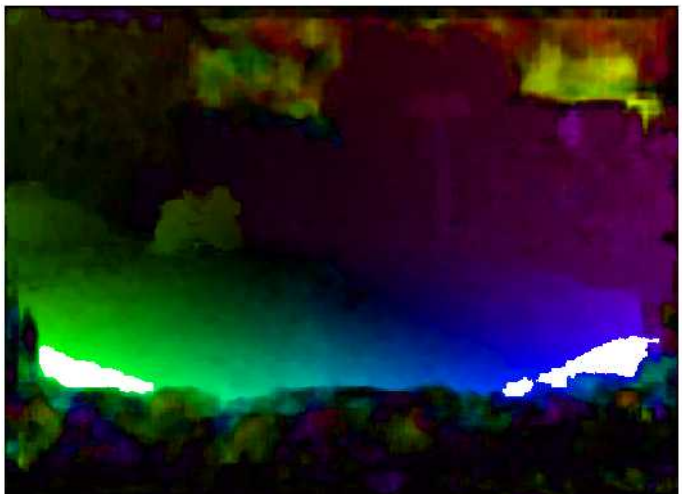
Julien MARZAT

3A ISA - CPDS

Rapport de Stage Ingénieur

Estimation temps réel du Flot Optique



Tuteur ENSEM : Didier WOLF

Tuteur INRIA : André DUCROT

2007-2008

Remerciements

Je tiens tout d'abord à remercier le centre de recherche INRIA Rocquencourt et plus particulièrement l'équipe IMARA pour l'accueil qui m'a été réservé et les très bonnes conditions de travail dans lesquelles j'ai évolué.

Pour son aide précieuse et son soutien je remercie Yann DUMORTIER, Doctorant en vision chez IMARA, qui m'a accompagné tout au long des différentes étapes du projet m'apportant de nombreux conseils.

Enfin, je remercie mon maître de stage André DUCROT, directeur de recherche, pour sa patience et la supervision de mon travail qui ont contribué de manière significative à la réussite de ce stage.

Résumé

Le calcul du flot optique est une étape de bas niveau en traitement d'images, permettant d'estimer le déplacement des objets d'une scène. Dans le cadre du stage, il sert de base à un processus de détection d'obstacles. Toutefois, les estimations denses et précises du flot optique sont habituellement coûteuses en temps de calcul. L'étude s'attachera donc à analyser les techniques existantes de calcul du flot optique, puis à choisir une méthode parallélisable donnant des résultats répondant aux attentes, et finalement à l'implémenter sur GPU (Graphical Processing Unit) à l'aide du nouvel environnement CUDA (Compute Unified Device Architecture) de manière à fournir des estimées en temps réel (environ 10 Hz).

Mots-clés : Vision monoculaire, Flot optique, Calcul parallèle, GPU, CUDA, Tensor Voting

Abstract

Optical Flow computation is a low-level step in Image Processing which gives an estimation of the motion of the objects in a scene. It serves as a basis for an obstacle detection process in the current context. Nevertheless, optical flow estimations providing both dense and accurate results are time-consuming. Therefore the study will take up with the analysis of the existing technics concerning optical flow estimation then with the choice of a parallelisable method giving suitable results and finally with the implementation on GPU (Graphical Processing Unit) in aid of the brand new API CUDA (Compute Unified Device Architecture) in order to provide real-time estimates (about 10 Hz).

Key words : Monocular Vision, Optical Flow, Parallel processing, GPU, CUDA, Tensor Voting

Sommaire

IV – CUDA, Implémentation Temps réel

Introduction

Le développement présent et futur de l'assistance à la conduite, puis à terme de véhicules entièrement automatisés, pose notamment le problème de la perception de l'environnement. Ainsi, il est nécessaire de repérer les objets mobiles pour être en mesure de prévenir tout risque de collision avec le véhicule instrumenté.

Un tel processus de détection peut se faire par le biais d'une seule caméra (vision monoculaire). Dans ce cadre, la première étape du processus consiste à mesurer le déplacement des objets de la scène observée : une estimation de ce déplacement est fournie par le calcul du flot optique.

Le flot optique est un champ de déplacement visuel qui permet d'expliquer des variations dans une image animée en terme de déplacement de points images. Il existe de nombreuses méthodes pour le calculer, toutes ayant en commun de ne pas être temps réel si l'on requiert une grande précision. Le meilleur compromis temps d'exécution/précision fournit en effet une estimée en six secondes.

L'objectif du stage est donc la production d'estimées précises du flot optique en temps réel (10 Hz souhaité) pour servir de base à un processus plus important de détection d'obstacles. L'avènement récent de techniques de calcul parallèle sur carte graphique et notamment de l'environnement de développement CUDA (Compute Unified Device Architecture) par NVIDIA doit permettre de réaliser cet objectif.

Après développement du contexte inhérent au stage, l'étude comportera un état de l'art des approches d'évaluation du flot optique, conclu par un premier tri des méthodes en accord avec le cahier des charges.

