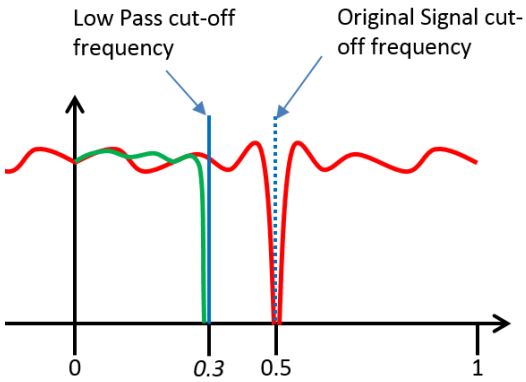
Le but de ce projet est de :

- Développer un filtre numérique avec l’outil `scilab` et coder le filtre en C++ pour réaliser une opération « *downsize* » sur une image 2D.
- Utiliser les instructions vectorielles de votre CPU en utilisant la librairie C++ dite « *intrinsic* » pour améliorer les performances du filtre.
- Améliorer le programme en ajoutant une classe abstraite.
- Paralléliser une fonction avec l’utilisation de la librairie `std::thread` qui permet d’activer les cœurs présents sur le CPU.

1 - Filtrage Numérique

1.1 Rappel sur les FIR

On se limite ici à quelques rappels sur les filtres de type FIR. Un filtre de type « *low-pass* » permet de couper les fréquences hautes comme ci-dessous, avec une fréquence de coupure à 30% de la fréquence d’échantillonnage (f_e). On note qu’initialement, la fréquence maximale des données ne dépasse pas $f_e/2$, soit 0.5 en fréquence normalisée.



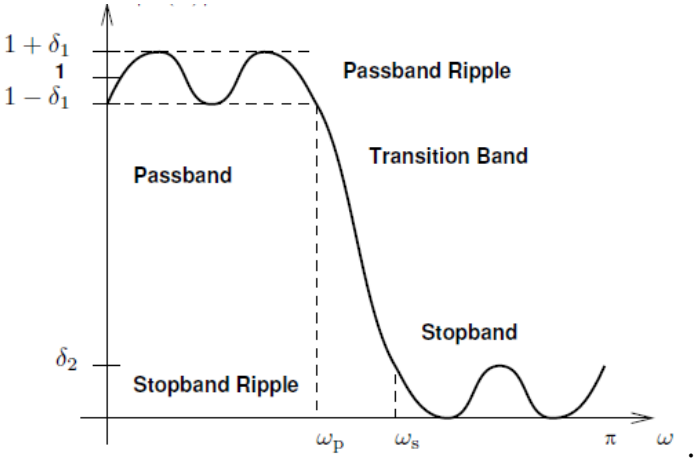
Pour un signal discret, représenté par des valeurs $x[n]$, et un filtre FIR donné par ses constantes $c[k]$, $-M_1 \leq k \leq M_2$, les valeurs filtrées, $y[n]$, sont calculées par convolution :

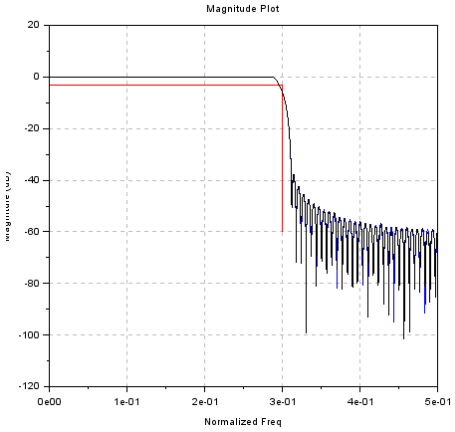
$$y[i] = \sum_{k=-M_1}^{M_2} c[k] \cdot x[i-k]$$

De nombreux outils, comme `mathlab` ou `scilab`, permettent de calculer les coefficients $c[k]$ d’un filtre répondant à un cahier des charges précis : fréquence de coupure, rebond maximum dans la bande de base, atténuation dans la bande filtrée, déphasage, …

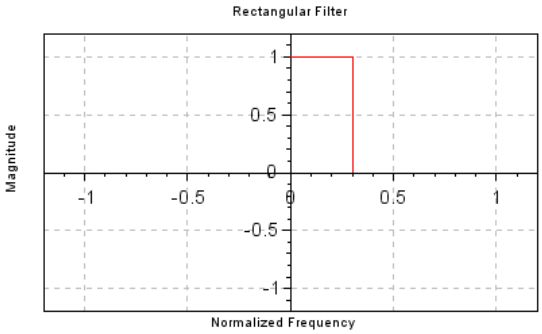
En page suivant, un exemple typique de FIR : à gauche le gabarit du filtre et à droite l’implémentation avec la réponse en fréquence filtre. On note la coupure à 30% de la fréquence d’échantillonnage.

