Détéction de ligne en vue d'une conduite autonome

Projet Informatique Embraquée

Mustapha KHELIFI
Mohamed DIAWARA
Othman EL HOUFI

Encadrant du Projet
Mehdi ABDELWAHED

8 mai 2021

Table des matières

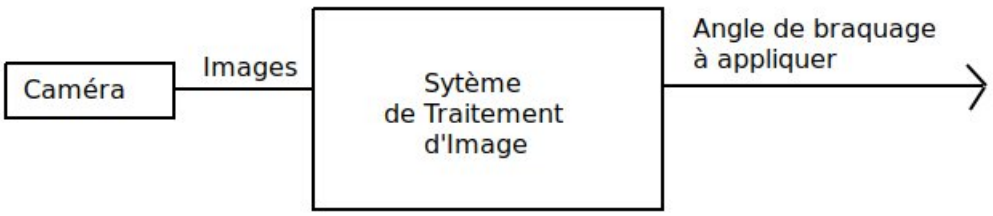
1 Introduction

1.1 Contexte du projet

Dans le cadre du module Informatique Embarquée, nous avons vu l'importance du temps de production des résultats dans un système temps réel et qu'un retard à ce niveau la entraîne une erreur dusystème du fait qu'on rate une échéance.

1.2 Objectif du projet

Dans ce projet nous avons pour but de réaliser un système de traitement d'image pour une voiture autonomes qui va prendre en entrée l'image d'un caméra et donner en sortie l'angle avec lequel tourner.

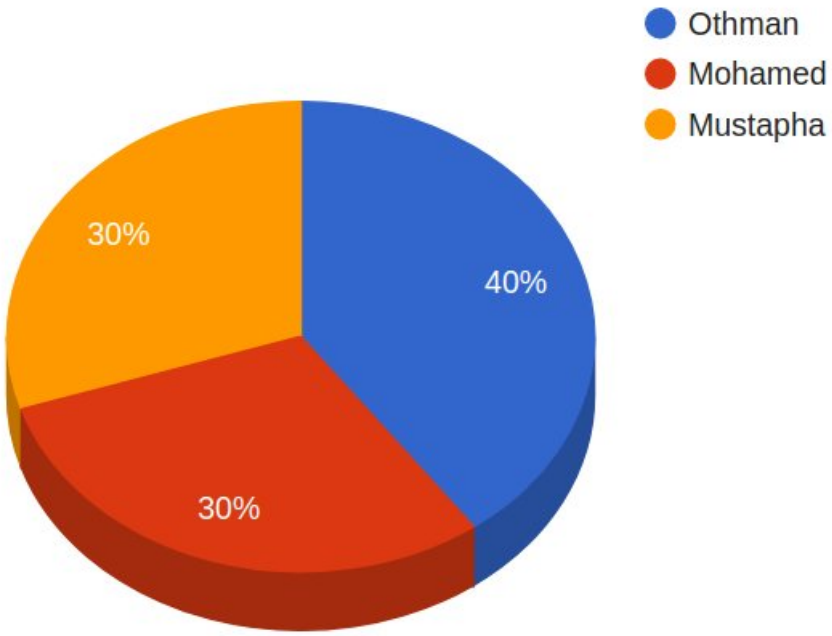


Pour un fonctionnement optimal nous devons connaître la vitesse de ce système et certifier la fréquence de fonctionnement tout en sachant que plus la fréquence est meilleurs plus on traitera les donnée plus rapidement et plus notre voiture autonome pourra aller plus vite.

1.3 Organisation

Pour réaliser ce projet nous avons commencé par installer sur nos PC OpenCv qui est indispensable pour exécuter le code initial et visualiser les résultats. Ensuite nous avons étudier ce code en commun dans le but de comprendre les principales fonctions pour ensuite se répartir les tâches pour la recherche d'optimisation du code initial et ainsi avoir plusieurs possibilité d'optimisation pour choisir la plus adaptée et plus performante.

1.3.1 Répartition des tâches



Tâches	Réalisation
Etude du code initiale	En commun
Etude d'une optimisation de Sobel	Mohamed et Mustapha
Etude d'une optimisation de Canny	Othman
Etude d'une optimisation de Hough	Othman
Etude sur l'utilisation de PThread	Mustapha et Mohamed
Etude de parallélisation avec OpenMP	Othman
Redaction du rapport	En commun

TABLE 1.1 – Répartition des tâches

2 Algorithmes de traitement d'images

Les algorithmes de prétraitement sont importants dans l'analyse des images pour extraire des informations. Une arête ou contour (edge) est définie par une discontinuité dans les valeurs de niveau de gris. Un pixel dont le niveau de gris est similaire au pixel voisin ne représente, probablement pas, un point du contour. Néanmoins un pixel avec un niveau de gris qui varies largement à celui des pixels voisins peut être considéré comme un point du contour. Une grande partie de la détection des contours est implémentée en utilisant des masques de convolution.

