

HPC - Optimisation des performances et architecture des processeurs

Rapport Projet : Optimisation des temps de calcul et processeur RISC Projet “Vision”



Carte SABRE/freescale IMX6

SOMMAIRE

I - Introduction	p3
II - Étape 1 : Filtre de Sobel	p4
III - Étape 2 : Filtre Médian	p9
IV - Conclusion	p12
V - Annexe	p13

Introduction

Nous avons comme objectif d'implémenter les filtres Sobel et Médian pour ensuite utiliser les techniques d'optimisation précédemment apprises. Ces filtres sont appliqués sur une image pour permettre de la modifier : le filtre de Sobel ne conserve que les contours et le filtre Médian "lisse" l'image en éliminant les impuretés, elle devient plus "flou". L'image traitée sera celle fournie par une webcam qui nous filmera en temps réel pour ensuite obtenir les performances des différents filtres. L'affichage des images est assurée avec les fonctions OpenCV.



Ces optimisations seront faites dans le contexte d'une chaîne de traitement d'image embarquée sur la carte SABRE/freescale IMX6 (ARM_Cortex-A9, quad-core). Pour cela, plusieurs types d'optimisations existent : l'optimisation algorithmique, l'optimisation pour les processeurs RISC comme le déroulage ou encore l'optimisation pour une utilisation efficace des ressources matérielles comme l'accès aux données, gestion de la mémoire cache, multi-thread.



Ainsi, on mesure dans un premier temps les performances du projet original puis on le remplace par nos propres méthodes pour les filtres Médian et Sobel.

Étape 1 : Filtre de Sobel

Dans cette première partie, nous allons implémenter le filtre de Sobel pour l'utiliser sur la vidéo issue de la webcam.

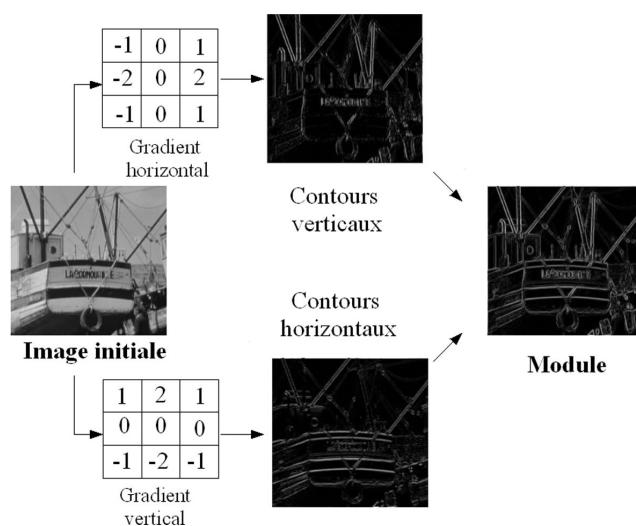
Le filtre de Sobel est un opérateur utilisé pour la détection de contours en traitement d'image.

Ce filtre, à réponse impulsionnelle finie, calcule le gradient de l'intensité de chaque pixel. Grâce à cela, on obtient la direction de la plus forte variation du clair au sombre, ainsi que le taux de changement dans cette direction. Par conséquent, on sait sur quels points se trouve un changement rapide de luminosité qui peut donc être un bord d'un objet, ainsi que l'orientation du bord.

Par exemple, sur les photos ci-dessous, on voit bien que la photo de droite représente les contours de la photo de gauche obtenus grâce au filtre de Sobel.