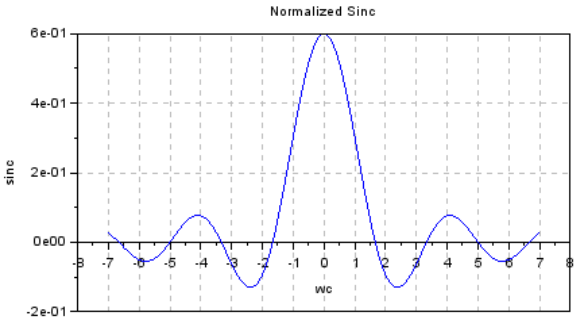
POLYTECH
NICE-SOPHIA



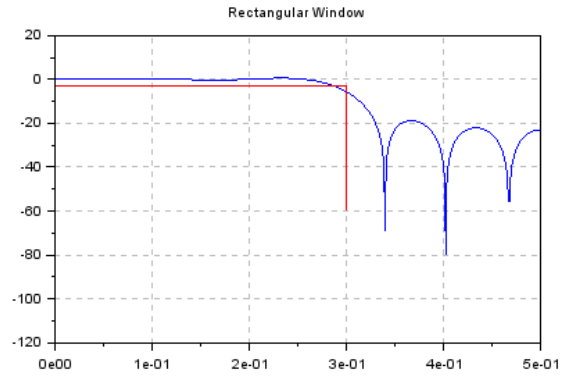


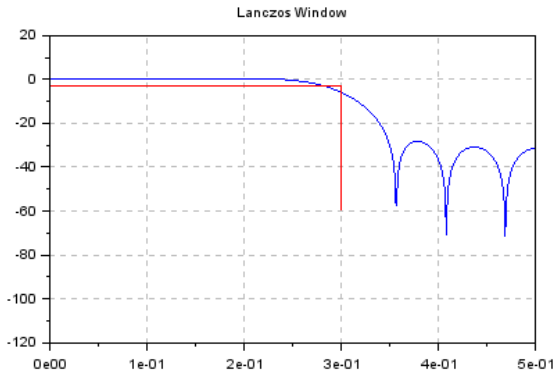
On rappelle qu'un filtre rectangulaire a une réponse impulsionnelle infinie en $\sin(\omega t)\,/\omega t$.





Pour obtenir un filtre de type FIR, il faut appliquer une fenêtre qui permet de ne garder que n coefficients. Un fenêtrage rectangulaire simple ne doit pas être utilisé. On trouve une large variété de fenêtre dans la littérature des FIR : Hamming, Lanczos, Barlett, … La figure ci-dessous compare la réponse en fréquence avec une fenêtre rectangulaire à gauche et une fenêtre Lanczos à droite.





On considère pour la suite les filtres symétriques avec un nombre impaire de coefficients : $n = 2M + 1$, l'équation de convolution du filtre devient :

$$y[i] = \sum_{k=-M}^{M} c[k] \cdot x[i-k]$$

POLYTECH NICE-SOPHIA

2 - Image « *Downsize* »

On cherche à réaliser un programme qui divise par r la hauteur et la largeur de l'image. La méthode choisie est de faire 2 opérations : une réduction de la largeur puis une réduction de la hauteur. Pour simplifier le TP, on impose $r = 1/2$. On divisera donc la taille de l'image par un facteur 4 :

2.1 Première étape

Vous devez installer l'archive fournie, puis compilez et vérifiez que votre résultat sur la console est identique à celui ci-dessous :

```
shell> downsize ../images/BigBuckBunny_1920x1080.yuv
#########################################################
##        YUV Image Decimation  version: base          ##
#########################################################
Info: loading image from ../images/BigBuckBunny_1920x1080.yuv
Info: initial image size   : 1920x1080
Info: downsized image size : 960x540
Info: downsized image calculated in 65ms, using 1 thread(s)
Info: downsized image stored in file downsize.yuv
```

Le temps d'exécution est directement lié à puissance de votre laptop.

