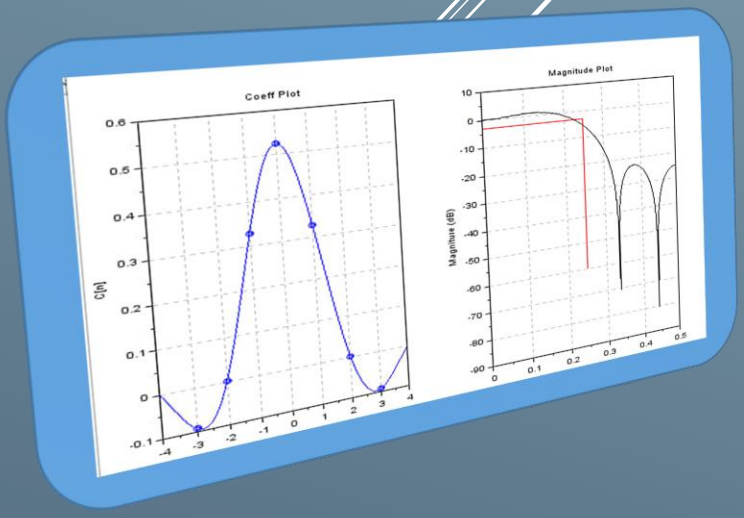
PROJET C++ POUR L'EMBARQUE
"FILTRAGE NUMERIQUE"



GUO Ran

BARRIGA Ricardo

Professeur Bernard PLESSIER

Elec4 2019-2020

10/04/2020

Sommaire

1) Introduction : _____ *2*

*2) Image « Downsize » :*_____ *2*

2.1) Première étape :_____ **2**

2.2) Analyse du code existant : _____ **3**

2.3) Amélioration :_____ **5**

2.5) A faire_____ **5**

*3) Classe abstraite*_____ *7*

3.1) Introduction _____ **7**

3.2) Conception de la classe abstraite _____ **9**

4) Utilisation des instructions vectorisées _____ **9**

*5)Amélioration des performances*_____ *11*

1) Introduction :

Le but de ce projet est de développer un filtre numérique pour réaliser une opération « downsize » sur une image 2D. Aussi on veut améliorer les performances du filtre avec les instructions vectorielles du CPU. On veut améliorer le programme en ajoutant une classe abstraite. En plus on va utiliser la librairies std ::thread pour activer les cœurs présents sur le CPU et paralléliser les fonctions.

2) Image « Downsize » :

2.1) Première étape :

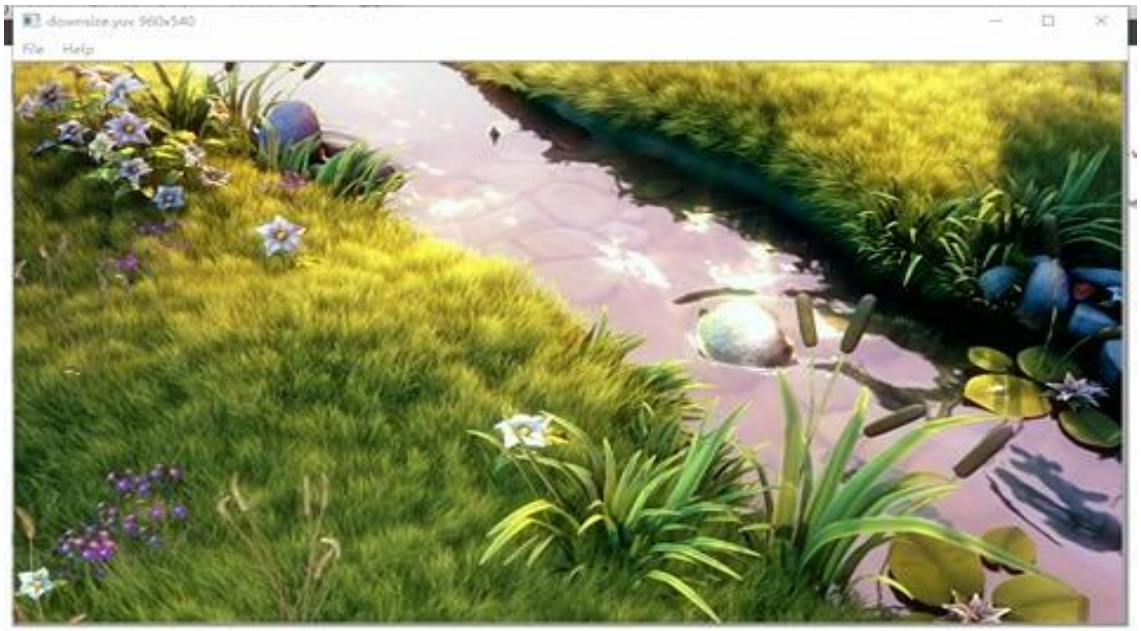
On installe YUV et on utilise le Makefile avec la décimation simple. On exécute le programme « downsize » sur l'image BigBuckBunny_1920x1080.yuv. On obtient l'image ci-dessous :

```
admin@DESKTOP-PCUVUF4 MSYS ~/test/downsampling-base
$ ./downsize ../images/BigBuckBunny_1920x1080.yuv
############################################################
##        YUV Image Decimation  version: base        ##
############################################################
Info: loading image from ../images/BigBuckBunny_1920x1080.yuv
Info:downsized image used method decimation simple
Info: initial image size    : 1920x1080
Info: downsized image size : 960x540
Info: downsized image calculated in 10ms, using 1 thread(s)
Info: downsized image stored in file downsize.yuv
```

L'image a été calculé en 10ms.Par rapport au sujet on a une meilleur qualité parce que cela dépend des caractéristiques de l'ordinateur. On peut voir aussi qu'on a utilisé 1 thread.