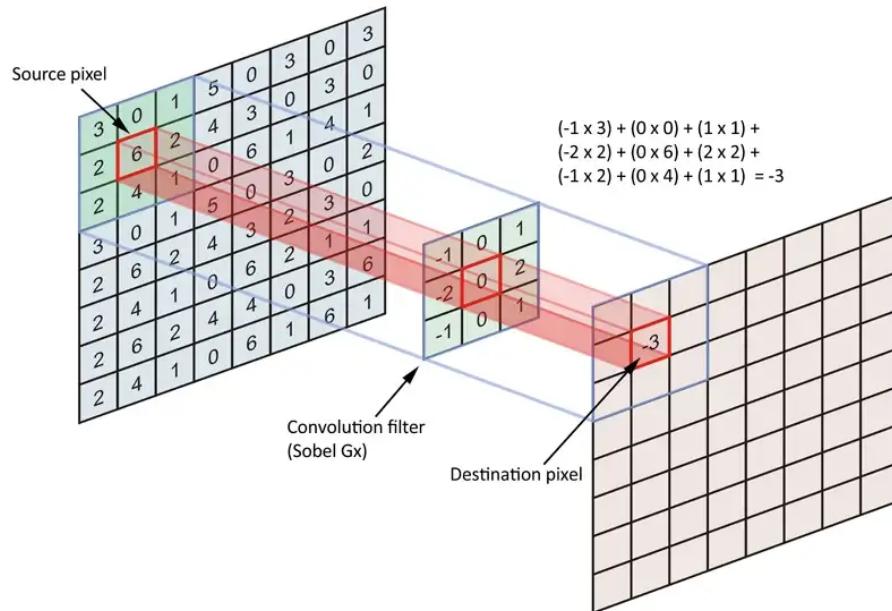
Pour calculer la matrice finale une fois que le filtre a été appliqué, on utilise des matrices de convolution pour calculer des approximations des dérivées horizontale et verticale :



En considérant A comme la matrice de l'image initiale, on a les différentes convolutions à effectuer :

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

De plus, la convolution de matrices se calcule comme ceci :



Une fois les matrices G_x et G_y calculées, on peut les combinées en chaque point pour obtenir une approximation de la norme du gradient, ici la **matrice G** :

$$G = \sqrt{G_x^2 + G_y^2}$$

On décide que tous les points sur le cadre extérieur valent 0 pour G_x et ne seront pas à traiter. Sur son image associée, les bords seront donc noirs. Ce choix est cohérent car les bords soient noirs pour les photos passées sous le filtre de Sobel attendues.



Une approximation possible pour le calcul de G est :

$$G = \frac{(|G_x| + |G_y|)}{8} \quad (1)$$

On commence par déterminer la performance du code de Sobel par défaut et on obtient : Sobel : **9866.000000**.

Ensuite, on code notre première implémentation du filtre de Sobel sans utiliser les méthodes d'OpenCV. Afin de nous aider dans notre compréhension de l'implémentation, nous avons fait des recherches dans la documentation de la méthode Sobel d'OpenCV.

Nous avons pour première performance : **86196.000000**. Celle-ci est très grande en comparaison avec le filtre par défaut. Nous allons donc l'optimiser.

Pour cela, nous faisons différentes améliorations. Tout d'abord, nous faisons un déroulage de boucle. Celui-ci, n'améliore pas les performances puisque l'on obtient : **86199.000000**. Nous décidons donc de ne pas conserver cette version.

Ensuite, dans les différentes techniques d'optimisation, on retrouve la création de variables intermédiaires. Ainsi, les performances sont quasiment doublées avec pour résultat : **46117.000000**.

Cependant, la technique du réagencement du code, soit de changer de place les variables aléatoire, ne change presque rien à la performance.

En prenant du recul sur notre code, nous voyons que nous appelons 9 fois la valeur d'une case mémoire dans notre grand tableau. Nous pouvons réfléchir à une astuce pour placer un pointeur à la première variable puis se rendre aux autres cases mémoires à partir de cette même variable. Après quelques recherches de documentation sur internet, nous avons trouvé la possibilité de placer un pointeur mais nous ne trouvons pas la possibilité d'ajouter X mémoire à cette variable. Nous faisons uniquement des suppositions mais au vu des améliorations de la création de pointeur intermédiaire, cette méthode aurait pu nous permettre d'optimiser grandement les performances de notre code.

De plus, comme vu précédemment, une approximation possible pour le calcul de G existe et est :

$$G = \frac{(|Gx| + |Gy|)}{8}$$

On teste cette approximation dans le code en remplaçant :

`out.at<unsigned char>(i, j) = (unsigned char)(sqrt(sumX*sumX + sumY*sumY));`

Par : `out.at<unsigned char>(i, j) = (unsigned char) ((abs(sumX)+abs(sumY)) / 8);`

Celle-ci fait gagner 5% de performance avec un résultat de **43809.000000**. Cependant, nous n'avons pas un résultat convaincant car le filtre ne délimite plus très bien les contours. Nous faisons le choix de perdre en performances mais de gagner en précision sur le filtre.