Il sera ensuite question d'évaluer les approches restantes de manière à concevoir l'algorithme de calcul du flot optique qui sera utilisé. S'ensuivra une étape de validation finale (Ground Truth). Finalement, je détaillerai l'architecture CUDA et la manière dont le résultat final temps réel a été obtenu.

I – Cadre du stage

1. L'INRIA

L'INRIA Rocquencourt est l'une des six unités de recherche de l'Institut National de Recherche en Informatique et en Automatique (INRIA), organisme de recherche spécialisé dans le domaine des Sciences et Technologies de l'Information et de la Communication. L'INRIA est un établissement public placé sous la double tutelle du ministère de la recherche et du ministère de l'économie, des finances et de l'industrie. Il accueille environ 3 600 personnes dont 2 800 scientifiques et a un budget annuel de l'ordre de 160 M€ dont 20% de ressources propres.

Combiner l'excellence scientifique et le transfert technologique est le fondement de la stratégie de l'institut, dans laquelle s'inscrit en particulier l'unité de recherche de Rocquencourt. Ainsi, les objectifs prioritaires de recherche – concevoir et maîtriser les réseaux et systèmes, traiter l'information distribuée, accroître l'expressivité et la sûreté des langages, concevoir et valider des algorithmes performants, modéliser le vivant – sont conduits avec le souci de concilier recherche amont du meilleur niveau international, applications et valorisation par le biais d'interactions constantes avec le monde socio-économique.

Les activités scientifiques sont menées en développant des partenariats étroits avec les équipes internationales de pointe, le monde de l'industrie et des services, en particulier dans le cadre des pôles de compétitivité, et de nombreux établissements d'enseignement supérieur et de recherche d'Île-de-France.

2. IMARA

Un projet (ou équipe-projet) INRIA est une équipe de recherche de taille limitée, avec une thématique donnée, des objectifs scientifiques bien définis et un chef de projet qui a la responsabilité de coordonner les travaux de l'équipe.

J'ai ainsi effectué mon stage au sein de l'équipe IMARA (Informatique, Mathématiques et Automatique pour la Route Automatisée) dirigée par Michel Parent.

Cette équipe de recherche est un projet "horizontal" à l'INRIA. Il est destiné à coordonner et à transférer les efforts de recherche de l'INRIA qui peuvent être appliqués au domaine de la "Route Automatisée", en particulier dans les domaines suivants :

- le traitement du signal (filtrage, calculs, traitement de l'image, ...)
- le contrôle-commande du véhicule (accélération, freinage, direction)
- les outils de programmation temps réel distribués
- les communications
- la modélisation
- le contrôle et l'optimisation des systèmes de transport.

L'objectif global de ces recherches est l'amélioration du transport routier en terme de sécurité, d'efficacité, de confort et de minimisation des nuisances. L'approche technique est centrée sur les aides à la conduite, pouvant aller jusqu'à une automatisation totale.

Le projet met à la disposition des diverses équipes participantes des moyens importants avec une flotte de cinq véhicules instrumentés, divers capteurs et des moyens de calculs et de simulation.

L'équipe IMARA fait partie du consortium français "La Route Automatisée" en coopération avec l'Ecole des Mines de Paris et est impliquée dans un grand nombre de projets européens sur les aides à la conduite et la gestion du trafic.

D'un autre coté, l'équipe participe activement au développement des "cybercars" (figure 1), véhicules urbains du futur avec conduite totalement automatisée. Une collaboration a été établie avec divers acteurs industriels du secteur comme Frog Navigation Systems (NL), Robosoft (F) et Yamaha Europe (NL).

Figure 1. CyCab

3. Contexte de l'étude : un processus de détection d'obstacles

L'omniprésence de la voiture aujourd'hui amène à s'interroger sur de nombreux points. En effet, l'occupation des véhicules est la plupart du temps très faible (entre 1 et 2 personnes par véhicule), et ceci, couplé à la multiplication des nœuds routiers, entraîne des temps de transit de plus en plus longs. Enfin, le nombre d'accidents a certes diminué de façon significative en France notamment, mais cette régression reste limitée par le facteur humain.

C'est pourquoi, lorsque l'on s'attache à imaginer le véhicule du futur (proche), plusieurs choix apparaissent d'eux-mêmes : le véhicule se doit d'être automatisé pour s'affranchir du facteur humain (et garantir zéro accident, sauf erreur humaine de conception) et de ne plus être individuel mais collectif, à savoir maximiser l'occupation non seulement à un instant donné mais également dans le temps en s'assurant que tout véhicule inoccupé est à nouveau mis à disposition des usagers.

L'automatisation d'un système mobile, outre les problématiques de sécurité et de redondance d'équipement, pose tout d'abord le problème de la perception de l'environnement. Il est en effet indispensable de reconnaître le champ d'action possible du véhicule et de détecter en temps réel les perturbations pouvant nuire à sa progression.

Un processus de détection d'obstacles en vision monoculaire a ainsi été développé dans le cadre de la thèse de Yann Dumortier («*Perception de l'environnement en vision monoculaire pour les véhicules autonomes*») dans laquelle s'inscrit mon stage [41].