Après la compilation on trouve le fichier downsize.yuv qui contient l'image décimé et qu'on peut ouvrir le logiciel Pyuv.

On ouvre PYUV et on insère les caractéristiques nécessaires comme la résolution et le format vidéo de 4 :2 :0. On obtient l'image décimé ci-dessous :



2.2) Analyse du code existant :

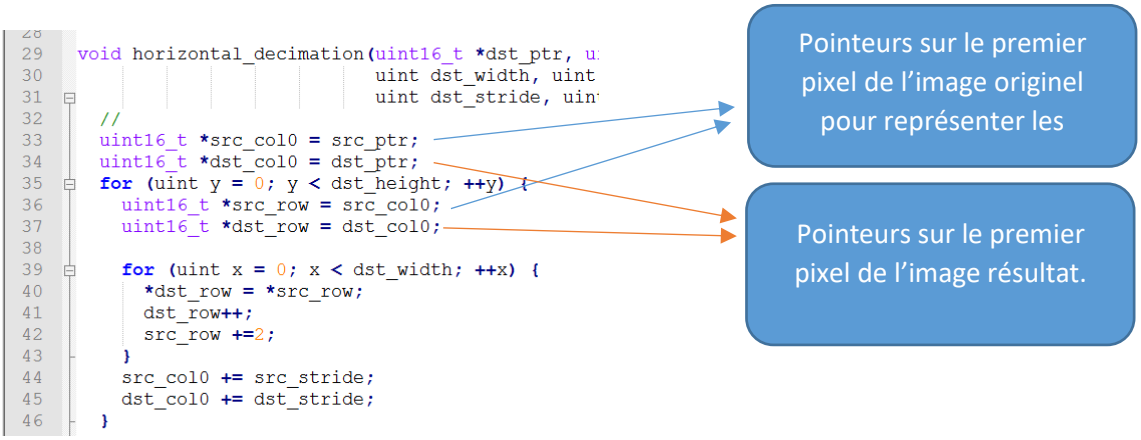
Dans cette partie on va effectuer une décimation simple de l'image. Nous avons plusieurs fichiers fournis. On trouve le cœur du programme dans le fichier decimation_simple.cpp. On voit que ce fichier fait appel à 2 fichiers : util.h et decimation.h. On trouve 2 fonctions pour faire la décimation horizontale et la décimation verticale de l'image :

```
1.  virtual void horizontal_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
2.                                     uint dst_width, uint dst_height,
3.                                     uint dst_stride, uint src_stride);
4.  virtual void vertical_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
5.                                   uint dst_width, uint dst_height,
6.                                   uint dst_stride, uint src_stride);
```

Dans ce prototype on trouve les paramètres :

- *src_ptr qui est un pointeur non signé de 16 bits sur l'image source (sur un pixel)
- *dst_ptr qui est un pointeur non signé de 16 bits sur un pixel de l'image destination (image décimé)
- dst_height qui est une variable non signé entier qui garde la taille de l'image destination
- dst_width qui est une variable non signé entier qui garde le largueur de l'image destination
- src_stride qui est une variable non signé entier qui va servir de foulée entre deux pixels et garder les pixels en multiple de 16.

Ensuite on regarde le corps des fonctions et on voit qu'on a 2 pointeurs :



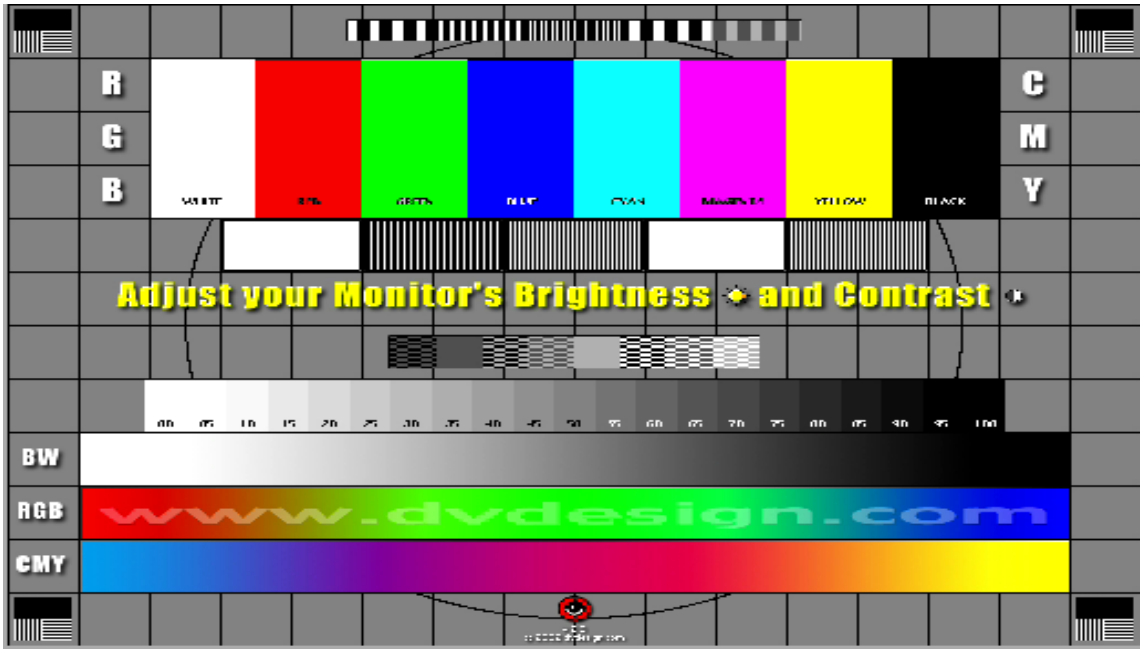
Comprendre l'algorithme utilisé. Est-il satisfaisant ?