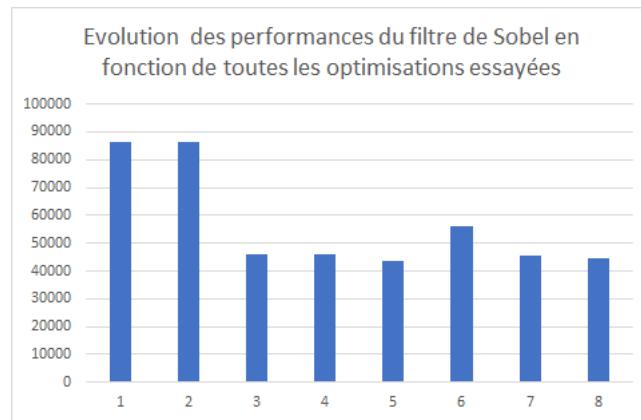
On essaie, par la suite, de mettre les tableaux Hx et Hy en "#define" mais on remarque que ce n'est pas optimisant car les performances **augmentent de 10000** environ, ce qui est beaucoup trop long.

Ensuite, on continue d'essayer d'optimiser notre filtre en supprimant les calculs qui concernent une multiplication par 0 puisque dans tous les cas la valeur finale sera 0. Tous ces calculs sont donc mis en commentaire dans le code car il n'y a rien besoin d'ajouter dans le calcul de la convolution. On réussit donc à optimiser les performances en gagnant environ 1000 en passant de 46117.000000 à **45341.000000**.

On supprime également la multiplication par 1 dans les calculs concernés en ajoutant directement la valeur dans le calcul de la convolution sans faire de multiplication. On obtient donc **44629.000000** ce qui représente également une optimisation d'environ 1000.

Ainsi, on peut regrouper toutes les optimisations essayées sur le filtre de Sobel en un tableau récapitulatif avec leur performance et leur gain en pourcentage. Un histogramme associé aux performances également tracé :

Optimisation	Performance	Gain (%)
1 Initial	86196	1,00
2 Déroulage boucle	86199	1,00
3 Variables	46117	0,54
4 Réagencement	46115	0,54
5 Approximation G	43809	0,51
6 Hx,Hy en define	56124	0,65
7 *0	45341	0,53
8 *1	44629	0,52



La première colonne du tableau correspond aux différentes optimisations. La deuxième détaille les performances de chacune des versions du code avec cette optimisation. La dernière est le calcul en pourcentage du gain c'est à dire :

$$Gain = \frac{\text{performance de l'optimisation } k}{\text{performance initial}}$$

Ainsi, la valeur initiale est 1,00 soit 100%.

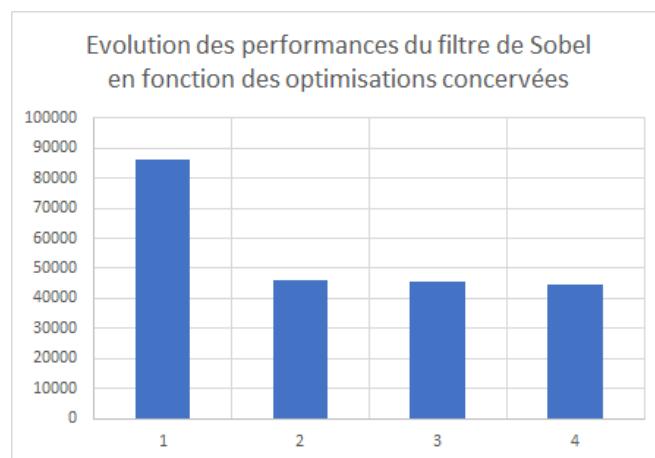
Les trois cases vertes du tableau correspondent aux cases dont les optimisations ont été conservées dans le code final puisque celles-ci apportent une bonne optimisation. En effet, le déroulage de boucle et le réagencement quant à eux ne modifient pas le gain.

De plus, l'histogramme nous montre bien que l'optimisation 6 n'est pas du tout efficace et fait perdre des performances. Ainsi, elle ne sera pas conservée dans le code final.

La meilleure performance est visible sur l'histogramme à l'optimisation 5 et correspond à l'approximation de G. Comme vu précédemment, elle n'est pas conservée car elle nuit à la qualité du filtre.

Le tableau récapitulatif suivant nous permet de faire un graphique sur toutes les optimisations qui ont été conservées dans le code final du filtre de Sobel :

Optimisation	Performance	Gain (%)
1 Initial	86196	1,00
2 Variables	46117	0,54
3 *0	45341	0,53
4 *1	44629	0,52



L'histogramme nous montre bien que les performances ont été quasiment améliorées par 2 à la fin de toutes les optimisations, visible également dans le tableau puisque le gain est de 52%.