2.1 Sobel Edge Detection

La méthode de Sobel est appliquée pour effectuer la détection des bords. Le détecteur de contour Sobel utilise deux masques de taille 3x3, l'un estimant le gradient dans la direction x et l'autre estimant le gradient dans la direction y. Le masque est glissé sur l'image, manipulant un carré de pixels à la fois. L'algorithme calcule le gradient de l'intensité de l'image en chaque point, puis donne la direction à suivre pour augmenter l'intensité de l'image en chaque point, du clair au foncé. Les zones de bords représentent de forts contrastes d'intensité qui sont plus sombres ou plus clairs.

Les algorithmes de Sobel utilisent une procédure mathématique appelée convolution et analysent généralement les dérivées ou dérivées secondes des nombres numériques dans l'espace. Nous mettons en œuvre la méthode de Sobel pour la détection des bords, qui est basée sur un tableau 3 par 3 qui est déplacé sur l'image principale. Ce tableau est donné par :

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Nous déplaçons les noyaux de Sobel sur un pixel particulier de l'image en entrée. Puis nous calculons une nouvelle valeur. Les noyaux de convolution de Sobel sont conçus pour répondre aux bords verticalement et horizontalement. Ces masques sont chacun convolutés avec l'image.
Nous calculons le gradient horizontal et vertical (Gx et Gy), puis nous les combinons pour trouver la magnitude absolue du gradient en chaque point et l'orientation de ce gradient. Nous utilisons ces chiffres pour calculer la magnitude du bord qui est donnée par :

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

2.2 Canny Edge Detection

Pour mettre en œuvre l'algorithme de détection des bords de Canny, il faut suivre une série d'étapes. Tout d'abord, lisser l'image avec un filtre gaussien. Ensuite, on calcule la magnitude et l'orientation

du gradient en utilisant des approximations de différences finies pour les dérivées partielles. Ensuite, on applique la suppression des non-maxima à l'amplitude du gradient. Après, on utilise l'algorithme du double seuil pour détecter et relier les bords.
Voici la formule mathématique utilisée :

$$G(x,y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

où x est la distance à l'origine sur l'axe horizontal, y est la distance à l'origine sur l'axe vertical, et σ est l'écart de la gaussienne et elle contrôle le degré de lissage. Ensuite, le gradient du tableau lissé $G(x, y)$ est utilisé pour produire les partiels x et y. Une fois que nous avons obtenu les partielles x et y, pn combine les dérivées directionnelles x et y pour obtenir la normale du gradient.
Une fois les directions des bords connues, il faut maintenant appliquer la suppression non maximale. Deux valeurs de seuil sont appliquées à la suppression non-maximale. Avec ces valeurs de seuil, deux images de bord seuillées T1[x, y] et T2[x, y] sont produites. La suppression des non-maxima est utilisée pour tracer le long du bord dans la direction du bord et supprimer toute valeur de pixel (en la mettant égale à 0) qui n'est pas considérée comme un bord. Cela donnera une ligne fine dans l'image de sortie.


Sobel Filter


Canny Filter

2.3 Hough Transform

La transformation de Hough est un algorithme inventé pour reconnaître des lignes complexes dans des photographies (Hough, 1962). Depuis sa création, l'algorithme a été modifié et amélioré pour pouvoir reconnaître d'autres formes telles que des cercles et des quadrilatères de types spécifiques. Pour comprendre le fonctionnement de l'algorithme de la transformée de Hough, il est important de comprendre quatre concepts : l'image de bord, l'espace de Hough et le mappage des points de bord sur l'espace de Hough, une autre façon de représenter une ligne, et comment les lignes sont détectées.

Comme mentionné, un point de contour produit une courbe cosinus dans l'espace de Hough. De ce fait, si nous devions faire correspondre tous les points de contour d'une image contour à l'espace de Hough, cela générerait un grand nombre de courbes cosinus. Si deux points de contour se trouvent sur la même ligne, leurs courbes cosinus correspondantes se croiseront sur une paire spécifique (ρ, θ). Ainsi, l'algorithme de la transformation de Hough détecte les lignes en trouvant les paires (ρ, θ) dont le nombre d'intersections est supérieur à un certain seuil. Il convient de noter que cette méthode de seuillage ne donne pas toujours le meilleur résultat si l'on ne procède pas à un prétraitement tel que la suppression du voisinage dans l'espace de Hough pour éliminer les lignes similaires dans l'image de bord.