On peut voir dans le code ci-dessus que la première boucle for nous permet de parcourir la hauteur de l'image qui est connue. Pour chaque itération on se place avec le pointeur src_row et src_col0 au premier pixel de l'ième ligne courant. On trouve une deuxième boucle for qui va nous permettre de parcourir la largeur du rectangle. En sortant de cette boucle on trouve la fin de la ligne et on rajoute la valeur stride au pointeur src_col0 pour passer à la ligne d'après.

Dans la boucle interne, tant que notre compteur x est inférieur à la largeur de l'image décimée, le pixel de cet image qui est modélisé par la valeur*dst_row reçoit 1 pixel sur 2 de l'image source. C'est ainsi qu'on divise la longueur par 2. Pour cela le pointeur de l'image source est incrémente de 2 alors que celui de l'image est augmenté de 1. On fera pareil pour le vertical. Une fois toutes les lignes balayées on sort de la boucle et de la fonction et on passe à la décimation verticale.

Pouvez-vous expliquer le résultat sur le fichier mire1024x768.yuv ?

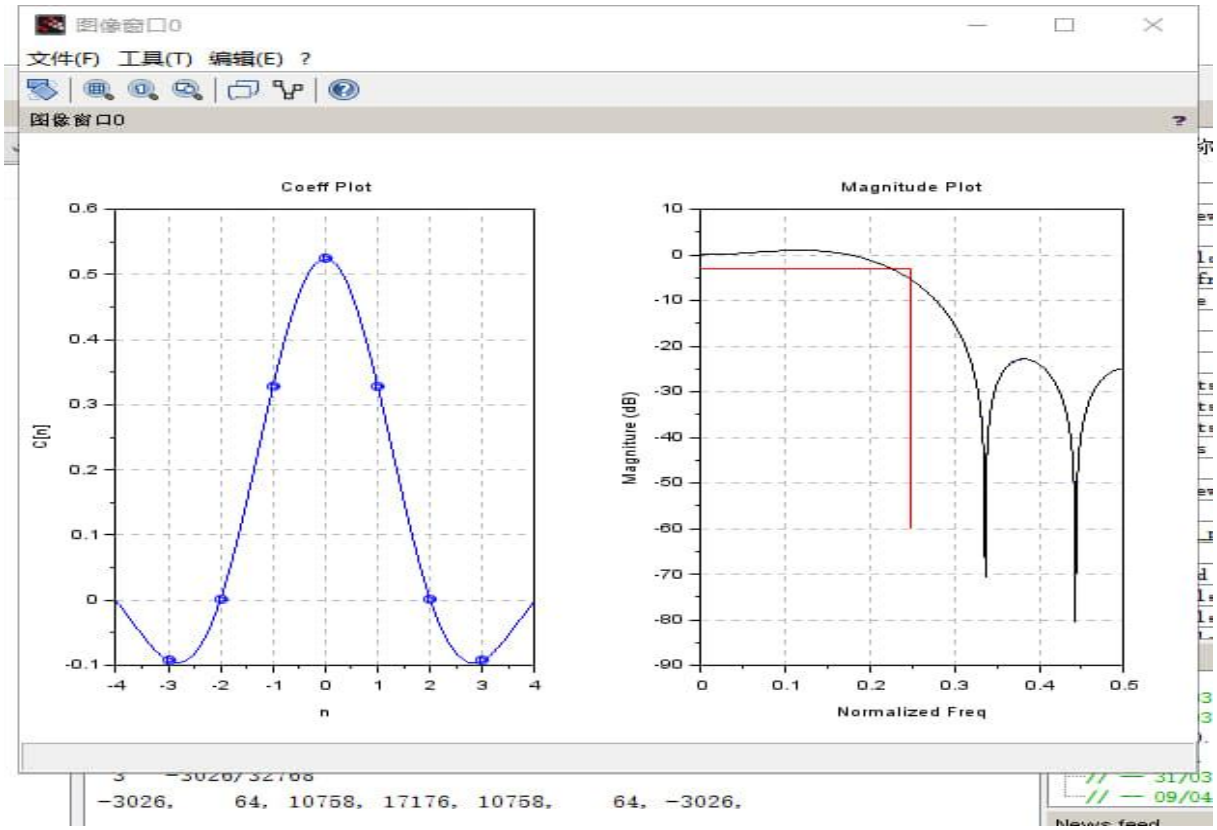
Ce code n'est pas satisfaisant en termes de mémoire et en termes d'efficacité. Apr€`s avoir appliqué la décimation dans l'image mire1024x268.yuv on constate une dégradation importante de l'image en raison du fait qu'on prend un pixel sur deux. Donc deux zones rayées sur les quatre sont maintenant noirs. Le problème est que nous n'avons pas adapté la fréquence d'échantillonnage à certaines zones de l'image. Le sous-échantillonnage provoque un recouvrement spectral. Pour cela on trouve que ce programme n'est pas satisfaisant. On peut voir ci-dessous l'image originale et ensuite l'image réduite :



2.3) Amélioration :

Le filtrage consiste à garantir la qualité de l'image après décimation. Le filtrage consiste à lisser l'image. Dans une image, des hautes fréquences sont caractérisés par des changements brusques. Dans notre cas, on a des zones où l'image a des fortes variations, notamment les zones rayées où on a obtenu des blocs noires après avoir fait la décimation. Pour éviter cela on veut appliquer un filtre sur chaque ligne et sur chaque colonne pour ne pas laisser passer les fréquences où on a des fortes variations, et ensuite on fera la décimation.