Ensuite, vu que nos résultats sont bien inférieurs à OpenCV, nous avons l'idée d'aller regarder les sources d'OpenCV lui-même. Après pas mal de temps d'essayer à comprendre certaines fonctionnalités, nos capacités en C sont limitées et ne nous permettent pas d'adapter le code et d'optimiser le nôtre.

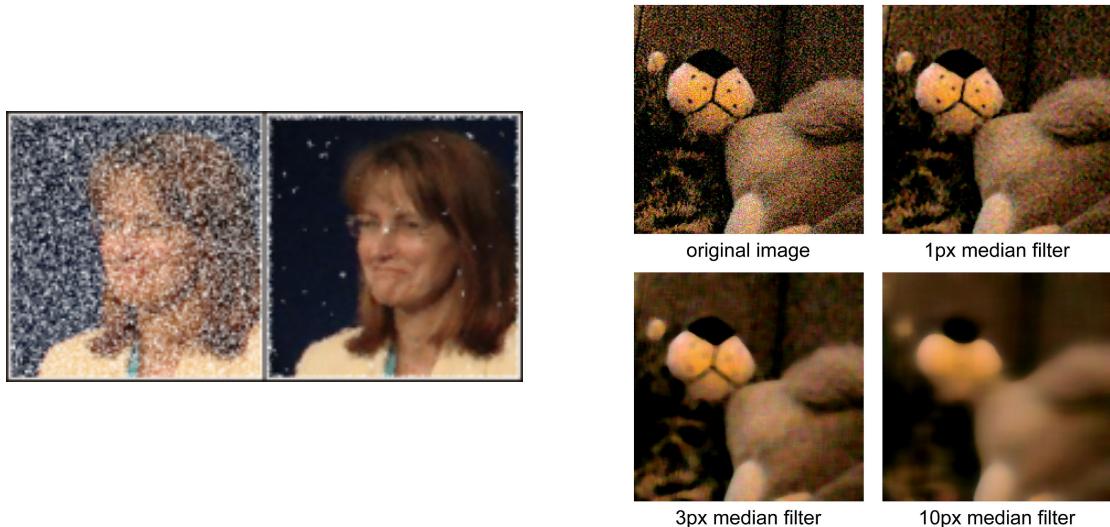
Source OpenCV : <https://github.com/opencv/opencv>

En comparaison avec le code de Sobel par défaut qui est de 9866.000000, notre résultat est bien inférieur. Mais on obtient tout de même pour performance finale pour le filtre de Sobel : **44629.000000**.

Lorsque nous lançons le programme, l'image est fluide et le filtre de Sobel fonctionne très bien car on voit bien le contour des éléments, comme visible sur la capture d'écran présente en introduction.

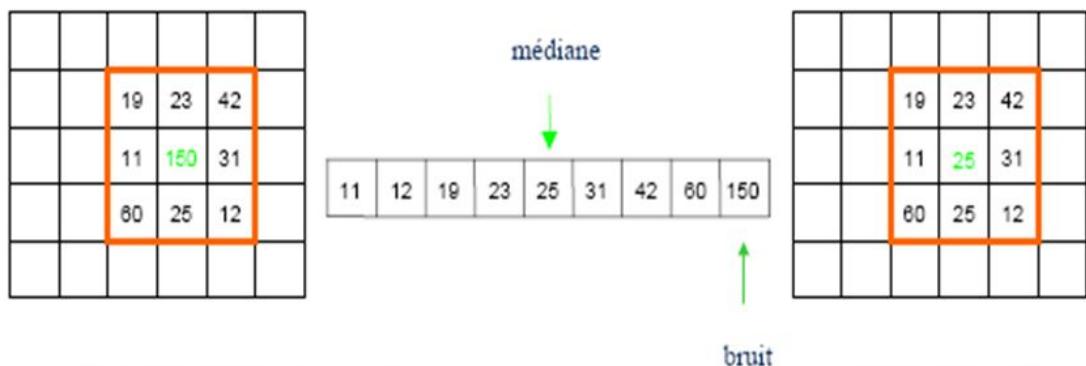
Étape 2 : Filtre Médian

Dans cette seconde partie, nous allons implémenter le filtre Médian. Celui-ci permet sous certaines conditions de réduire le bruit tout en conservant les contours de l'image. Il respecte les contours et élimine les valeurs extrêmes.