Le choix d'une unique caméra est motivé non seulement par le coût de ce type de capteur, largement inférieur à celui d'un laser par exemple, mais également par la facilité d'utilisation. On s'affranchit ainsi de la calibration de deux caméras. On dispose d'informations bidimensionnelles significatives avec une seule image et d'informations tridimensionnelles avec deux images successives : on a ainsi une richesse d'informations comparable à ce qui est disponible en stéréovision.

La détection des obstacles se fait en extrayant deux informations de la séquence d'images acquise : le plan de la route d'une part, les obstacles mobiles d'autre part, ces deux ensembles étant disjoints (figure 2).

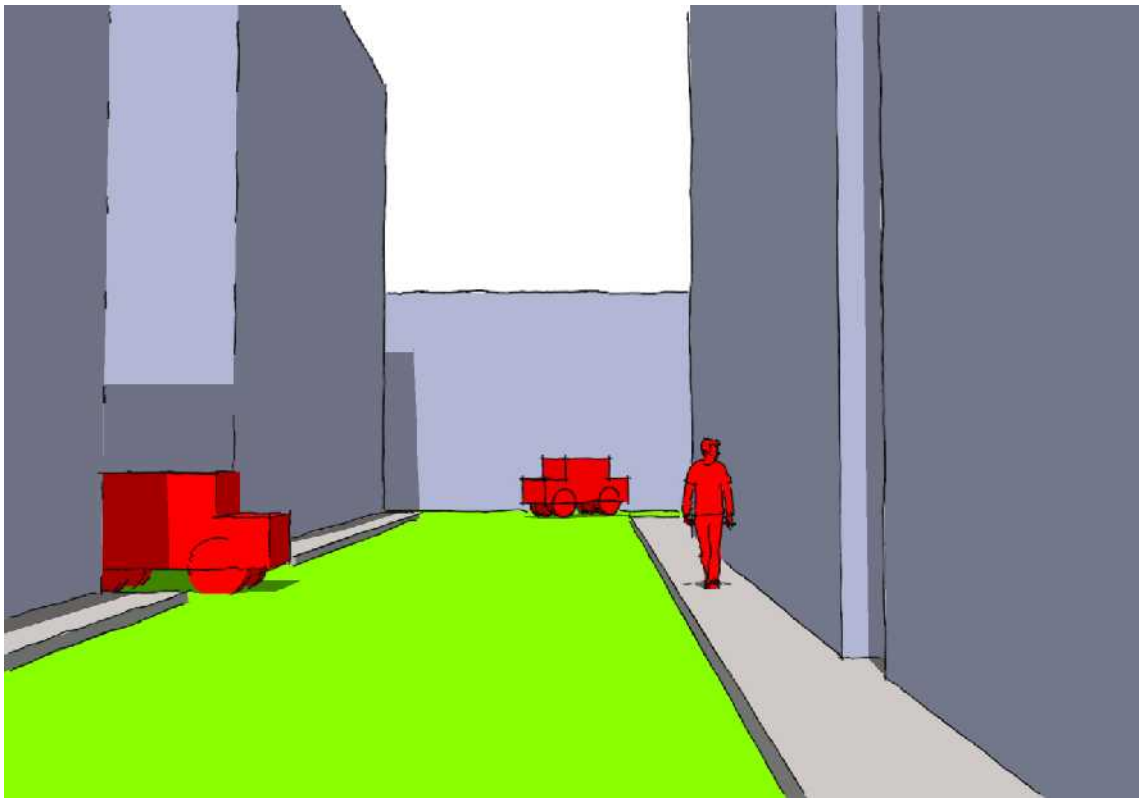


Figure 2. Route et obstacles

La segmentation repose principalement sur l'hypothèse suivante : sur un intervalle de temps donné, les objets à identifier forment des ensembles connexes de vitesse homogène. C'est pourquoi la base du processus est la détermination du flot optique, à savoir le déplacement des pixels d'une image à la suivante, ce qui fournit une estimation du mouvement réel des objets dans la scène.

La détermination du plan de la route s'effectue en estimant l'homographie de ce plan (asservissement visuel) obtenue par la technique RANSAC (RANdom SAmple Consensus : tous les 30 points, 4 points sont tirés au hasard, et l'on discrimine itérativement les inliers et les outliers).

Les objets mobiles sont quant à eux discriminés à l'aide de contraintes de rigidité.