On utilise le script scilab pour concevoir un filtre passe-bas avec une fréquence de coupure de 0.249 normalisé. On obtient les graphiques suivantes :



2.5) A faire :

On a modifié le fichier decimation.h et decimation_convolution.cpp pour coder cette opération de convolution entière.

```
//
int fircoeff[7]={-3026,64,10758,17176,10758,64,-3026};
uint16_t *src_col0 = src_ptr;
uint16_t *dst_col0 = dst_ptr;
for (uint y = 0; y < dst_height; ++y) {

  uint16_t *src_row = src_col0;
  uint16_t *dst_row = dst_col0;

  for (uint x = 0; x < dst_width; ++x) {
      *dst_row = *src_row;
  //filtrage convolution
      int sum=0;
      int i=x;
      for(int k=-3;k<=3;k++){
      int idx=i-k;
      //condition aux limites
      if (idx<0) idx=0; //gouche
      if (idx>static_cast<int> (dst_width))
          idx=dst_width-1;//droit

      int32_t data=static_cast<int32_t>(src_row[idx]);
      sum+=data*fircoeff[k+3];
      }
      dst_row[x]=(sum+16384)>>15;
  //decimation
    dst_row++;
    src_row +=2;
  }
  src_col0 += src_stride;
  dst_col0 += dst_stride;
}
```

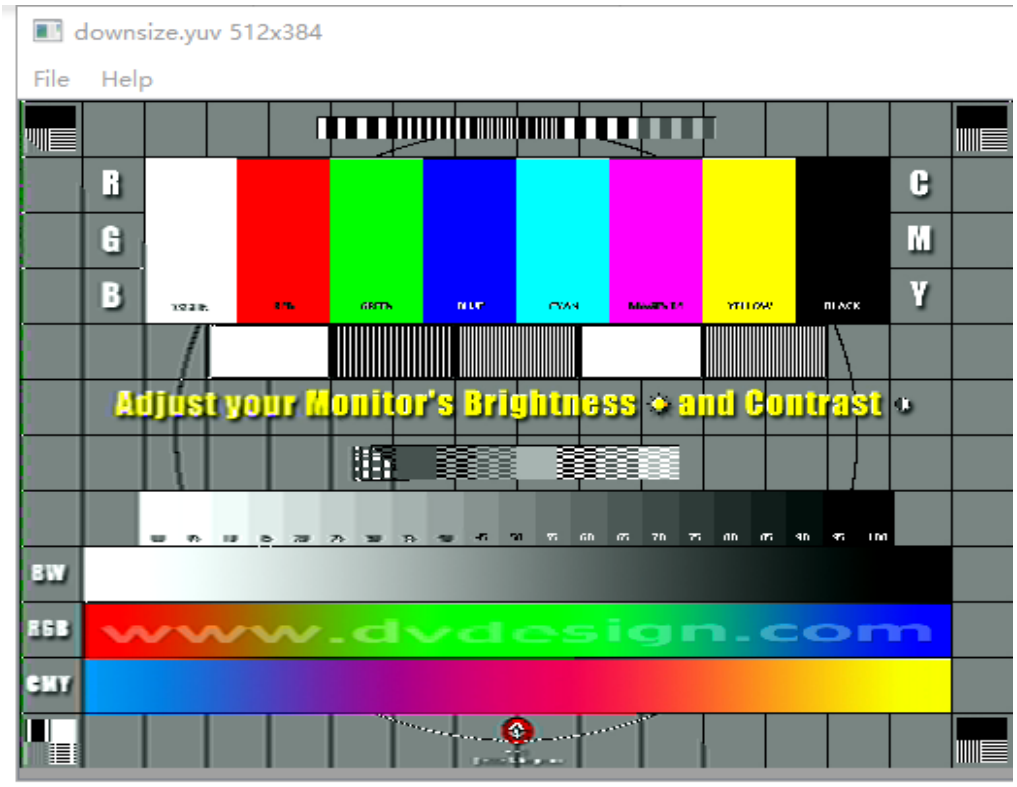
Coefficients du filtre passe bas

Filtrage et convolution

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

Partie décimation(Même technique que dans la décimation simple)

Après avoir testé ce code on obtient le résultat suivant :



Regardez les effets sur la mire. Expliquez ?

On note bien que le filtre a aidé un peu à pourvoir récupérer de l'information dans les parties où on avait des hautes fréquences. C'est dû à l'application du filtre qu'on obtient des zones gris au lieu de zones noires, car nous avons filtré tout ce qui correspondait aux hautes fréquences.

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 ?

Il faut se rappeler que un tableau a[i] est égal à *(a + i). Donc on peut insérer des indexes négatives, mais c'est quelque chose que nous ne devrions jamais faire. C ++ ne vérifie pas les limites des tableaux simples intégrés, donc techniquement, nous pouvons accéder à des emplacements qui sont hors de l'espace alloué (qui n'est que de 4 entiers et non de 7), mais nous pouvons finir par produire des erreurs.

C'est légal, mais on perd la moitié de l'information, car la case mémoire va lire à partir du milieu du tableau ou quand on aura des indexes positives. Dans notre cas on va lire à partir de l'élément central du tableau.