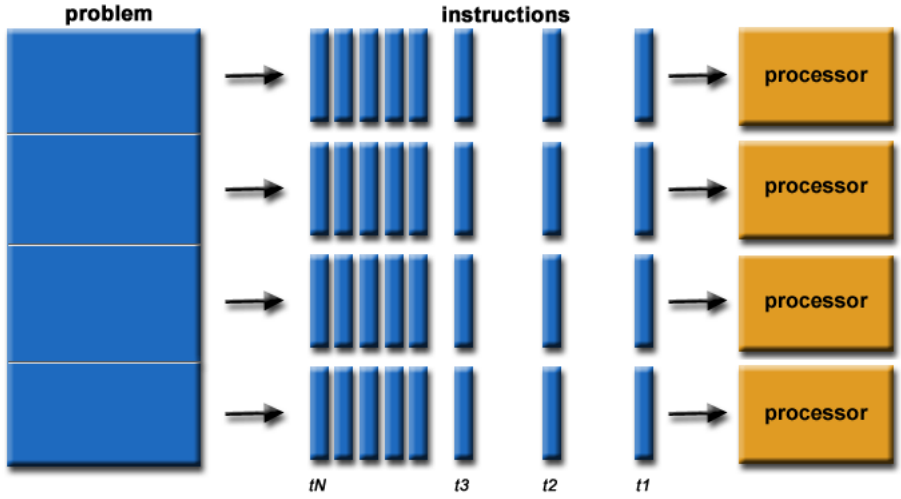

0
100
200
300
400
500
0
200
400
600
800
0
100
200
300
400
500
0
200
400
600
800

3 Parallélisation avec OpenMP

3.1 Concept du Parallélisme

Le traitement parallèle est une méthode informatique qui consiste à utiliser deux ou plusieurs processeurs (CPU) pour traiter des parties distinctes d'une tâche globale. La répartition des différentes parties d'une tâche entre plusieurs processeurs permet de réduire le temps d'exécution d'un programme. Tout système doté de plus d'une unité centrale peut effectuer un traitement parallèle, de même que les processeurs multicœurs que l'on trouve couramment sur les ordinateurs aujourd'hui.

Les processeurs multi-cœurs sont des puces de circuits intégrés qui contiennent deux processeurs ou plus pour de meilleures performances, une consommation d'énergie réduite et un traitement plus efficace de tâches multiples. Ces configurations multi-cœurs sont similaires à l'installation de plusieurs processeurs distincts dans le même ordinateur. La plupart des ordinateurs peuvent avoir entre deux et quatre cœurs, et jusqu'à 12 cœurs.



Le traitement parallèle est généralement utilisé pour effectuer des tâches et des calculs complexes. Les spécialistes des données utilisent généralement le traitement parallèle pour les tâches de calcul et de données intensives.

3.2 OpenMP

OpenMP est un ensemble de directives de compilation ainsi qu'une API pour les programmes écrits en C, C++ ou FORTRAN qui fournit un support pour la programmation parallèle dans les environnements à mémoire partagée. OpenMP identifie les régions parallèles comme des blocs de code qui peuvent être exécutés en parallèle. Les développeurs d'applications insèrent des directives de compilation dans leur code au niveau des régions parallèles, et ces directives indiquent à la bibliothèque d'exécution OpenMP d'exécuter la région en parallèle.
Le programme C suivant illustre une directive de compilation au-dessus de la région parallèle contenant l'instruction printf() :

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
   /* sequential code */
   #pragma omp parallel{
      printf("I am a parallel region.");
   }
   /* sequential code */
   return 0;
}
```

Quand OpenMP rencontre la directive :