Le processus complet peut être décrit par le schéma suivant :

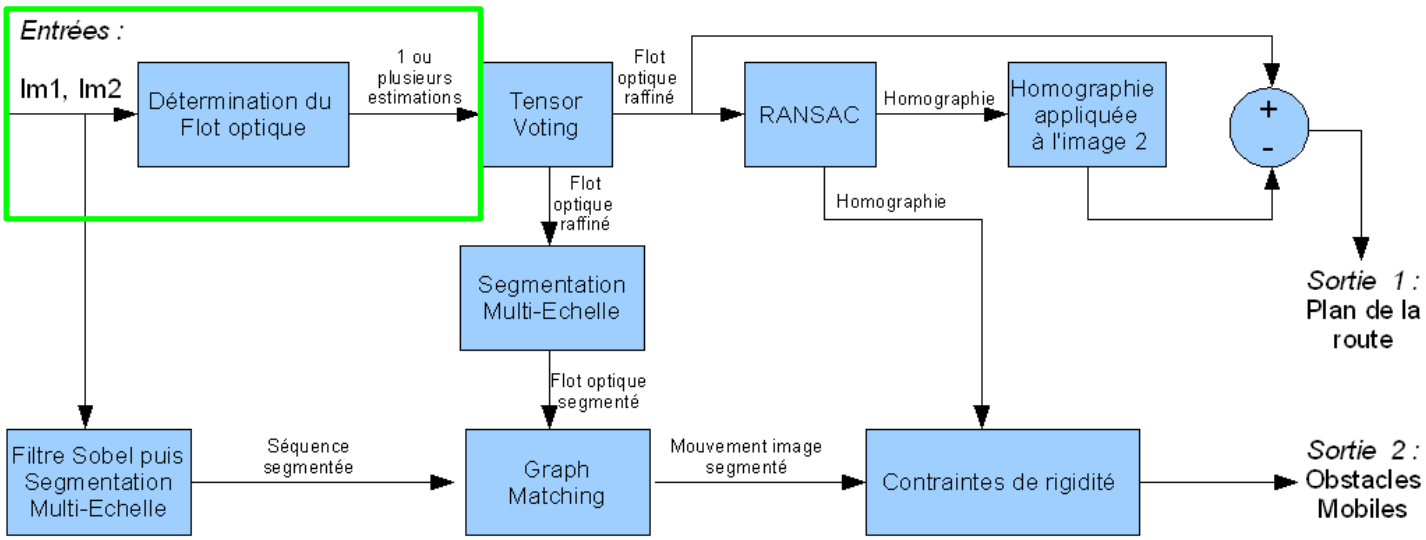


Figure 3. Processus de détection d'obstacles. En vert : partie traitée au cours du stage

4. Cahier des charges et planification

L'étude conduite au cours du stage porte donc sur le choix d'un algorithme de calcul du flot optique en accord avec le processus de détection, c'est-à-dire répondant au cahier des charges suivant :

- L'algorithme doit fournir une estimation du mouvement pour chaque pixel : flot **dense.**
- L'estimation se fait sur **deux images** successives dans une séquence vidéo (N et N-1, flux causal).
- L'estimation doit être aussi **précise** que possible, **subpixelique** (déplacements inférieurs à un pixel identifiés).
- L'estimation doit utiliser le moins de filtrage possible (perte d'information) : le post-traitement effectué par le Tensor Voting [40] effectuant un lissage « intelligent ».
- L'algorithme ne doit pas utiliser de modèle paramétrique ou de décomposition orientée, de manière à n'avoir aucun a priori sur le flot calculé et pouvoir repérer les objets mobiles à une distance la plus lointaine possible.
- Dans un souci de mise en œuvre temps réel à l'aide de GPU (processeur graphique doté de multiples processeurs), l'algorithme doit être **parallélisable.**

Le déroulement logique de l'étude comprend donc un état de l'art des approches pour le calcul du flot optique, suivi d'un choix de méthodes à tester. Les tests conduisent au choix de l'algorithme final. Une fois l'algorithme clairement défini, il s'agit d'étudier sa parallélisation puis de l'implémenter sur GPU à l'aide de l'API CUDA de Nvidia.

Il est possible de résumer le déroulement de l'étude selon le diagramme suivant (figure 4):

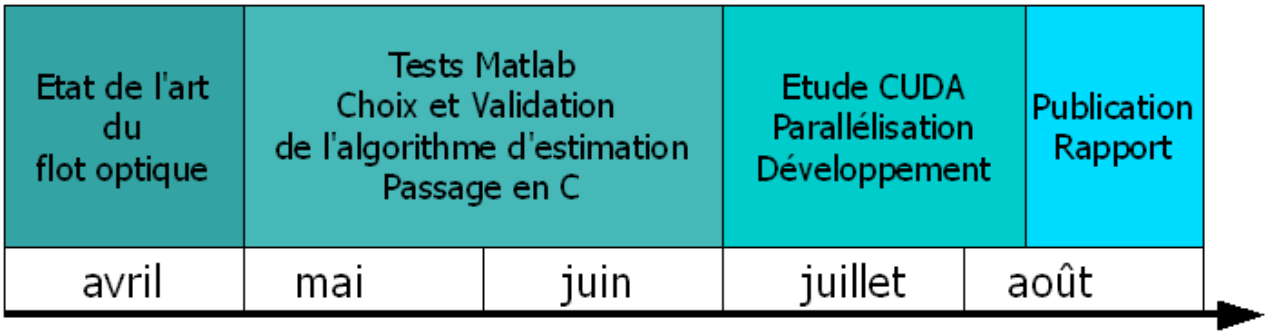


Figure 4. Planification du stage

Après avoir présenté dans cette partie le cadre du stage, ainsi que les actions qui y ont été menées, je vais maintenant détailler l'état de l'art des approches pour l'estimation du flot optique, suivi des premiers choix menés en fonction du cahier des charges.