Pour cela on applique les conditions aux limites qu'on montre dans le code ci-dessous :

```
for(int k=-3;k<=3;k++){
int idx=i-k;
//condition aux limites
if (idx<0) idx=0; //gouche
if (idx>static_cast<int> (dst_width))
    idx=dst_width-1;//droit

int32_t data=static_cast<int32_t>(src_row[idx]);
sum+=data*fircoeff[k+3];
}
```

3) Classe abstraite

3.1) Introduction

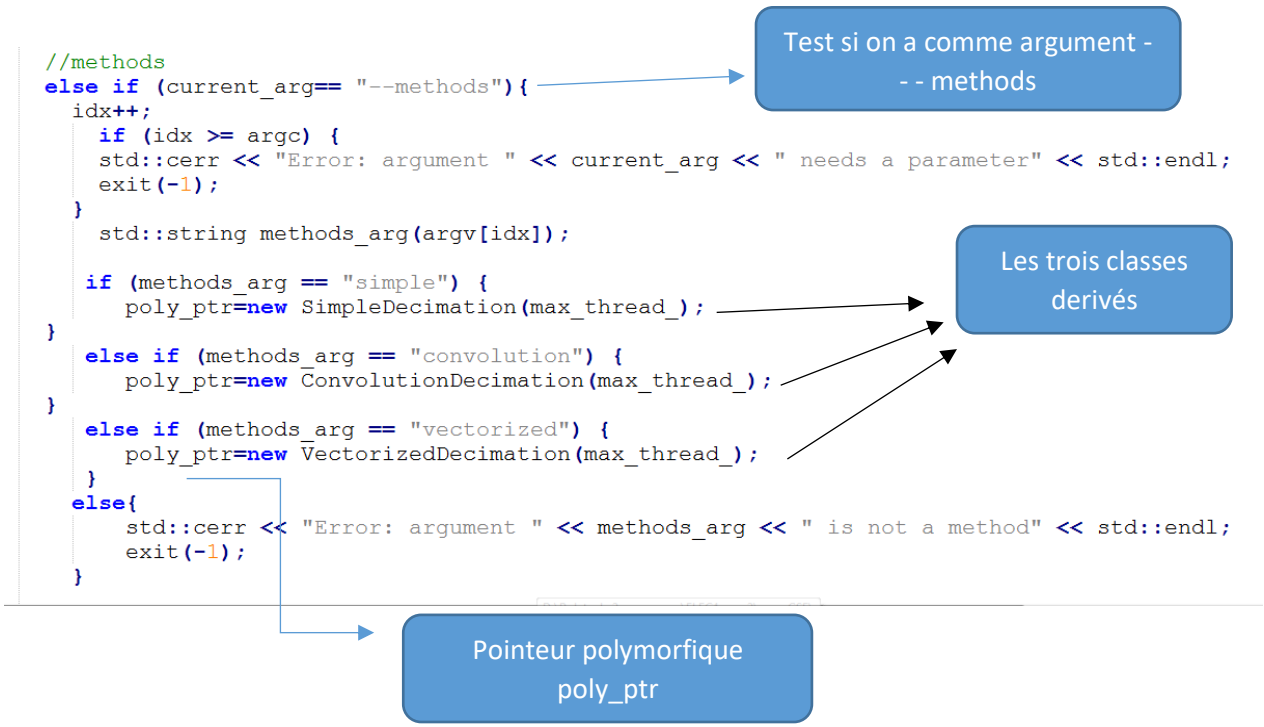
On trouve le code ci-dessous dans le fichier decimation_simple.h fourni dans l'archive :

```
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);
```

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

On remarque que l'implémentation n'est pas complète et elle n'est pas instanciable. Le mécanisme des classes abstraites permet de définir des comportements (méthodes) dont l'implémentation (le code dans la méthode) se fait dans les classes filles.

On modifie le main.cpp pour ajouter l'option - - method qui permet de choisir entre l'une des 3 méthodes : simple, convolution et vectorisé. On obtient le code ci-dessous :




```
//methods
else if (current_arg== "--methods"){
  idx++;
    if (idx >= argc) {
    std::cerr << "Error: argument " << current_arg << " needs a parameter" << std::endl;
    exit(-1);
  }
    std::string methods_arg(argv[idx]);

   if (methods_arg == "simple") {
      poly_ptr=new SimpleDecimation(max_thread_);
}
   else if (methods_arg == "convolution") {
      poly_ptr=new ConvolutionDecimation(max_thread_);
}
   else if (methods_arg == "vectorized") {
      poly_ptr=new VectorizedDecimation(max_thread_);
   }
  else{
      std::cerr << "Error: argument " << methods_arg << " is not a method" << std::endl;
      exit(-1);
  }
```

Ce pointeur polymorphique va pointer sur un objet de la classe dérivée et va se faire passer pour un pointeur de la classe base. Cela va nous permettre pouvoir choisir l'une des 3 classes dérivées.