```c
#pragma omp parallel
```

Il crée autant de threads que de cœurs de traitement dans le système. Ainsi, pour un système à deux cœurs, deux threads sont créés, pour un système à quatre cœurs, quatre sont créés, et ainsi de suite. Ensuite, tous les threads exécutent simultanément la région parallèle. Lorsque chaque thread sort de la région parallèle, il est terminé. OpenMP fournit plusieurs directives supplémentaires pour exécuter des régions de code en parallèle, y compris la parallélisation des boucles.
En plus de fournir des directives pour la parallélisation, OpenMP permet aux développeurs de choisir parmi plusieurs niveaux de parallélisme. Par exemple, ils peuvent définir manuellement le nombre de threads. Il permet également aux développeurs d'identifier si les données sont partagées entre les threads ou si elles sont privées à un thread.

3.3 Exemple de Multiplication de Matrices

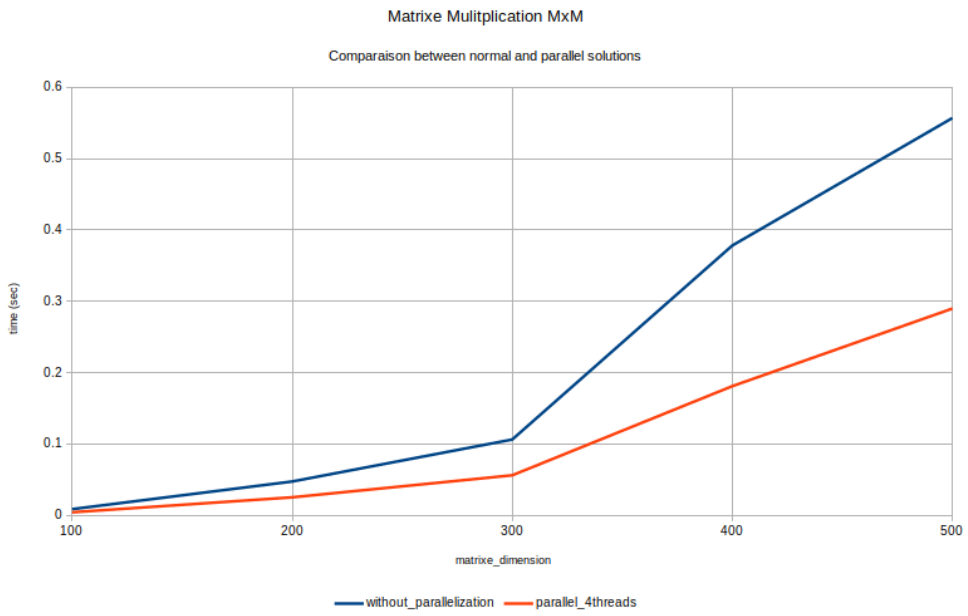
Dans cet exemple, nous allons voir la différence entre la multiplication de matrices sans parallélisation puis avec parallélisation.
Tout d'abord, voyons comment nous pouvons multiplier deux matrices de mêmes dimensions de façon normale :

```c
int i; int j; int k;
int n = 500; //matrixe dimension
for ( i = 0; i < n; i++ ){
  for ( j = 0; j < n; j++ ){
    c[i][j] = 0.0;
    for ( k = 0; k < n; k++ ){
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

Voyons maintenant comment cela pourrait être écrit avec OpenMp :

```c
int i; int j; int k;
int n = 500; //matrixe dimension
#pragma omp parallel shared ( a, b, c, n) private ( i, j, k ) {
  # pragma omp for
  for ( i = 0; i < n; i++ ){
    for ( j = 0; j < n; j++ ){
      c[i][j] = 0.0;
      for ( k = 0; k < n; k++ ){
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
```

L'exécution de ces deux solutions en augmentant les dimensions des matrices nous montre une réelle différence de performance comme nous pouvons le voir dans ce graphique :



4 Parallélisation des algorithmes de traitement d'images

Après avoir définit nos algorithmes ainsi que la façon dont laquelle on utilisera OpenMP afin de paralléliser les tâches sur les différents threads disponible sur notre machine, on est prêt pour une études comparative.

4.1 Parallélisation des algorithmes

Voici le code de Sobel avec la directive Pragma qui permettra une parallélisation :

```
void Sobel1_pragma(Mat frame, int * filterGX,int * filterGY, int size, Mat * out, int
    limit){

  //Convolution Time ! (adding threshold to result might improve performances
    significantly)
  int step = std::floor(size/2);
  float sumX,sumY;

  #pragma omp parallel for private (sumX,sumY) num_threads(NUMTHREADS)
  for(int y=1;y<frame.cols-1;y++){
    for(int x=1;x<frame.rows-1;x++){
      sumX=0;
      sumY=0;

      for(int i=0;i<size;i++){
        for(int j=0;j<size;j++){
          sumX+=filterGX[i*size+j]*frame.at<uchar>(Point(y-step+i,x-step+j));
          sumY+=filterGY[i*size+j]*frame.at<uchar>(Point(y-step+i,x-step+j));
        }
      }
      out->at<uchar>(Point(y,x)) = sqrt(pow(sumX,2)+pow(sumY,2))/4;

      if(sqrt(pow(sumX,2)+pow(sumY,2))/4<limit)
        out->at<uchar>(Point(y,x)) = 0;

    }
  }
}
```