II – Estimation du flot optique : état de l'art

Dans cette partie, on s'attachera à décrire les approches de calcul du flot optique existante, de manière à effectuer des premiers choix (mise à l'écart de certaines approches). Les méthodes de résolution pour les approches retenues seront détaillées dans la partie III.

1. Définitions et hypothèses

(a) Définitions

Mouvement image : projection du mouvement 3D réel de la scène dans le plan 2D de l'image.

Flot optique : champ des vitesses mesuré à partir des variations de la luminance.

Notations utilisées :

- (x,y) représente les coordonnées d'un pixel d'une image, l'origine étant le coin haut gauche.
- (vx,vy) est la vitesse dans le plan de l'image $vx=\frac{dx}{dt}, vy=\frac{dy}{dt}$. On note $\omega=\begin{bmatrix} vx \\ vy \end{bmatrix}$.
- f1(x, y, t) et f2(x, y, t+dt) sont les deux images successives considérées.
- I représente la valeur de l'intensité lumineuse (entier entre 0 et 255).
- On note $I_u=\frac{\partial I}{\partial u}$ la dérivée de I par rapport à u.
- $\nabla I(x,y)=\begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}$ est le gradient spatial de I
- $\Delta u(x,y)=\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}$ est le laplacien de u
- $\hat{f}$ représente la transformée de Fourier de f

(b) Mesure du flot optique : problèmes et hypothèses

La plupart des méthodes d'estimation du flot optique reposent sur une hypothèse fondamentale : l'intensité lumineuse (ou une autre variable photométrique) se conserve entre deux images successives. [1]

Cela s'écrit sous la forme générale $I(x+\omega,t+1)-I(x,t)=0$.

L'hypothèse de conservation peut également s'écrire sous forme différentielle :

$\frac{dI}{dt}=0$ ce qui conduit à $\frac{dI(x(t),y(t),t)}{dt}=\frac{\partial I}{\partial x}.\frac{dx}{dt}+\frac{\partial I}{\partial y}.\frac{dy}{dt}+\frac{\partial I}{\partial t}=0$

puis $\left[\frac{\partial I}{\partial x} \quad \frac{\partial I}{\partial y}\right].\begin{bmatrix}\frac{dx}{dt}\\ \frac{dy}{dt}\end{bmatrix}+\frac{\partial I}{\partial t}=0$ et donc enfin $\boxed{(\nabla I)^T\omega+I_t=0}$ **(1)**

(1) est appelée *équation de contrainte du mouvement apparent.*

Remarque : L'hypothèse de conservation de la luminance est forte car elle suppose que les variations temporelles de luminance sont uniquement dues au mouvement. On a par ailleurs gardé simplement les termes du premier ordre.

Cette contrainte est insuffisante pour déterminer le flot complet ω car le problème est **mal posé** : on dispose d'une équation linéaire pour deux inconnues. Cela signifie que l'on est simplement en mesure d'évaluer $\omega_\perp$, la vitesse normale dirigée selon le gradient local.

Concrètement, on est en présence de trop de solutions possibles pour évaluer la solution réelle, il faut donc particulariser la solution, ajouter une contrainte de régularisation notamment.

On peut évaluer la vitesse normale de la manière suivante :

Suite à (1), on a $\vec{\omega}=\frac{I_t}{(\nabla I)}$. On définit par ailleurs le vecteur normal $\vec{n}=\frac{\nabla I}{|\nabla I|}$.

La projection de la vitesse sur la normale est alors $s=\omega^T n$.

La vitesse normale vaut $\vec{\omega_\perp}=(\vec{\omega}.\vec{n}).\vec{n}=\frac{-I_t}{\nabla I}.\frac{\nabla I}{|\nabla I|}.\frac{\nabla I}{|\nabla I|}=\frac{-I_t\nabla I}{|\nabla I|^2}$.

Le flot optique est exactement équivalent au mouvement image si les hypothèses suivantes sont respectées :

- éclairage uniforme
- surface à réflexivité Lambertienne
- mouvement de translation pure, parallèle au plan de l'image

Ces conditions ne sont bien sûr jamais rigoureusement respectées dans le cas d'images réelles. On peut toutefois les supposer vérifiées localement.

Le point suivant détaille les améliorations génériques applicables à toute méthode de calcul du flot optique, les paragraphes suivants présenteront les approches en elles-mêmes.

2. Améliorations applicables à toute méthode

Quelle que soit la méthode mise en œuvre, il est toujours possible de l'améliorer en appliquant deux principes qui expriment la même idée à deux échelles différentes :

- **le raffinement itératif** :

Cela consiste à minimiser l'écart entre les deux images successives en exécutant à nouveau l'algorithme après avoir déplacé une des deux images selon le dernier champ de vitesse calculé. Si l'algorithme est stable, la méthode converge.