POLYTECH NICE-SOPHIA

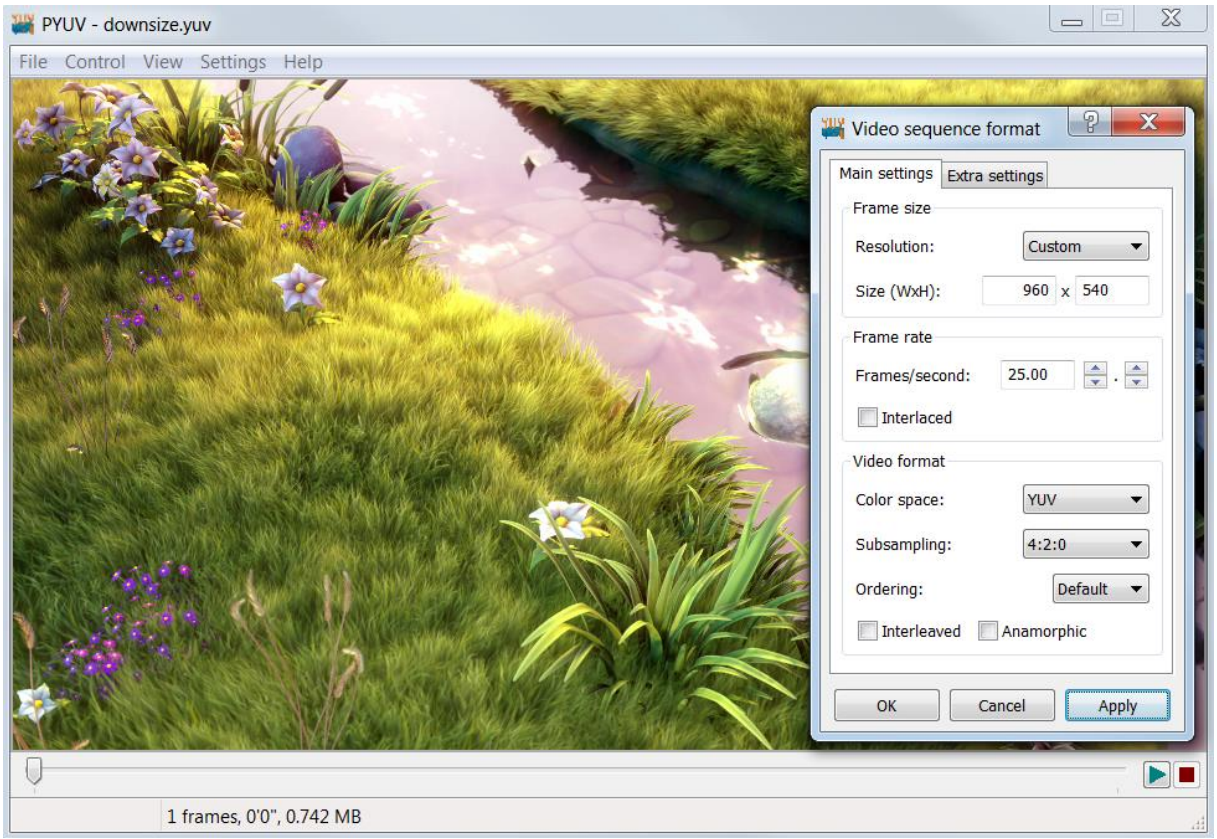
2.1.1 Voir des images YUV

Le projet QT `yuv-viewer-base` fourni dans l'archive vous permettra de visualiser les images au format yuv.

D'autres alternatives sont possibles et disponibles sur le web avec un peu de recherche. J'ai sélectionné pour vous : PUYV
voir http://dsplab.diei.unipg.it/software/pyuv_raw_video_sequence_player

Vous pouvez télécharger les versions Windows, Mac ou Linux en cliquant sur « download »

Lorsque vous charger le fichier `download.yuv`, bien renseigner les champs de résolution et les de format video, sinon vous obtenez une image noire.



2.2 Analyse du code existant

Le cœur du programme est situé dans le fichier `simple_decimation.cpp`.
Comprendre l'algorithme utilisé. Est-il satisfaisant ?
Pouvez-vous expliquer le résultat sur le fichier `mire1024x768.yuv` ?

Le signe Q dans la marge indique une question dont la réponse devra se trouver dans votre rapport.

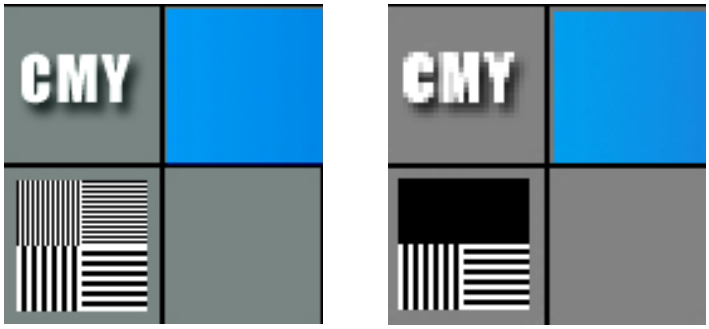
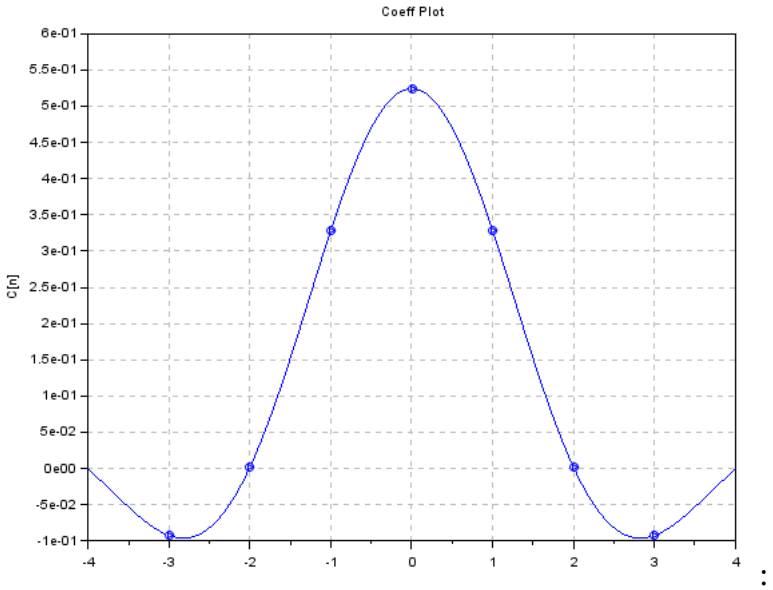


image originale image réduite

2.3 Amélioration

Il faut réaliser un filtrage des données sur chaque line avant de procéder à la décimation horizontale, puis un filtrage sur chaque colonne avant la décimation verticale. Pourquoi ?
Que vaut la fréquence de coupure de votre filtre ?