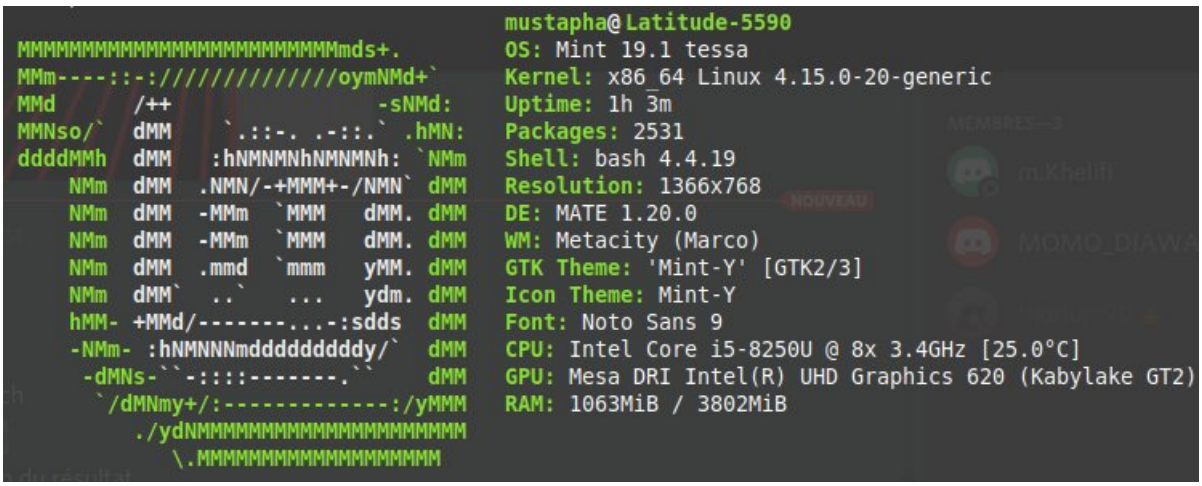
Voici le code de Hough avec la directive Pragma qui permettra une parallélisation :

```
void Hough_pragma(Mat frame, Mat* acc, Mat *f){
  int channels = frame.channels();
  int nRows = frame.rows;
  int nCols = frame.cols * channels;
  //const uchar* image = frame.ptr();
  int step = (int)frame.step;
  int stepacc = (int)acc->step;
  if (frame.isContinuous())
  {
      nCols *= nRows;
      nRows = 1;
```

```
12  }
13
14  int i,j;
15  double rho;
16  #pragma omp parallel for private (rho) num_threads(NUMTHREADS)
17  for( i = 0; i < frame.rows; i++ ){
18    for( j = 0; j < frame.cols; j++ ){
19      if(frame.data[i * step + j]!=0){
20        for(int theta=0;theta<180; theta+=thetaStep){
21          rho = j*cos((double)theta*PI/180)+i*sin((double)theta*PI/180);
22          if(rho!=0)
23            acc->at<ushort>(Point(cvRound(rho) ,(int)cvRound(theta/thetaStep)))+=1;
24        }
25      }
26    }
27  }
28  cv::Point min_loc, max_loc;
29  cv::Point min_loc_old, max_loc_old;
30  double min, max;
31  cv::minMaxLoc(*acc, &min, &max, &min_loc_old, &max_loc_old);
32
33  Point pt1, pt2;
34  double a ,b;
35  double x0, y0;
36  double theta;
37  acc->data[max_loc_old.y * stepacc +max_loc_old.x]=0;
38
39  #pragma omp parallel for private (theta, pt1, pt2, a, b, x0, y0) num_threads(
      NUMTHREADS)
40  for(int i=0;i<40;i++){
41    cv::minMaxLoc(*acc, &min, &max, &min_loc, &max_loc);
42    if(abs(max_loc_old.x-max_loc.x)>5 || abs(max_loc_old.y-max_loc.y)>5){ //might be
      interesting to use that ....
43      theta = (double)max_loc.y*thetaStep;
44      a = cos(theta*PI/180); //compute hough inverse transform from polar to
      cartesian
45      b = sin(theta*PI/180);
46      x0 = a*max_loc.x;
47      y0 = b*max_loc.x;
48      pt1.x = cvRound(x0 + 1000*(-b)); //compute first point belonging to the line
49      pt1.y = cvRound(y0 + 1000*(a));
50      pt2.x = cvRound(x0 - 1000*(-b)); //compute second point
51      pt2.y = cvRound(y0 - 1000*(a));
52      line( *f, pt1, pt2, Scalar(0,0,255), 3, LINE_AA);
53      acc->at<ushort>(Point(max_loc.x ,max_loc.y))=0;
54      max_loc_old.x = max_loc.x;
55      max_loc_old.y = max_loc.y;
56    }
57    else{
58      acc->at<ushort>(Point(max_loc.x ,max_loc.y))=0;
59      i--;
60    }
61  }
62 }
```