En faisant la commande ./downsize - - help on obtient l'image ci-dessous :

```
admin@DESKTOP-PCUVUF4 MSYS ~/test/downsampling-base
$ ./downsize -h
##########################################################
##        YUV Image Decimation  version: base           ##
##########################################################

Usage: downsampling [options] filename
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)
--methods string  Choices:simple,convolution or vectorized(default :simple)

admin@DESKTOP-PCUVUF4 MSYS ~/test/downsampling-base
$ ./downsize ../images/mire1024x768.yuv --methods convolution
##########################################################
##        YUV Image Decimation  version: base           ##
##########################################################
Info: loading image from ../images/mire1024x768.yuv
Info:downsized image used method decimation convolution
Info: initial image size    : 1024x768
Info: downsized image size : 512x384
Info: downsized image calculated in 55ms, using 1 thread(s)
Info: downsized image stored in file downsize.yuv
```

3.2) Conception de la classe abstraite

On crée d'abord une classe Decimation (classe abstraite) en modifiant le fichier decimation.h.

```
extern uint nbThread ;

class Decimation{
        private:
        uint nb_thread;
        public:
        Decimation(const uint nb_thread_): nb_thread(nb_thread_) {}
        virtual void horizontal_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
                            uint dst_width, uint dst_height,
                            uint dst_stride, uint src_stride);
        virtual void vertical_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
                    uint dst_width, uint dst_height,
                    uint dst_stride, uint src_stride);
        virtual void print_method();
};
```

Et comme classe dérive SimpleDecimation on a :

```
class SimpleDecimation:public Decimation{
        public:
        explicit SimpleDecimation(const uint nb_thread) : Decimation(nb_thread) {}
        void horizontal_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
                            uint dst_width, uint dst_height,
                            uint dst_stride, uint src_stride) override;
        void vertical_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
                    uint dst_width, uint dst_height,
                    uint dst_stride, uint src_stride) override;
        void print_method() override;
};
```

Pour les classes vectorized et convolution on fera pareil parce que les trois classes héritent de la classe abstraite Decimation les mêmes méthodes horizontal_decimation, vertical_decimation et print_method. A chaque fois avec le mot clé override pour indiquer qu'on va hériter et réécrire cette fonction. Ce qui va changer dans chaque cas est le fichier .cpp.

4) Utilisation des instructions vectorisées

Dans cette partie, on va remplacer la boucle de calcule de convolution par quelques instructions SIMD. La logique utilisée pour faire la convolution et la présentation de l'image est identique à celle de la partie précédente (la méthode convolution).

Nous utilisons l'instruction SIMD pour générer un ensemble de coefficients de filtre. En utilisant les instructions de type _mm_XXX16, toutes les 8 parties doivent être 16 bits.

```
#define ALIGN16(X)   __declspec(align(16)) X
#else
#error "Error: not alignment directive"
#endif
```

Tout d'abord, nous ajoutons une définition ALIGN16(X) pour faire l'alignement. C'est la même chose que le macro ALIGN32. Grâce à ce macro et mm_load_si128, nous avons crée un registre 128 bits «*r128_coef_fir* » qui *enregistre les* coefficients de filtrage. Car il y a au total 7 coefficients, on met 0 au premier des 16 bits dans ce registre :

Puis on met les datas dans registre «r128_data » avec le sens décroissant d[7..0] pour réaliser c[k] *x[i-k] :

```
int16_t ALIGN16(mem_data[])={d[7],d[6],d[5],d[4],d[3],d[2],d[1],d[0]};
__m128i r128_data=_mm_load_si128(reinterpret_cast<__m128i const*>(mem_data));
static __m128i r128_coef_fir=_mm_load_si128(reinterpret_cast<__m128i const*>(mem_fir_coeff));
```

Et puis on fait madd :$\sum_{k=-M}^{M} c[k] * x[i-k]$ en utilisant « _mm_madd_epi16 », après ça on obtient un registre 128bits avec 4 « 32bits » .

r128_coef_fir	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]

Après faire le madd, on obtient comme ci-dessous :

```
__m128i rdata1=_mm_madd_epi16(r128_data,r128_coef_fir);/
```

Rdata1	0*d[7]-3026*d[6]	64*d[5]+10758*d[4]	17176*d[3]+10758*d[2]	64*d[1]-3026*d[4]

Et puis on fait une décalage :

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

On fait une décalage 2^{15}, donc $\frac{n}{2^{15}} = (n + 2^{14}) \gg 15$

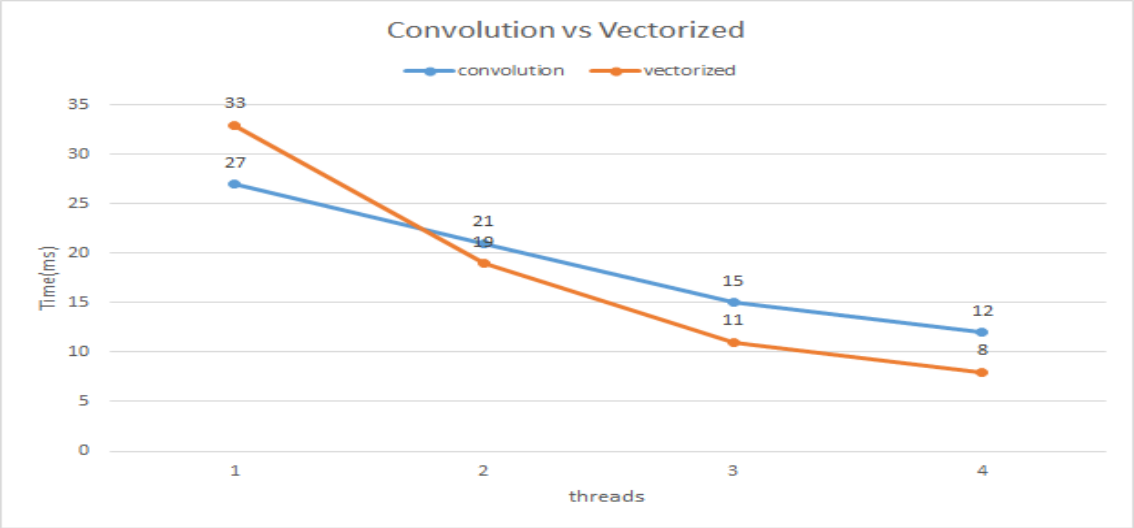
```
__m128i rdata2=_mm_hadd_epi32 (rdata1, rconst);//+16384
__m128i rdata3=_mm_srai_epi32 (rdata2,15);// >>15
```

rdata2	rdata1[0]+16384	rdata1[1]+16384	rdata1[2]+16384	rdata1[3]+16384
rdata3	rdata2[0]>>15	rdata2[1]>>15	rdata2[2]>>15	rdata2[3]>>15