- **implémentation pyramidale** (figure 5):

On définit une hauteur Lm de pyramide (en pratique, Lm = 2, 3 ou 4). A chaque niveau de la pyramide, on sous-échantillonne l'image d'un facteur 2 pour les deux images successives considérées, le niveau 0 correspondant à l'image initial, le niveau Lm correspondant au niveau le plus grossier. [22]

Au niveau Lm, on calcule le flot optique (avec n'importe quel méthode dense décrite dans cette partie), puis on le propage au niveau inférieur en translatant l'image f1 avec l'a priori calculé au niveau supérieur, avant d'exécuter l'algorithme à nouveau et ainsi de suite jusqu'au niveau 0 qui correspond à l'image initiale. On récupère alors le flot optique final.

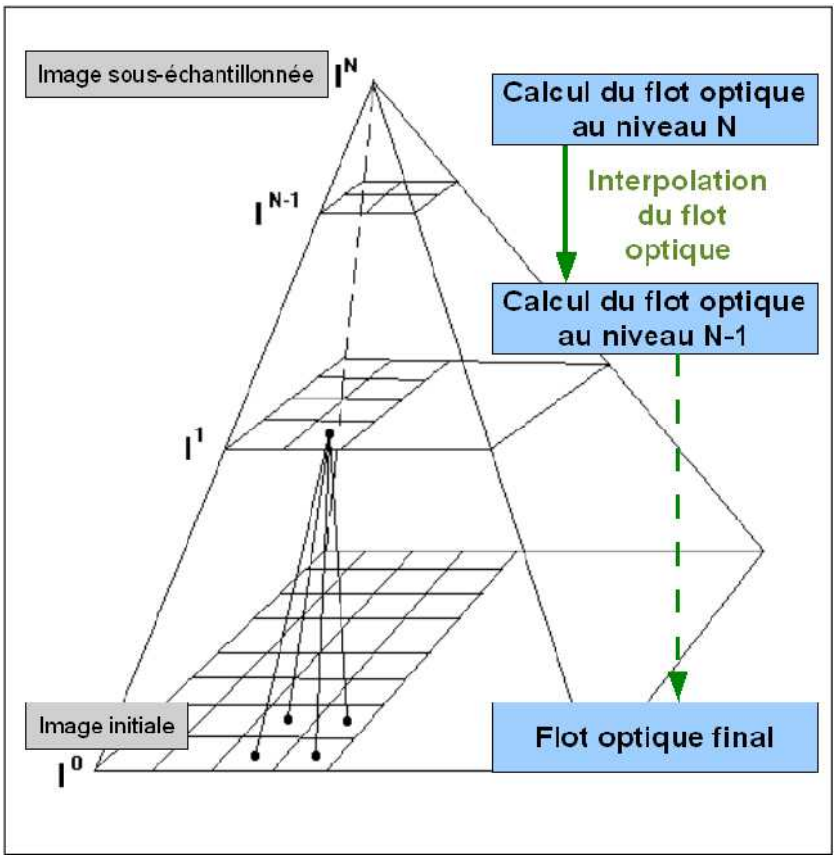


Figure 5. Implémentation pyramidale d'une méthode de calcul du flot optique

Remarque 1 : Construction pratique des pyramides

Pyramide gaussienne : le niveau zéro est l'image originelle. Pour passer au niveau supérieur, on effectue une moyenne pondérée (masque gaussien) des pixels dans une fenêtre 3x3 (ou 5x5) qui devient la valeur du pixel correspondant à la fenêtre au niveau supérieur (filtrage passe-bas), suivi d'un sous-échantillonnage de facteur 2 et ainsi de suite.

$g_0 = I$, $g_k = reduire(g_{k-1})$ $pour\ k \geq 1$

Remarque 2 **:** Une autre amélioration possible est celle proposée par Dériche & Papadopoulo [REF] qui ont remarqué que calculer le flot optique entre f1 et f2 ne donne pas exactement le même résultat (cela dépend toutefois de la méthode) que calculer le flot optique de f2 vers f1. On peut donc se servir de ces informations supplémentaires pour établir une estimée raffinée. Cela double toutefois le temps de calcul, ce qui est un Inconvénient dans un souci d'implémentation temps réel.

3. Approches Variationnelles

Les méthodes basées sur une approche variationnelle consistent à résoudre un problème d'optimisation (local ou global) en minimisant une fonctionnelle, généralement basée sur l'équation (1) à laquelle on ajoute une contrainte pour particulariser les solutions.

(a) Méthodes Variationnelles Globales

Ce type d'approche consiste à minimiser sur le domaine entier de l'image une fonctionnelle prenant en compte l'équation (1) du flot optique ainsi qu'un terme de lissage, c'est à dire en ajoutant une contrainte de régularisation portant sur le gradient, le laplacien (ou ordre supérieur) du champ de vitesse.