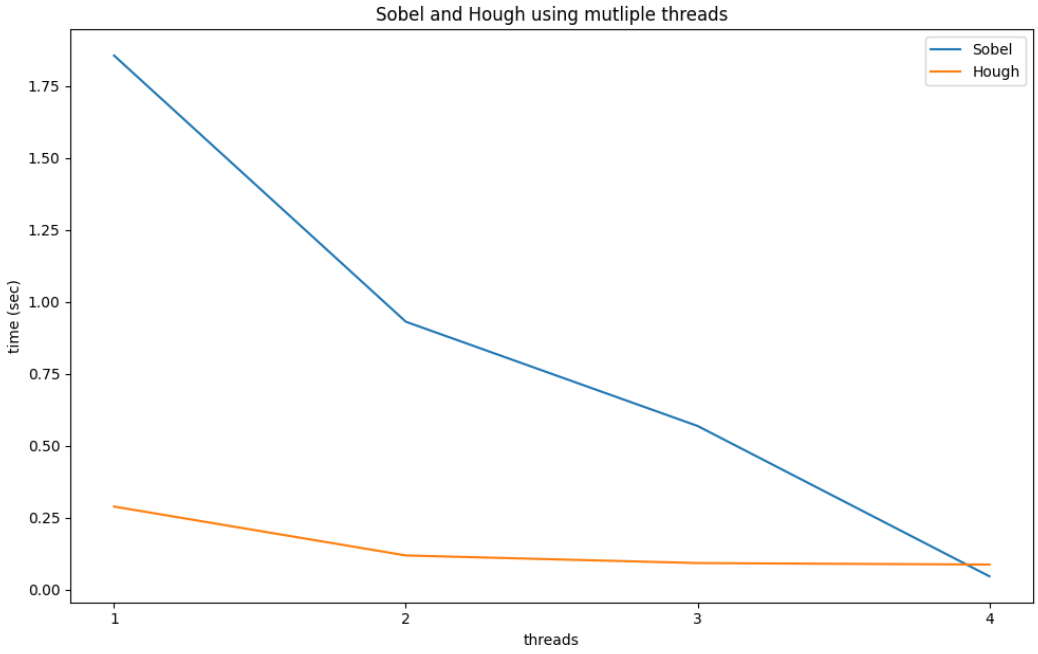
4.2 Configuration de la machine

On sait très bien que le temps d'exécution d'un programme varie d'une machine à une autre tel que la configuration du hardware ainsi que le système d'exploitation influence sur les résultats des tests, pour cela nous utilisant cette configuration de machine et de système :



4.3 Comparaison avec une image en entrée

Finalement on trouve ceci comme différence de temps d'exécution tout en augmentant le nombre de threads pour l'algorithme de Sobel ainsi que Hough en utilisant une image de très grande dimense (3880x2160 pixels) :