A la fin on extraite dst_row[x]=rdata3[0]+ rdata3[1]+ rdata3[2]+ rdata3[3] en utilisant « _mm_extract_epi32 »

```
int sum=0;
for(int i=0;i<4;++i)
sum = sum + _mm_extract_epi32(rdata3, i);
dst_row[x]=sum;
```

On peut voir ci-dessous la courbe du temps de calcul en fonction des threads. On note un diminution du temps de calcul de 26% environ.



5) Amélioration des performances :

5.1) Votre machine :

On utilise avec CPUZ les caractéristiques suivantes de mon ordinateur :

```
--------------------------------------------------------------------------

Socket 1            ID = 0
    Number of cores     2 (max 2)
    Number of threads   4 (max 4)
    Manufacturer        GenuineIntel
    Name            Intel Core i5 7200U
    Codename        Kaby Lake-U/Y
    Specification       Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
    Package (platform ID)   Socket 1515 FCBGA (0x7)
    CPUID           6.E.9
    Extended CPUID      6.8E
    Core Stepping       B0
    Technology      14 nm
    TDP Limit       15.0 Watts
    Tjmax           100.0 °C
    Core Speed      1396.9 MHz
    Multiplier x Bus Speed  14.0 x 99.8 MHz
    Base frequency (cores)  99.8 MHz
    Base frequency (ext.)   99.8 MHz
    Stock frequency     2700 MHz
    Max frequency       3100 MHz
    Instructions sets   MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3
    Microcode Revision  0x8E
```

Nom du processeur Intel Core i5 7200

Nombre de cœurs 2

Nombre de threads 4

Fréquences de base 2.5 GHz

Instructions SSE, SSE, SSE2, SSE3, SSSE3, SSE4.2, AVX, AVX2

5.2) En Parallèle avec les « threads » :

Quand on veut optimiser une méthode il faut s'assurer que la fonction choisie représente une part significative du temps total.

On va faire l'hypothèse que les fonctions de décimation représentent la majorité du temps passé. Pour le vérifier on va utiliser l'outile gprof qui calcule le temps d'exécution de chaque fonction. Pour pouvoir utiliser cet outil on ajoute dans le makefile le flag -pg

```
LDFLAGS                 := -lpthread -pg
CC                      := g++
OBJDIR                  := ./build

PROG_VERSION             := base
```

D'abord on exécute le programme downsize avec la méthode downsize simple et on obtient l'image ci-dessous :

```
(base) ricardo@Ricardo:~/Téléchargements/downsampling-base (2)/downsampling-base$ ./downsize --methods simple ../images/BigBuckBunny_1920x1080.
yuv
######################################################
##       YUV Image Decimation  version: base        ##
######################################################
Info: loading image from ../images/BigBuckBunny_1920x1080.yuv
Info:downsized image used method decimation simple
Info: initial image size   : 1920x1080
Info: downsized image size : 960x540
Info: downsized image calculated in 23ms, using 1 thread(s)
Info: downsized image stored in file downsize.yuv
```

On applique donc gprof au exécutable downsize.

```
(base) ricardo@Ricardo:~/Téléchargements/downsampling-base (2)/downsampling-base$ gprof downsize
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.01      0.03     0.03                             SimpleDecimation::do_vertical_decimation(unsig
unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
```

Pour le cas de la décimation convolution on obtient :

```
(base) ricardo@Ricardo:~/Téléchargements/downsampling-base (2)/downsampling-base$ ./downsize --methods convolution ../images/BigBuckBunny_1920x
1080.yuv
######################################################
##       YUV Image Decimation  version: base        ##
######################################################
Info: loading image from ../images/BigBuckBunny_1920x1080.yuv
Info:downsized image used method decimation convolution
Info: initial image size   : 1920x1080
Info: downsized image size : 960x540
Info: downsized image calculated in 38ms, using 1 thread(s)
Info: downsized image stored in file downsize.yuv
(base) ricardo@Ricardo:~/Téléchargements/downsampling-base (2)/downsampling-base$ gprof downsize
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 57.15      0.04     0.04                             ConvolutionDecimation::do_horizontal_decimation(unsigned short*, unsigned short*, unsigne
d int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
 28.57      0.06     0.02                             ConvolutionDecimation::do_vertical_decimation(unsigned short*, unsigned short*, unsigned
int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
 14.29      0.07     0.01                             YuvImage::write(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<c
har> > const&) const
```

Et pour la méthode vectorisé on obtient :

```
(base) ricardo@Ricardo:~/Téléchargements/downsampling-base (2)/downsampling-base$ ./downsize --methods vectorized ../images/BigBuckBunny_1920x1
080.yuv
#####################################################
##        YUV Image Decimation  version: base        ##
#####################################################
Info: loading image from ../images/BigBuckBunny_1920x1080.yuv
Info:downsized image used method decimation vectorized
Info: initial image size   : 1920x1080
Info: downsized image size : 960x540
Info: downsized image calculated in 59ms, using 1 thread(s)
Info: downsized image stored in file downsize.yuv
(base) ricardo@Ricardo:~/Téléchargements/downsampling-base (2)/downsampling-base$ gprof downsize
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 44.45      0.04     0.04                             VectorizedDecimation::do_vertical_decimation(unsigned short*, unsigned short*, unsigned i
nt, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
 44.45      0.08     0.04                             VectorizedDecimation::do_horizontal_decimation(unsigned short*, unsigned short*, unsigned
 int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned int)
 11.11      0.09     0.01                             YuvImage::write(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<c
har> > const&) const
```

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 ?