La première méthode de calcul du flot optique a été développée en 1980 par Horn & Schunck [7] et consiste à minimiser sur l'ensemble de l'image la fonctionnelle :

$$J_{HS} = \int \int [((\nabla I)^T \omega + I_t)^2 + \alpha((\nabla v_x)^2 + (\nabla v_y)^2)]\, dx\, dy$$

Ce type d'approche a l'avantage de fournir un flot dense (un résultat pour chaque pixel), ce qui est requis par le cahier des charges.

Toutefois, le résultat est assez lisse (la méthode étant globale, les mouvements principaux sont identifiés mais pas les variations locales) et il est difficile de régler les pondérations. A cela s'ajoute les problèmes numériques dûs au choix de la méthode de minimisation (forcément itérative). Ces méthodes nécessitant un nombre assez élevé d'itérations successives (qu'il faudra également régler). Cela ne plaide pas en faveur de la parallélisation de la méthode.

Par ailleurs, les résultats sont assez médiocres : en plus de la non identification des petits mouvements, le calcul est très sensible au bruit.

Il est également possible de régulariser en formulant des hypothèses localement vérifiées.

(b) Méthodes Variationnelles Locales

Les méthodes locales consistent à prendre en compte des hypothèses supplémentaires sur un domaine de taille réduite pour particulariser le flot optique. On minimise alors un critère sur un petit domaine, et on obtient ainsi le flot optique de ce petit domaine.

La méthode locale la plus célèbre est celle de Lucas & Kanade [9] : la vitesse locale est supposée constante sur un voisinage spatial Ω, on minimise alors la fonctionnelle $J_{LK}=\sum_{\Omega} W^2[\nabla I.\vec{\omega}+I_t]^2$ sur ce domaine.

W est une fenêtre locale, pouvant également être interprétée comme la pondération du critère des moindres carrés. On donne généralement une importance plus grande au pixel central (filtrage de type gaussien, facultatif).

Des variantes de cette méthode existent, avec la prise en compte des dérivées secondes, par exemple la contrainte locale supplémentaire $(\nabla\nabla I)\omega^T=-\nabla I_t$.

Uras, Girosi et al. [15] proposent notamment l'utilisation d'une contrainte de constance temporelle du gradient : $\frac{d\nabla I}{dt}=0 \Rightarrow \begin{cases} I_{xx}v_x+I_{yx}v_y+I_{tx}=0 \\ I_{xy}v_x+I_{yy}v_y+I_{ty}=0 \end{cases}$

La résolution de ce système s'effectue alors localement sur des fenêtre de taille réglable. Cette variante a l'avantage de fournir une meilleure discrimination des discontinuités, mais l'hypothèse n'est toutefois pas toujours vérifiée, notamment lorsqu'un objet nouveau apparaît dans la scène.

Les méthodes variationnelles locales sont intéressantes car hautement parallélisables (chaque calcul sur une petite fenêtre est indépendant des autres). Les résultats sont par ailleurs moins sensibles au bruit et permettent le calcul de mouvements locaux.

Il faut toutefois régler convenablement la taille de la fenêtre de recherche, et être conscient que les mouvements de grande taille ne peuvent être repérés si la taille de la fenêtre est trop faible (ce problème sera en partie résolu à l'aide d'une implémentation pyramidale).

(c) Hybride : local – global

La littérature récente présente des méthodes dites « locales globales », que je détaille ci-après.

Utilisons les notations simplifiées suivantes :

$$\omega=\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \nabla u=\begin{bmatrix} u_x \\ u_y \end{bmatrix}, \nabla_3 f=\begin{bmatrix} I_x \\ I_y \\ I_t \end{bmatrix}, J_\rho(\nabla_3 f)=K_\rho*(\nabla_3 f \nabla_3 f^T)$$

où Kρ est une gaussienne centrée en 0 d'écart type ρ.

La régularisation Spatiale s'écrit : $\Phi_{LG}=\int\int([\omega^T J_\rho(\nabla_3 f)\omega]+\alpha(|\nabla v_x{}^2|+|\nabla v_y{}^2|))\,dxdy$.

et la version spatiotemporelle : $\Phi_{LG} = \int_{\Omega x[0,T]} ([\omega^T J_\rho(\nabla_3 f)\omega] + \alpha(\nabla\omega^2 + |\nabla v^2|))\, dxdydt$.

Remarque : cela revient finalement simplement à ajouter un filtrage temporel aux méthodes précédentes, en effet dans le cas où T = 1, on se ramène exactement au problème spatial. Or, comme l'on évalue le flot entre deux images successives, on est dans le cas T=1.