Modifiez le script `scilab` pour concevoir un filtre passe-bas avec une fréquence de coupure choisie et 7 coefficients. Vous devriez obtenir une courbe assez similaire à la courbe suivante



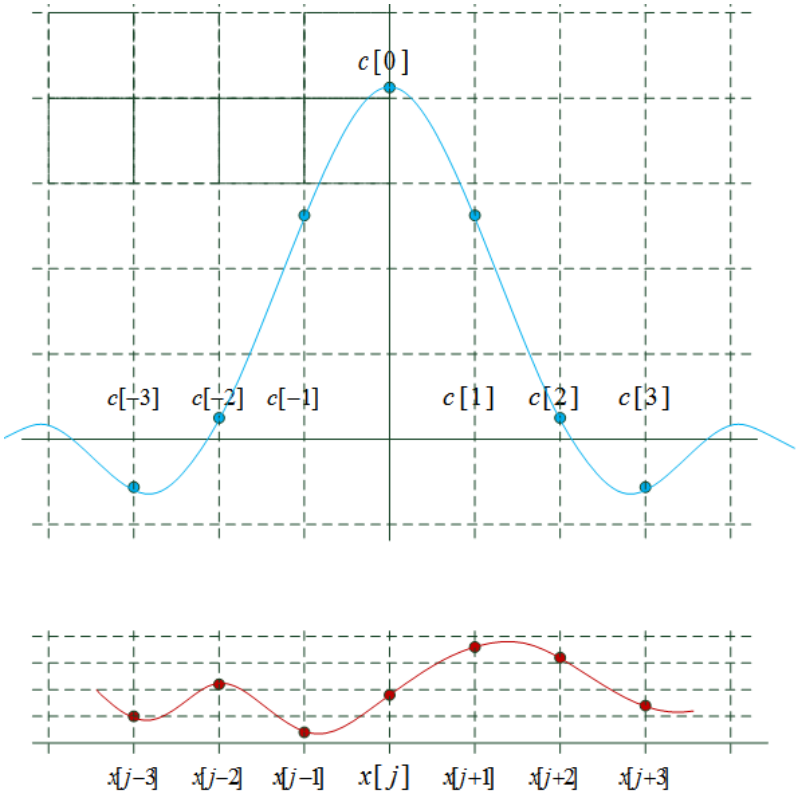
:

La console `scilab` affiche aussi les coefficients du filtre en valeurs flottantes et en valeurs entières arrondies. On utilisera les valeurs entières.

POLYTECH NICE-SOPHIA

2.4 Convolution numérique

Le schéma ci-dessous permet de bien comprendre la convolution numérique entre les coefficients du filtre et les données (dans notre cas c'est l'intensité lumineuse de chaque pixel).




```
x[-3] =  5        c[-3] = -0.0789
x[-2] = 11        c[-2] =  0.0090
x[-1] =  2        c[-1] =  0.3172
x[0]  =  9        c[0]  =  0.5054
x[1]  = 18        c[1]  =  0.3172
x[2]  = 16        c[2]  =  0.0090
x[3]  =  7        c[3]  = -0.0789
```

after convolution (**c*x**), we have
y[0] =10.19

Pour éviter un calcul en nombre flottant, on utilise une approximation entière des coefficients et dans notre cas, les $c[k]$ sont tels que :

$$\sum_{k=-3}^{3} c[k] = 2^{15}$$

et donc la convolution devient

$$y[i] = \sum_{k=-3}^{3} c[k] \cdot x[i-k] \Big/ 2^{15}$$

POLYTECH NICE-SOPHIA

2.5 A faire

Modifier les fichiers `decimation.h` et `decimation_convolution.cpp` pour coder cette opération de convolution entière
Testez votre programme.
Regardez les effets sur la mire. Expliquez ?

image originale image réduite (simple) image réduite (convolution)

2.6 Pour Vous Aider

On rappelle que pour faire une division entière par une puissance de 2, il est plus rapide d'utiliser un décalage arithmétique vers la droite, mais il ne faut pas oublier la gestion de l'arrondi. La formule est donc :

$$\frac{n}{2^q} = (n + 2^{q-1}) \gg q$$

Je trouve dans mon code :

```
int sum = round;
for (int i = -3; i <= 3; ++i) {
  int idx = k + i;
  idx =                                                    ; // to be completed
  int32_t data = static_cast<int32_t>(src_col0[idx]);
  sum += data * fir_coeff[i];
}
...
```

On remarque que les indexes des coefficients du filtre peuvent être négatifs. Est-ce une erreur de codage ou bien légal en C++ ? Quelle case mémoire lit-on ?

POLYTECH NICE-SOPHIA

2.7 Complément : Condition aux limites

Lorsque l'on calcule une convolution, il faut faire attention aux conditions limites. Dans notre cas, cela va être les bords de l'image.

Dans le code ci-dessus, applicable en décimation horizontale, on regarde le cas `k=0`, on remarque alors que `src_col0[idx]`, vaut `src_col0[-3]`. Mais, cela n'a pas de sens d'accéder au pixel en colonne -3.
Il faut donc prendre une hypothèse sur ces pixels en dehors de l'image : vous utiliserez une extension simple de duplication du pixel en bord d'image.

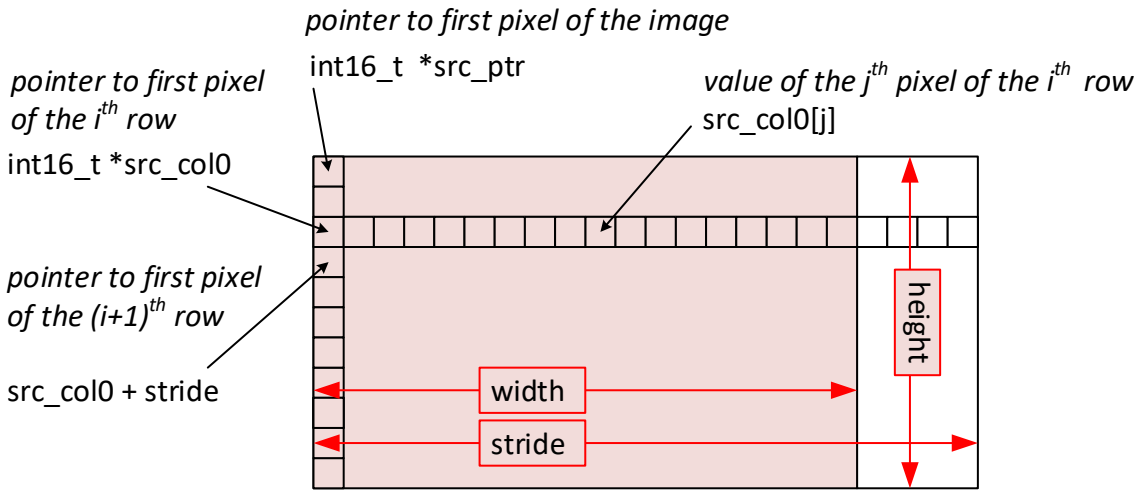
Cette extension est applicable pour tous les bords : gauche, droite en décimation horizontale et haut, bas en décimation verticale : pour le bord