Ainsi sur la première image, les points blancs de la photo de gauche représentent le bruit de l'image, les impuretés. Le filtre permet d'obtenir la photo de la meuf de droite qui est beaucoup plus lisse et ne comporte quasiment plus de valeurs aberrantes, ici les points blancs. Sur la deuxième image, on remarque que le filtre lisse la photo et a donc tendance à un peu la flouter.

Pour cela, le filtre médian remplace chaque entrée par la valeur médiane de son voisinage :



On décide que tous les points sur le cadre extérieur conservent leur valeur initiale. Sur son image associée, les bords seront donc les mêmes que sur l'image initiale. Ce choix est cohérent car comme visible sur l'image avec la femme, les pixels sur le contour restent inchangés.

On commence par déterminer la performance du code Médian par défaut : **18550.000000**.

Ensuite, on code notre première implémentation du filtre Médian sans utiliser les méthodes d'OpenCV. Afin de nous aider dans notre compréhension de l'implémentation, nous avons fait des recherches dans la documentation de la méthode Sobel d'OpenCV.

Nous avons pour première performance : **770169.000000**. Celle-ci est beaucoup plus grande que celle du filtre par défaut avec une différence d'environ 750000. Nous allons donc l'optimiser.

Notre première amélioration concerne le tri du tableau (sort) de la valeur sélectionnée et de ses voisins. Après quelques recherches concernant les performances de chaque algorithme de tri d'un tableau sur internet, nous utilisons, dans un premier temps, l'algorithme l'algorithme std::sort(). Cependant, en faisant cette recherche, nous réfléchissons et nous nous disons que le début ou la fin du tableau ne nous intéresse pas. En effet, on cherche la médiane des valeurs qui est donc la valeurs du milieu. Ainsi, lorsque la première moitié des valeurs sera triée et que nous obtenons notre médiane, il n'est pas utile de trier la suite du tableau. Nous remplaçons donc notre std::sort() qui tri tout le tableau par un std::partial_sort() qui triera le tableau jusqu'à la médiane.

Documentation : <https://en.cppreference.com/w/cpp/algorithm/sort>
https://en.cppreference.com/w/cpp/algorithm/partial_sort

Par exemple, pour le tableau de valeurs suivant : 32 71 12 45 26 80 53 33 45,
Après le tri, le tableau est :

12 26 32 33 | 45 | 71 80 53 45

médiane

Finalement, on obtient comme performances : **758912.000000** ce qui est une bonne amélioration puisque nous avons une diminution de plus de 10000.

Ensuite, la technique du déroulage de boucle est utilisée et apporte une amélioration plus importante que pour le filtre de Sobel car on améliore nos résultats d'environ 5000 : **754565.000000**

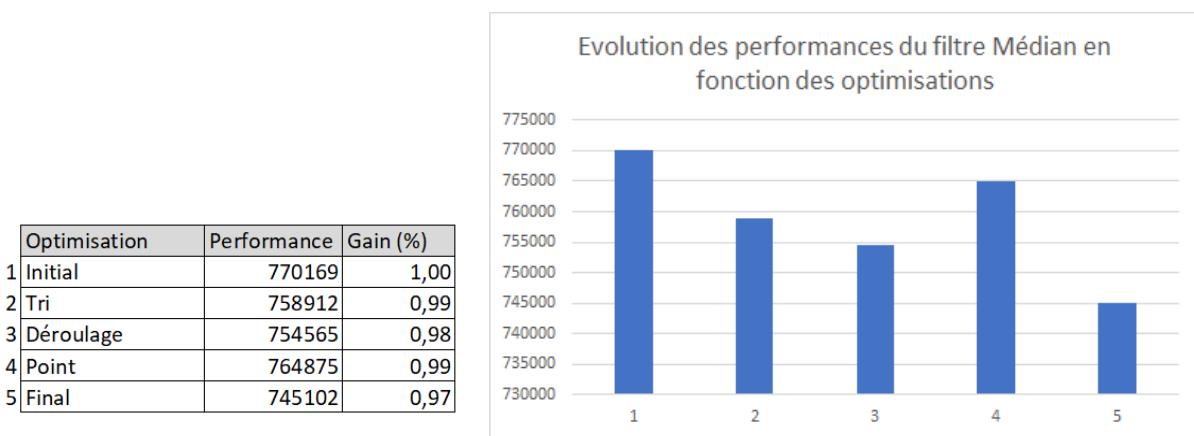
En cherchant dans la documentation openCV, nous remarquons qu'il existe un structure Point et qu'il existe une méthode sur la structure Mat qui renvoie notre valeur du pixel à l'aide de Point. Nous voulons tester voir si ceci est plus rapide de mettre un point ou de mettre des abscisses/ordonnées directement.