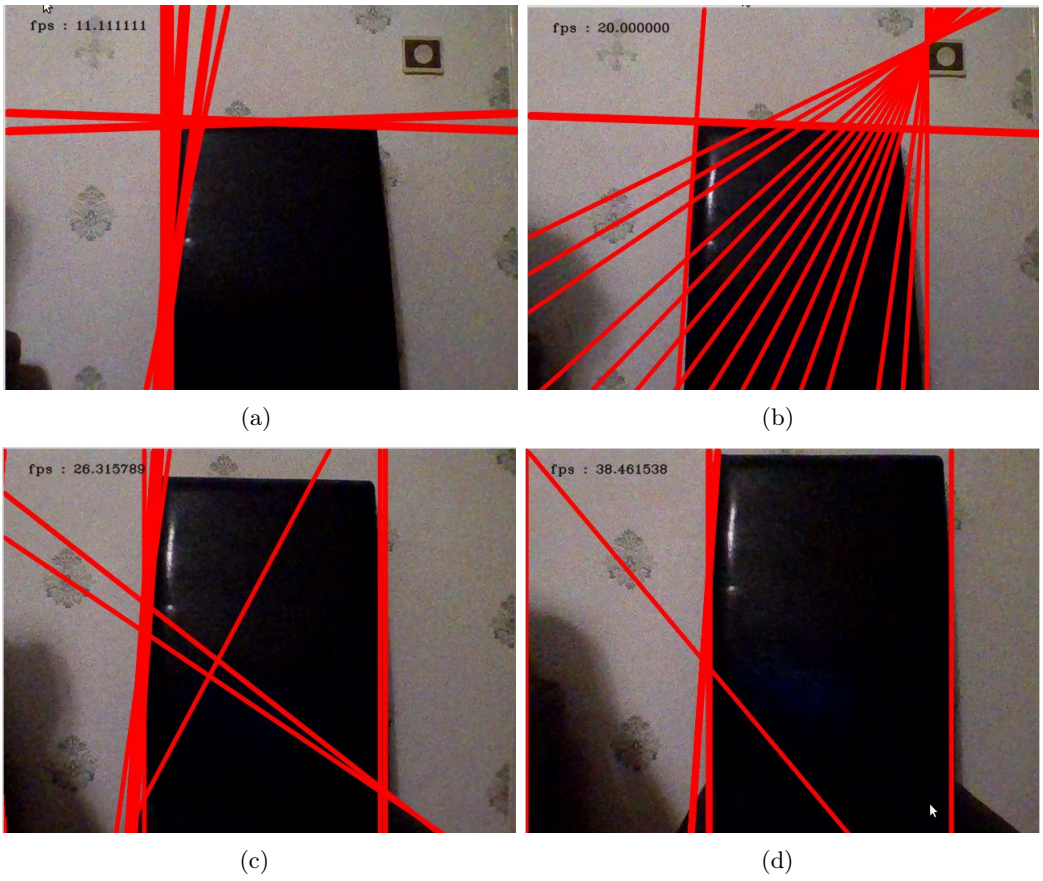
4.4 Comparaison via le flux vidéo de la caméra

Voici des captures d'écrans des tests avec la caméra d'ordinateur tout en augmentant le nombre de threads, ainsi on visualise la valeur fps (frames per seconds) :

4.4.1 Sobel parallélisé + Hough parallélisé

Nombre de threads	FPS
1	11.11
2	20.0
4	26.31
8	38.46

TABLE 4.1 – Différence des FPS en fonction du nombre de threads



(a) (b)

(c) (d)