L'implémentation est assez facile avec une ligne, c'est d'ailleurs celle marquée « *to be completed* » dans le code ci-dessus. Exemple pour le bord gauche et droit :

Bord Gauche	Bord Droit
`src_col0[-1] = src_col0[0];`	`src_col0[w]   = src_col0[w-1];`
`src_col0[-2] = src_col0[0];`	`src_col0[w+1] = src_col0[w-1];`
`src_col0[-3] = src_col0[0];`	`src_col0[w+2] = src_col0[w-1];`

2.8 Complément : format de l'image

Le diagramme ci-dessous vous aide à comprendre l'organisation mémoire pour une composante de l'image.
Vous remarquez que 4 pixels sont ajoutés en fin de chaque ligne, c'est pour cela que le passage d'une ligne à l'autre utilise la variable `stride` plutôt que la variable `width`. C'est une technique classique qui permet par exemple de garantir l'alignement des pointers `src_col0` sur un multiple de 16 ou 32.



POLYTECH NICE-SOPHIA

3 - Classe abstraite

3.1 Introduction

J'ai trouvé le code ci-dessous dans le fichier `decimation_simple.h` livré dans l'archive :

```
extern void convolution_horizontal(
            uint16_t *dst_ptr, uint16_t *src_ptr,
            uint dst_width, uint dst_height,
            uint dst_stride, uint src_stride);
extern void convolution_vertical(
            uint16_t *dst_ptr, uint16_t *src_ptr,
            uint dst_width, uint dst_height,
            uint dst_stride, uint src_stride);
```

Q Pourquoi ces lignes ont-elles le tampon « WTF » ?

En lisant tout le TP, vous constatez qu'il y a 3 implémentations possibles pour faire une décimation :

1. Méthode simple, donnée en exemple,
2. Méthode avec convolution,
3. Méthode avec instructions vectorisées.

Vous pouvez donc modifier `main.cpp` et ajouter une option `--method` qui permet de choisir l'une des 3 méthodes ci-dessus.

```
shell> downsize -help
#########################################################
##        YUV Image Decimation  (version student)    ##
#########################################################

Usage: downsampling[options] [integer]
Options:
-h | --help         Display this help
--width integer     Width of the image in pixel (default: auto)
--height integer    Height of the image in pixel (default: auto)
--percent integer   Resize percentage between 10 to 100 (default: 50)
--output filename   Name of the output file (default: downsize.yuv)
--threads integer   Number of threads (default: 1)
--method string     Choices: simple, convolution or vectorized (default: simple)
```

Il faut maintenant coder des classes qui implémentent les fonctions de décimations correspondant à l'option donnée en argument.
On est typiquement dans le cas d'une classe abstraite et de classes dérivées.

POLYTECH NICE-SOPHIA

3.2 Conception de la classe abstraite

Il faut créer une classe `Decimation` (classe abstraite). Il faut pour cela faire quelques modifications dans le fichier `decimation.h`, à vous de trouver…

```
        void horizontal(uint16_t *src_ptr, uint16_t *dst_ptr, uint total_rows);
        void vertical(uint16_t *src_ptr, uint16_t *dst_ptr, uint total_cols);
```

Il faut maintenant ajouter des classes dérivées. Pour vous aider, je trouve dans mon code :

```
class SimpleDecimation : public Decimation {
 public:
  explicit SimpleDecimation(const uint nb_thread) : Decimation{nb_thread} {
  }
  void horizontal ...;
  void vertical ....;
};

class ConvolutionDecimation ...  {
...
};
```

Vous devez aussi utiliser un pointer polymorphique dans le fichier `main.cpp`. Je vous laisse trouver ce qui faut faire.

Pour mémoire, le pointeur polymorphique est un pointeur sur un objet d'une classe dérivée qui se fait passer pour un pointer de la classe de base :

```
Decimation *poly_ptr = new SimpleDecimation(…) ;
```

Ce pointer permet d'appeler la bonne méthode, par exemple :

```
poly_ptr->horizontal(…);
```