En mettant en place ce changement, nous obtenons un résultat de **764875.000000**. Nous supposons que la création de variable Point est plus coûteuse que si nous donnons directement un x et y.

Documentation : https://docs.opencv.org/3.4/db/d4e/classcv_1_1Point_.html

Nous supprimons donc toutes nos modifications, remettons le code avec nos x;y et nous obtenons de meilleurs résultats. Nous remarquons aussi que les résultats ne sont pas stables et qu'il faut faire une moyenne sur plusieurs frames pour avoir un résultat fiable. Nous pouvons expliquer ce changement avec le fait que les images à traiter ne soient jamais les mêmes ou que la carte peut choisir un autre processus à traiter.

Ainsi, on peut regrouper toutes les optimisations essayées sur le filtre Médian en un tableau récapitulatif avec leur performance et leur gain en pourcentage. Un histogramme associé aux performances également tracé :



Les colonnes sont les mêmes que pour le filtre de Sobel, seul le contenu est différent.

On remarque sur l'histogramme que l'optimisation 4 n'est pas une bonne optimisation puisque les performances sont moins bonnes avec. Ainsi, à par celle-ci, toutes les optimisations sont conservées car elles apportent toutes une meilleure performance au filtre final.

Cependant, dans la troisième colonne du tableau, on remarque que le gain n'est pas autant amélioré que pour le filtre de Sobel. En effet, celui-ci possède un gain final de 52% alors que pour le filtre Médian, celui-ci est de 97% ce qui fait une amélioration de 3%. Ceci est très faible mais représente tout de même une diminution de 25000.

En comparaison avec le code du filtre Médian par défaut qui est de 18550.000000, notre résultat est bien inférieur puisque notre performance finale est de : **745102.000000** mais nous avons tout de même réussi à l'optimiser.

Lorsque nous lançons le programme du filtre Médian, l'image n'est pas fluide mais celle-ci est bien flouté, comme visible sur la capture d'écran présente en introduction.

Ainsi, lorsque le programme avec les deux filtres s'exécute, la vidéo du filtre de Sobel est également impactée par la performance du filtre Médian. Celle-ci est donc beaucoup moins fluide que lorsque nous exécutons le filtre de Sobel seul.

On remarque bien cette différence dans la vidéo de présentation qui dans l'ordre montre :

- le programme avec les deux filtres
- le filtre Sobel
- le filtre Médian

Conclusion

À l'aide de nos différents TPs et de nos connaissances personnelles, nous avons réussi à optimiser nos propres algorithmes pour les filtres de Sobel et Médian et avons pu creuser et mettre en situation nos résultats. Cependant, nous n'obtenons pas de meilleurs résultats que OpenCV même en ayant regardé et appris de leur code source. Nous supposons aussi que nos capacités, compétences et connaissances en C/C++ nous limitent vu que nous n'utilisons aucune fonctionnalité spécifique à ce langage.

Nos algorithmes ont la possibilité d'être améliorés si nous utilisons des pointeurs ou des fonctionnalités liées à la mémoire. Nous pouvons aussi creuser que l'utilisation de pointeurs sur les matrices et non de variable.

Pour finir, nous avons réussi à optimiser nos codes à notre échelle et obtenir un programme fluide et fonctionnel.