Pour la méthode convolution on obtient à l'inverse et pour le vectorisé on obtient le temps égal.

5.3) Méthode à suivre :

On a choisi la méthode de découpage de l'image en n bandes horizontales et verticales.

On modifier la méthode Decimation ::do_horizontal_decimation(…) pour appeler max_thread fois la fonction horizontal sur des bandes successives de l'image. On ajoute le code ci-dessous pour le fichier decimation_simple.cpp. On utilise i_ pour couper l'image jusqu'à max_thread_.

```
    uint i_begin,i_end;

i_begin=i_*src_height/max_thread_;
i_end=(i_+1) *src_height/max_thread_;

    for (uint y =i_begin; y < i_end; ++y) {
uint16_t *src_row = src_col0;
uint16_t *dst_row = dst_col0;
```

Pour la décimation vertical notre premier boucle for commence par for (uint x =i_begin; x < i_end; ++x) et la deuxième boucle est donc for (uint y = 0; y < dst_height; ++y). On procède de la façon suivante :

```
void SimpleDecimation::do_vertical_decimation(uint16_t *dst_ptr, uint16_t *src_ptr,
                    uint src_width, uint dst_height,
                    uint dst_stride, uint src_stride,uint i_,uint max_thread_) {


 //
 uint16_t *src_row0 = src_ptr;
 uint16_t *dst_row0 = dst_ptr;

 uint i_begin,i_end;

   i_begin=i_*src_width/max_thread_;
   i_end=(i_+1) *src_width/max_thread_;


       for (uint x = i_begin; x < i_end; ++x) {
   uint16_t *src_col = src_row0;
   uint16_t *dst_col = dst_row0;
   for (uint y = 0; y < dst_height; ++y) {
     *dst_col = *src_col;
     dst_col += dst_stride;
     src_col += 2 * src_stride;
   }
   src_row0++;
   dst_row0++;
 }
}
```

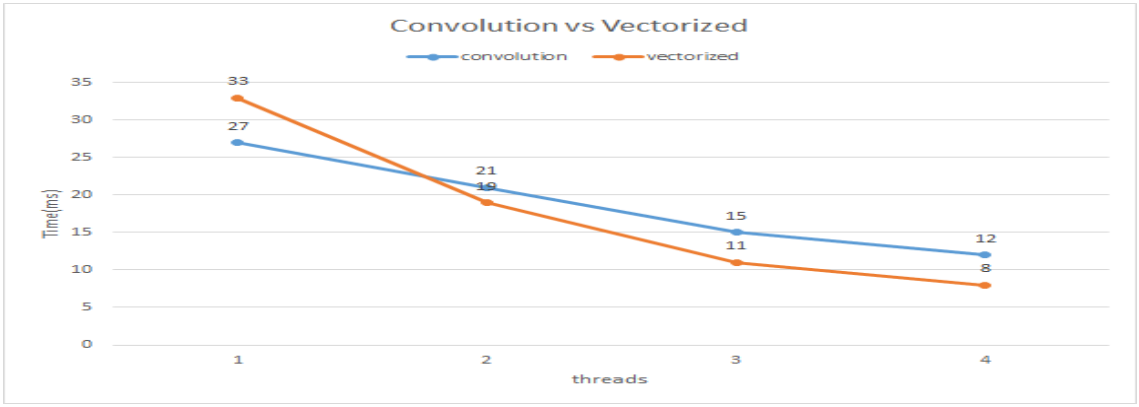
On modifie aussi le fichier yuvimage.cpp en ajoutant uint i_ et uint max_thread_ de la façon suivante :

```
void YuvComponent::downsize_from(const YuvComponent &src,uint i_,uint max_thread_) const {
  //
  //
  YuvComponent tmp(width_, src.height_,poly_ptr);

  src.poly_ptr->do_horizontal_decimation(
      tmp.ptr_, src.ptr_,
      width_, src.height_,
      tmp.stride_, src.stride_,i_,max_thread_);

  src.poly_ptr->do_vertical_decimation(
      ptr_, tmp.ptr_,
      src.width_, height_,
      stride_, tmp.stride_,i_,max_thread_);
}
```

Et on a modifié le fichier main pour appliquer le parallélisme des fonctions avec la fonction anonyme [=] qui capture tous les occurrences libres par copie.

```
//thread
        for(uint i_=0;i_<(max_thread_-1);++i_){
                threads.emplace_back([=](){
       result_image.downsize_threaded_by_component(input_image,i_, max_thread_);
          });
        }
        result_image.downsize_threaded_by_component(input_image,max_thread_-1, max_thread_);
        for (auto &thread_elem:threads){
                thread_elem.join();
        }
```

On trouve la courbe du temps d'exécution en fonction du nombre de threads.



On peut noter que ça diminue presque linéairement de 1 à 4 threads. On voit que ça commence à varier peu proche de 4 threads. Cela est parce que on ne peut paralléliser assez rapidement les fonctions. C'est comme si on utiliserait 4 threads même si on a plus.