Il appelle la méthode : `SimpleDecimation::horizontal(…) ;`

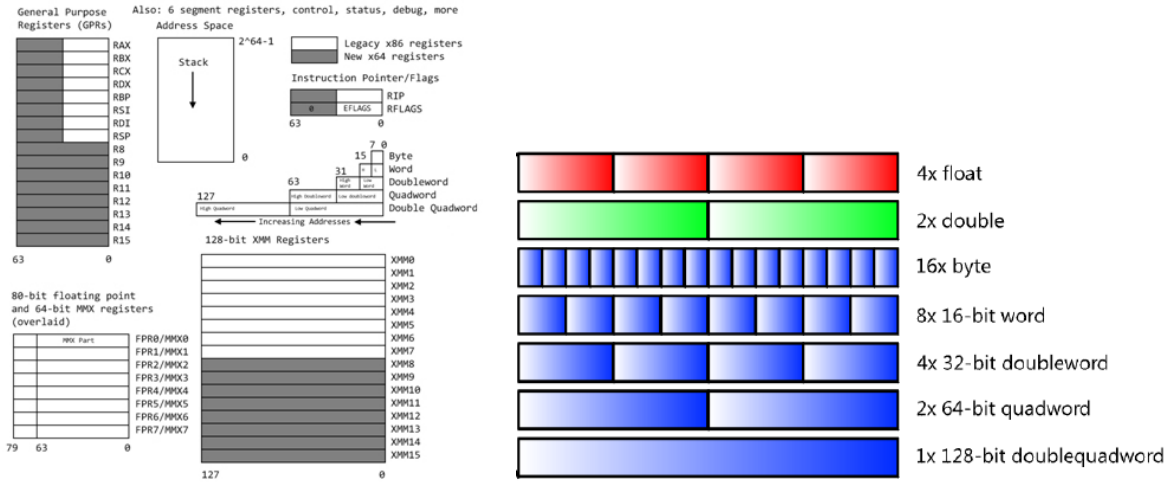
L'option `-method` permettra de choisir l'une des 3 classes dérivées.

POLYTECH NICE-SOPHIA

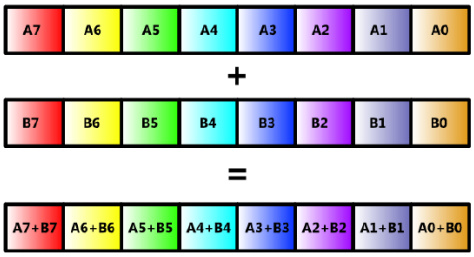
4 - Utilisation des instructions vectorisées

Tous les micro-processeurs modernes ont des instructions de type SIMD (« *single instruction multiple data* » (AVX chez Intel, Neon chez ARM).

En fonction des architectures, Intel offre 16 ou 32 registres SIMD de 128, 256 ou 512 bits. Ce sont les registres XMM de la figure de gauche ci-dessous. A droite, on donne le détail d'un registre 128 bits, qui peut stocker de multiple format : 16 entiers sur 8 bit ou 8 entiers sur 16 bit ou bien 4 entier sur 32 bit…



Intel a développé des fonctions compatibles avec le langage C++ permettant une utilisation facile des instructions SIMD : c'est la technologie « *intrinsic* ».
Par exemple, pour ajouter 2 registres SIMD de 128 bits contenant 8 entiers codés sur 16 bits, comme ci-dessous :



On écrit :

```
__m128i r128_a = ...;  // voir en page suivante pour savoir comment
__m128i r128_b = ...;  // initialialiser des registres de 128 bits

__m128i r128_r = _mm_add_epi16(r128_a, r128_b);
```

On remarque l'introduction d'un nouveau type, `__m128i`, qui est définie par Intel.
Les noms de variables `r128_a`, `r128_b`, … permettent d'identifier facilement les variables associées avec les registres SIMD.

POLYTECH NICE-SOPHIA

4.1 Introduction

Nous allons chercher à remplacer la boucle de calcule de convolution par quelques instructions SIMD.

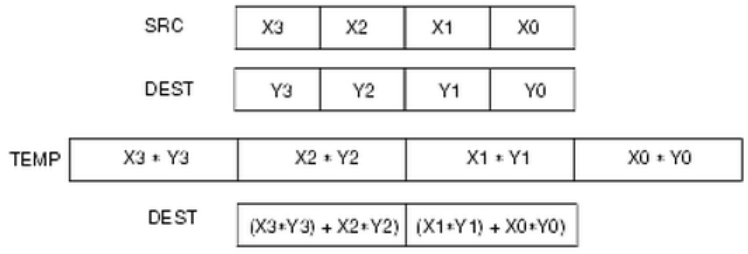
```
int sum = 0;
for (int i = -3; i <= 3; ++i) {
  sum += data[k+i] * fir_coeff[i];
}
```

L'instruction _mm_loadu_si128 permet de charger un registre 128 bits avec le contenu d'une zone mémoire consécutive de 16 octets. Si on représente les 7 coefficients du filtre avec des entiers sur 16 bits, on peut donc charger en une instruction ces coefficients dans un registre.

La même instruction permet aussi de charger les données. On a donc après chargement dans 2 registres :

r128_coeff	0	-3026	64	10758	17176	10758	64	-3026
r128_data	d[7]	d[6]	d[5]	d[4]	d[3]	d[2]	d[1]	d[0]

L'instruction _mm_madd_epi16 permet de multiplier 2 registres de 128 bits (organisés en 8 mots de 16 bits), comme suit (diagramme sur les parties basses du registre) :



On fait donc en une instruction un produit vectoriel et une somme partielle.