Annexe

Le code de notre projet est le suivant :

```
/*
 * Libraries standarts
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <algorithm>      // std::sort
#include <vector>          // std::vector

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/*
 * Libraries OpenCV "obligatoires"
 *
 */
#include "highgui.h"
#include "cv.h"
#include "opencv2/opencv.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

/*
 * Définition des "namespace" pour évite cv::, std::, ou autre
 *
 */
using namespace std;
using namespace cv;
using std::cout;

/*
 * Some usefull defines
 * (comment if not used)
 */
#define PROFILE
#define VAR_KERNEL
#define N_ITER 100

#ifndef PROFILE
#include <time.h>
#include <sys/time.h>
#endif

int sort_array(int tab[9]){
    std::vector<int> myvector (tab, tab+9);           // 32 71 12 45 26 80 53 33 45
                                                        // 12 26 32 33 45 71 80 53 45

    // using default comparison (operator <):
    std::partial_sort (myvector.begin(), myvector.begin() + 5, myvector.end());

    std::cout << "myvector contains:";
```

```

//for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
//  std::cout << ' ' << *it;
//std::cout << '\n';
return myvector[4] ;
}

void newMedian(Mat in, Mat out, int rows, int cols, int n)
{
    int sumX, sumY;
    for(int j = 1; j < cols - 1; j++)
        for(int i = 1; i < rows - 1; i++)
    {

        int c1 = (int)in.at<unsigned char>(i - 1, j - 1);
        int c4 = (int)in.at<unsigned char>(i-1, j);
        int c7 = (int)in.at<unsigned char>(i - 1, j + 1);

        int c2 = (int)in.at<unsigned char>(i, j - 1 );
        int c5 = (int)in.at<unsigned char>(i, j);
        int c8 = (int)in.at<unsigned char>( i, j + 1);

        int c3 = (int)in.at<unsigned char>(i + 1, j - 1);
        int c6 = (int)in.at<unsigned char>(i + 1, j);
        int c9 = (int)in.at<unsigned char>(i + 1, j + 1);

        int array[9] = {c1, c2, c3, c4, c5, c6, c7, c8, c9};

        int medianint = sort_array(array);
        out.at<unsigned char>(i, j) = (unsigned char) medianint ;
    }
}

void newSobel(Mat in, Mat out, int rows, int cols)
{
    int Hx[9] = {-1,0,1,-2,0,2,-1,0,1};
    int Hy[9] = {1,2,1,0,0,0,-1,-2,-1};
    int sumX, sumY;
    for(int j = 1; j < cols - 1; j++)
        for(int i = 1; i < rows - 1; i++)
    {
        sumX = 0; sumY = 0;// count = 0;

        unsigned char c1 = in.at<unsigned char>(i - 1, j - 1);
        unsigned char c4 = in.at<unsigned char>(i - 1, j);
        unsigned char c7 = in.at<unsigned char>(i - 1, j + 1);

        unsigned char c2 = in.at<unsigned char>(i, j - 1);
        unsigned char c5 = in.at<unsigned char>(i, j);
        unsigned char c8 = in.at<unsigned char>(i, j + 1);

        unsigned char c3 = in.at<unsigned char>(i + 1, j - 1);
        unsigned char c6 = in.at<unsigned char>(i + 1, j);
        unsigned char c9 = in.at<unsigned char>(i + 1, j + 1);

        sumX += c1 * Hx[0];
        sumY += c1 /* Hy[0];
    }
}

```

```

        //sumX += c2 * Hx[1];
        sumY += c2 * Hy[1];

        sumX += c3 ; /* Hx[2];
        sumY += c3 ; /* Hy[2];

        sumX += c4 * Hx[3];
        //sumY += c4 * Hy[3];

        //sumX += c5 * Hx[4];
        //sumY += c5 * Hy[4];

        sumX += c6 * Hx[5];
        //sumY += c6 * Hy[5];

        sumX += c7 * Hx[6];
        sumY += c7 * Hy[6];

        //sumX += c8 * Hx[7];
        sumY += c8 * Hy[7];

        sumX += c9 ; /* Hx[8];
        sumY += c9 * Hy[8];

        out.at<unsigned char>(i, j) = (unsigned char)(sqrt(sumX*sumX + sumY*sumY));
        //out.at<unsigned char>(i, j) = (unsigned char) (( abs(sumX)+abs(sumY) )/8) ;
    }
}

/*
*
*----- MAIN FUNCTION -----
*
*/
int main () {
//-----
// Video acquisition - opening
//-----

int port = 0 ;
for(int i=0 ; i<5 ; i++) { // Code pour débug le port des vidéos
    VideoCapture cap(i) ;
    if(cap.isOpened()) {
        cout << "Port à utiliser : " << i << endl ;
        port = i ;
        break ;
    }
}

VideoCapture cap(port); // le numéro 0 indique le point d'accès à la caméra 0 => /dev/video0
if(!cap.isOpened()){
    cout << "Erreur" ;
    return -1;
}
// HD resolution 1 920 × 1 080,
int rows = 240; // 480, 1080
int cols = 320; // 640, 1920
cap.set(CAP_PROP_FRAME_WIDTH, cols);