FIGURE 4.1 – (a) 1 Thread (b) 2 Threads (c) 4 Threads (d) 8 Threads

4.4.2 Canny + Hough simple

Nombre de threads	FPS
1	66.66
2	76.92
4	76.92
8	83.33

TABLE 4.2 – Différence des FPS en fonction du nombre de threads

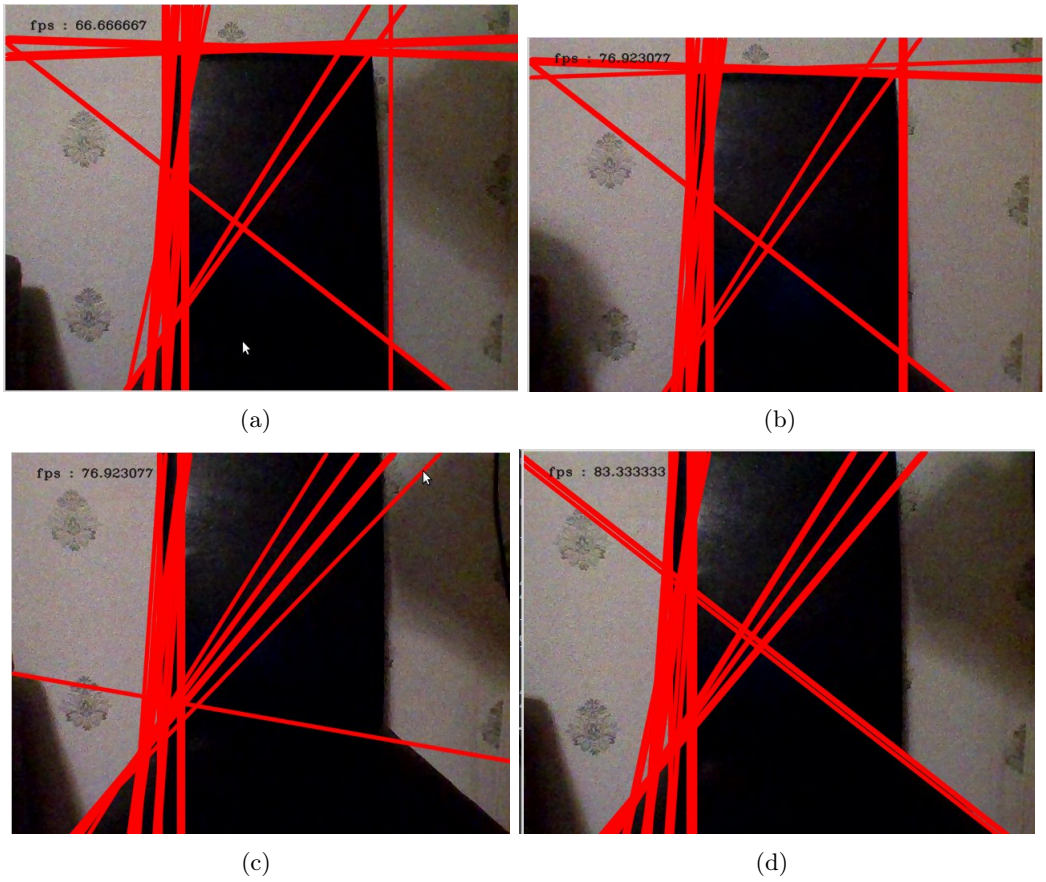


FIGURE 4.2 – (a) 1 Thread (b) 2 Threads (c) 4 Threads (d) 8 Threads

4.4.3 Canny + Hough parallélisé

Nombre de threads	FPS
1	66.66
2	90.90
4	100.00
8	100.00

TABLE 4.3 – Différence des FPS en fonction du nombre de threads

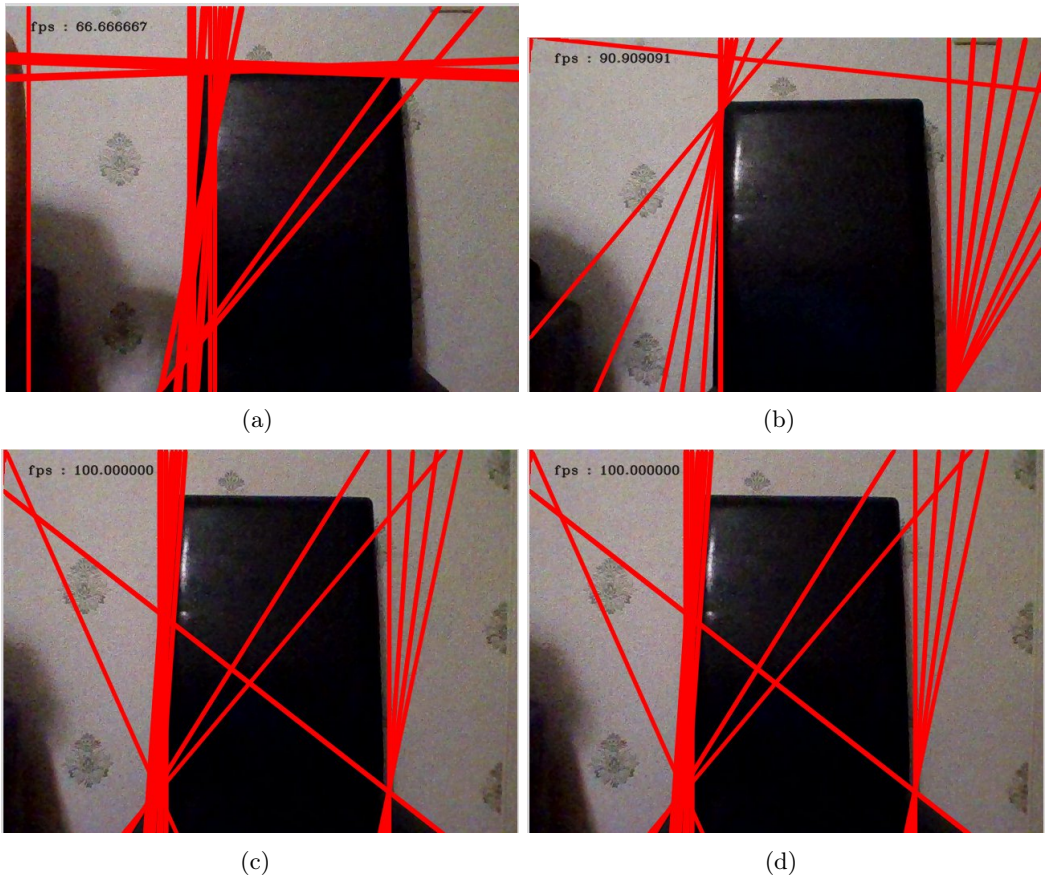


FIGURE 4.3 – (a) 1 Thread (b) 2 Threads (c) 4 Threads (d) 8 Threads