Il faut maintenant additionner les 4 mots de 32 bits du registre résultat.
Il n'y a pas d'instruction qui permet de faire ceci directement mais en lisant bien la documentation Intel, on trouve l'instruction _mm_hadd_epi32 qui permet de calculer une somme de registres de la façon suivante :

	127 … 96	95 … 64	63 … 32	31 … 0
r128_a	a[3]	a[2]	a[1]	a[0]
r128_b	b[3]	b[2]	b[1]	b[0]
r128_r	b[3] + b[2]	b[1] + b[0]	a[3] + a[2]	a[1] + a[0]

POLYTECH NICE-SOPHIA

4.2 Alignement

Les 2 instructions `_mm_load_si128(__m128i const* mem_addr)` et `_mm_loadu_si128(__m128i const* mem_addr)` ne diffèrent que par des contraintes d'alignement. La première instruction exige que le pointeur de donné soit un multiple de 16, ce qui permet d'optimiser le chargement des lignes de cache. Vous devez donc vous assurez que les données sont bien alignées sur un multiple de 16.
Vous avez besoin d'utiliser d'attributs spécifiques qui indique au compilateur où mettre les données en mémoire. La macro `ALIGN32` dans le fichier `decimation_vectorized.cpp` permet de faire ceci simplement :

```
static int16_t ALIGN32(mem_fir_coeff[]) = { 0, … };
```

A vous d'utiliser cette macro, si besoin.

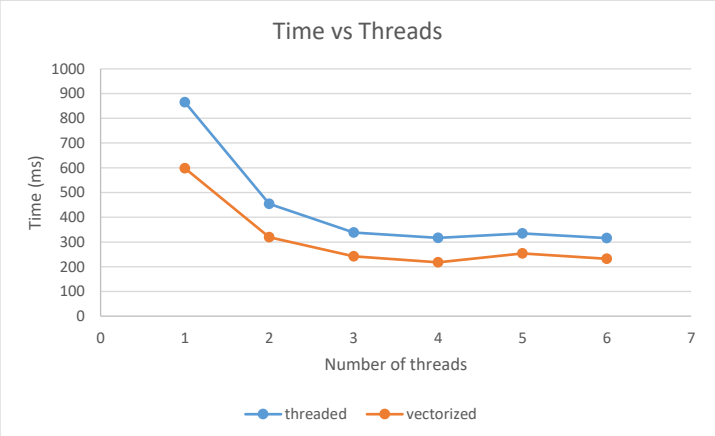
4.3 A faire

Vous avez toutes les informations pour terminer votre programme…
Pour vous aider, voici la liste de toutes les instructions SIMD que je trouve dans mon programme :

```
_mm_extract_epi32
_mm_hadd_epi32
_mm_insert_epi16
_mm_load_si128
_mm_loadu_si128
_mm_madd_epi16
_mm_set1_epi32
_mm_shuffle_epi8
```

Comparez les performances entre la version normale et la version vectorisée.

Pour ma part, j'obtiens les courbes suivantes, avec une diminution du temps de calcul de 30% environ.



POLYTECH NICE-SOPHIA

5 - Amélioration des performances

5.1 Votre machine

Vous devez trouver pour votre ordinateur les éléments suivants (avec par exemple CPUZ)

Nom du Processeur	*Intel Core i7 4800MQ*
Nombre de cœurs	*4*
Nombre de threads	*8*
Fréquence de base	*2.7GHz*
Instructions SSE	*SSE 4.1 / 4.2, AVX 2.0*

5.2 En Parallèle avec les « *threads* »

A lire : https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial.

On peut vite perdre énormément de temps à optimiser une méthode, il faut donc s'assurer que la fonction choisie représente une part significative du temps total (voir loi de Amdahl)

On fait l'hypothèse que les fonctions de décimation représentent bien la majorité du temps passé. Pour mémoire, les boucles principales font une itération sur tous les pixels de l'image