```

```

cap.set(CAP_PROP_FRAME_HEIGHT, rows);

//-----
// Déclaration des variables - imagesize
// Mat - structure contenant l'image 2D niveau de gris
// Mat3b - structure contenant l'image 2D en couleur (trois canaux)
//
Mat3b frame; // couleur
Mat frame1; // niveau de gris
Mat frame_gray; // niveau de gris
Mat grad_x;
Mat grad_y;
Mat abs_grad_y;
Mat abs_grad_x;
Mat grad;

// variable contenant les paramètres des images ou d'exécution
int ddepth = CV_16S;
int scale = 1;
int delta = 0;
unsigned char key = '0';

#define PROFILE

#ifndef PROFILE
// profiling / instrumentation libraries
#include <time.h>
#include <sys/time.h>
#endif

//-----
// Création des fenêtres pour affichage des résultats
// vous pouvez ne pas les utiliser ou ajouter selon ces exemple
//
cvNamedWindow("Video input", WINDOW_AUTOSIZE);
cvNamedWindow("Video gray levels", WINDOW_AUTOSIZE);
cvNamedWindow("Video Mediane", WINDOW_AUTOSIZE);
cvNamedWindow("Video Edge detection", WINDOW_AUTOSIZE);
// placement arbitraire des fenêtre sur écran
// sinon les fenêtres sont superposée l'une sur l'autre
cvMoveWindow("Video input", 10, 30);
cvMoveWindow("Video gray levels", 800, 30);
cvMoveWindow("Video Mediane", 10, 500);
cvMoveWindow("Video Edge detection", 800, 500);

// -----
// boucle infinie pour traiter la séquence vidéo
//

while(key!='q'){
//
// acquisition d'une trame video - librairie OpenCV
    cap.read(frame);
//conversion en niveau de gris - librairie OpenCV
    cvtColor(frame, frame_gray, CV_BGR2GRAY);
}

```

```

// image smoothing by median blur
//
int n = 5;

#ifndef PROFILE
struct timeval start, end;
gettimeofday(&start, NULL);
#endif

// -----
// calcul de la mediane - librairie OpenCV
frame1 = frame_gray.clone();
newMedian(frame_gray, frame1, rows, cols, n);
//medianBlur(frame_gray, frame1, n);

#ifndef PROFILE
gettimeofday(&end, NULL);
double e = ((double) end.tv_sec * 1000000.0 + (double) end.tv_usec);
double s = ((double) start.tv_sec * 1000000.0 + (double) start.tv_usec);
printf("New Mediane : %f\n", n, (e - s));
#endif

#ifndef PROFILE
gettimeofday(&start, NULL);
#endif

// -----
// calcul du gradient- librairie OpenCV
grad = frame1.clone();
/// Gradient Y
/*Sobel( frame1, grad_x, ddepth, 1, 0, 3, scale, delta, BORDER_DEFAULT );
/// absolute value
convertScaleAbs( grad_x, abs_grad_x );
/// Gradient Y
Sobel( frame1, grad_y, ddepth, 0, 1, 3, scale, delta, BORDER_DEFAULT );
/// absolute value
convertScaleAbs( grad_y, abs_grad_y );
/// Total Gradient (approximate)
addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );*/
newSobel(frame1, grad, rows, cols);

#ifndef PROFILE
gettimeofday(&end, NULL);
e = ((double) end.tv_sec * 1000000.0 + (double) end.tv_usec);
s = ((double) start.tv_sec * 1000000.0 + (double) start.tv_usec);
printf("Sobel : %f\n", (e - s));
#endif

// -----
// visualisation
// taille d'image réduite pour meilleure disposition sur écran
//      resize(frame, frame, Size(), 0.5, 0.5);
//      resize(frame_gray, frame_gray, Size(), 0.5, 0.5);
//      resize(grad, grad, Size(), 0.5, 0.5);

```

```
imshow("Video input",frame);
imshow("Video gray levels",frame_gray);
imshow("Video Mediane",frame1);
imshow("Video Edge detection",grad);

key=waitKey(5);
}
```