J'ai confirmé cette hypothèse avec l'outil `gprof` qui calcule le temps d'exécution de chaque fonction. Voici ci-dessous le résumé du rapport `gprof`.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds   name
 56.32      0.49     0.49   vertical(uint16_t*, uint16_t*, uint, ...)
 41.38      0.85     0.36   horizontal(uint16_t*, uint16_t*, uint, ...)
  1.15      0.86     0.01   YuvImage::write(std::string const&)
  1.15      0.87     0.01   YuvImage::YuvImage(std::string const&, uint, uint)
```

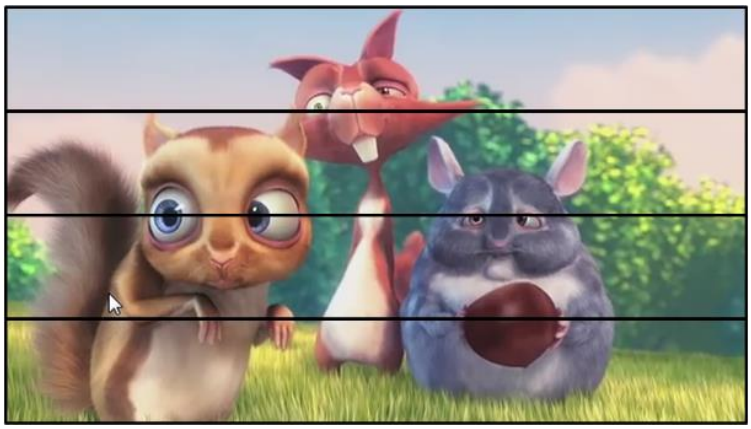
Pourquoi le temps de décimation vertical est-il plus grand que le temps de décimation horizontal alors que la taille de l'image source à parcourir est inférieure de moitié ?

5.3 Méthode à suivre

La mise au point de programme « *multi-thread* » devient vite difficile. Il faut donc procéder pas à pas
- Identification du parallélisme
- Modification et test avec un seul « *thread* ».
- Modification (qui doit être mineur) pour exécution avec des « *threads* ». Le programme doit facilement pouvoir s'exécuter en version « *single-thread* »

5.3.1 Identification du parallélisme

La décimation horizontale peut être parallélisée assez facilement en découpant l'image en *n* bandes horizontales.



Une autre possibilité consiste à paralléliser chaque composante :



Quelle méthode allez-vous choisir ? Rappel le format d'image est YUV420.

POLYTECH NICE-SOPHIA

5.3.2 Développement « *single thread* »

Il faut modifier la méthode `Decimation::do_horizontal()` pour appeler *n* fois la fonction `horizontal(...)` sur des bandes successives de l'image.

Testez et valider.

Faire de même avec la décimation verticale, méthode `Decimation::do_vertical()`.

Comment allez-vous découper l'image dans le cas vertical ?

5.3.3 Ajout du « *multi-thread* »

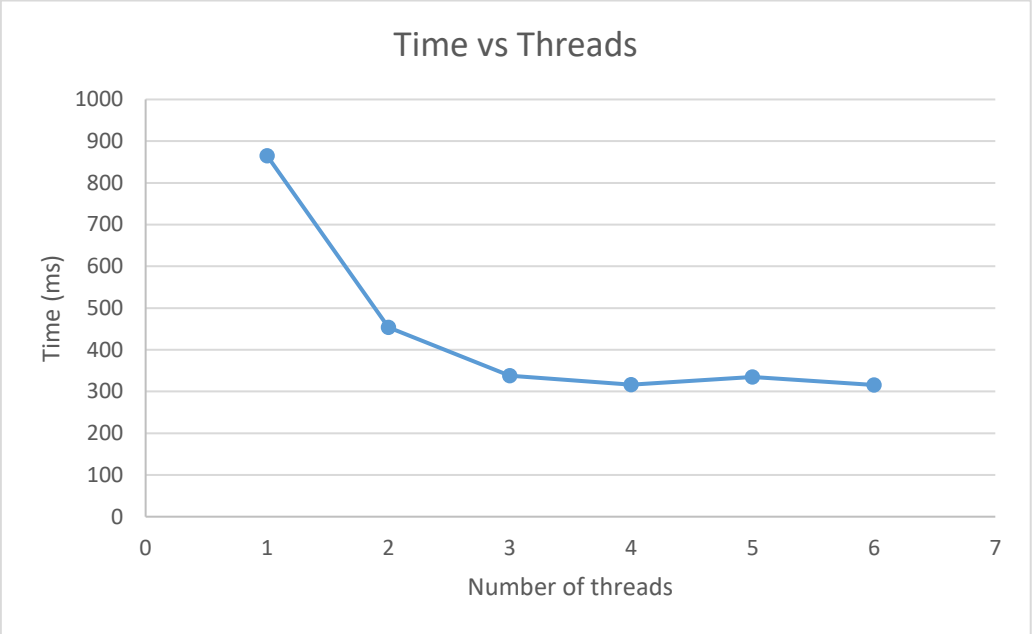
La mise en place des « threads » se fait ensuite facilement, vous pouvez vous inspirant de l'exemple suivant. N'oubliez pas un objectif simple : lorsque vous développez une application « *multi-thread* », il est important de pouvoir repasser en version avec un thread très facilement et donc ici en mettant la variable `max_thread` à 1 sans autre changement dans le code.

```
std::vector<std::thread> threads;

for (i = 0; i < (nb_thread-1); ++i) {
  threads.emplace_back([=]() {
    my_parallel_function(i, max_thread);
  });
}
my_parallel_function(max_thread-1, max_thread);

for (auto &thread_elem : threads) {
  thread_elem.join();
}
```

Faire varier le nombre de threads et tracez une courbe : temps d'exécution en fonction du nombre de threads. Commentez !

J’obtiens ce type de courbe.

Pourquoi ne voit-on pas d’amélioration sensible au-delà de 4 « *threads* ».



POLYTECH NICE-SOPHIA

6 - Livrables

Comme d'habitude :

Une archive avec votre code source
- decimation.h
- decimation_convolution.cpp
- decimation_simple.cpp
- decimation_vectorized.cpp
- filter_design_7tap.sce
- main.cpp
- Makefile
- util.h
- yuvimage.cpp
- yuvimage.h

Un rapport expliquant les différents choix, et les réponses aux questions avec des captures d'écran montrant les améliorations successives.

Question optionnelle (mais vivement recommandée) :
Faire tourner le code sur l'image `TallBuilding_3840x2160.yuv` et `mire1024x768.yuv`.
Notez rapport de taille entre les 2 fichiers et le nombre d'itérations que fait le programme :
12441600/1179648=10.5469
Le nombre d'itérations permet de s'assurer que le volume de données à lire et écrire est identique dans les 2 cas.

Notez ensuite les 2 temps d'exécution : `Tall`=4685ms, `mire`=857ms

Expliquez pourquoi le temps d'exécution est 5.4 fois plus lent pour faire 100 itérations sur l'image `Tall` par rapport à faire 1054 itérations sur l'image `mire`, alors que le volume de donnée est identique ?

POLYTECH NICE-SOPHIA

7 - Références

[1] Introduction to Intel® Advanced Vector Extensions
https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf

[2] Downsampling, Upsampling, and Reconstruction
http://www.cppsim.com/BasicCommLectures/lec10.pdf

[3] FIR introduction
http://www.mikroe.com/chapters/view/72/chapter-2-fir-filters/

[4] Design of Digital Filter
http://web.eecs.umich.edu/~fessler/course/451/l/pdf/c8.pdf