Brox, Bruhn, Papenberg, Weickert [14] (sur un domaine Ω) ont formalisés mathématiquement les choses sous la forme suivante :

Les hypothèses faites consistent en :

- la conservation de l'intensité lumineuse
- la conservation du gradient de l'intensité
- un filtrage global
- une approche multi-échelles (voir implémentation pyramidale)

On minimise alors sur un voisinage la fonctionnelle suivante :

$$J = \int_\Omega \Psi([I(x+v_x, y+v_y) - I(x,y)]^2 + \gamma[\nabla I(x+v_x, y+v_y) - \nabla I(x,y)]^2)\, dxdy$$

On peut ajouter à cette fonctionnelle le terme de régularisation suivant :

$$J_{smooth} = \int_\Omega \Psi(|\nabla v_x^2| + |\nabla v_y^2|)\, dxdy$$

Ce n'est finalement que la réécriture formelle des approches énoncées précédemment, le terme de méthode hybride étant un peu abusif, car certes le lissage et le filtrage temporel peuvent être considérés comme étant effectués sur toute l'image, mais la résolution de l'algorithme reste purement locale. Par ailleurs, les résultats fournis sont très lissés et donc peu exploitables en l'état.

(d) Robustification des critères utilisés

Le terme de lissage utilisée dans la plupart des fonctionnelles quadratiques décrites précédemment ne permet pas réellement la prise en compte des discontinuités spatiales possibles (lissage important). Une solution consiste à utiliser une fonction Ψ de la manière suivante :

$J = \int \Psi(\nabla I).[(\nabla I)^T \omega + I_t]^2\, dxdy$ avec $\Psi(0) = 1, \Psi \underset{(x,y)=>0}{=>} \infty$

On peut même étendre l'approche en résolvant un problème d'optimisation semi-quadratique, par exemple pour Weickert et Schnörr [13] on considère la fonctionnelle :

$\Phi_{LG} = \int_\Omega \Psi_1([\omega^T J_\rho(\nabla_3 f)\omega]) + \Psi_2(|\nabla v_x^2| + |\nabla v_y^2|)\, dxdy$ où Ψ_1 et Ψ_2 sont des fonctions non linéaires convexes, $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$ avec $\epsilon > 0$ petit (10^{-6} à 10^{-3} sont des valeurs couramment utilisées). Ce type de pondération conserve globalement les discontinuités.

Les méthodes variationnelles sont historiquement les premières méthodes de calcul du flot optique. Elles ont l'avantage de reposer sur des bases mathématiques clairement établies et de fournir des résultats denses et cohérents. Je vais maintenant m'atteler à présenter les différentes approches fréquentielles de calcul du flot optique.

4. Approches fréquentielles (énergétiques)

La transformée de Fourier de la séquence d'images que l'on considère est $\hat{I}(f_x, f_y, f_t) = \hat{I}_0(f_x, f_y)\delta(\omega^T [f_x f_y] + f_t)$ où f_x, f_y sont les fréquences spatiales et f_t la fréquence temporelle.

On obtient alors à partir de l'équation de du flot optique : $\boxed{v_x f_x + v_y f_y + f_t = 0}$ **(2)**

(a) Filtrages

Les **Filtres de Gabor** sont des filtres 3D formés par le produit d'une gaussienne et d'une fonction trigonométrique, soit :

$$g(x, y, t) = \frac{1}{(2\Pi)^3 \sigma_x \sigma_y \sigma_t} e^{\frac{-x^2}{2\sigma_x^2} \frac{y^2}{2\sigma_y^2} - \frac{t^2}{2\sigma_t^2}} \cos(2\Pi(f_{x0}x + f_{y0}y + f_{t0}t))$$

où : $(\sigma_x, \sigma_y, \sigma_t)$ est l'écart type de la gaussienne et (f_{x0}, f_{y0}, f_{t0}) la fréquence centrale du filtre. Ces filtres sont passe bande.

La transformée de Fourier de ce filtre est une gaussienne centrée en (f_{x0}, f_{y0}, f_{t0}) , et d'écart type $\left(\frac{1}{\sigma_x} \frac{,1}{\sigma_y} \frac{,1}{\sigma_t}\right)$.

Remarque : Le filtre peut également s'écrire avec une fonction sinus. La somme des carrés des réponses à deux filtres de Gabor en quadrature (un cos et un sin) donne une mesure d'énergie indépendante de la phase du signal.

Sur cette base, plusieurs méthodes ont été développées :

- Heeger [3]: Il s'agit d'utiliser une pyramide gaussienne pour évaluer différents ordres de grandeur de vitesse. Chaque niveau de la pyramide est filtré par une famille de 12 filtres de Gabor, puis on effectue une estimation au sens des moindres carrés sur l'énergie des filtres pour satisfaire l'équation fréquentielle du flot optique (2).

- Méthode « rapide » en utilisant des triades de filtres de Gabor (écart type de la gaussienne et fréquences centrales spatiales identique pour les 3, mais fréquence temporelle différente). Cette méthode est très imprécise car elle est conçue pour analyser un mouvement global plutôt que pour détecter des petits déplacements.

- Weber et Malik [5] : Soit une famille de n filtres spatiotemporels $W_i(x, y, t)$ (gaussiennes). On convolue la séquence des deux images avec ces filtres, on obtient donc n images $f_i = f * W_i$. On est donc ensuite en présence du système suivant : $\begin{bmatrix} f_{1x} f_{1y} \\ f_{2x} f_{2y} \\ \dots \\ f_{nx} f_{ny} \end{bmatrix} \omega = \begin{pmatrix} -f_{1t} \\ -f_{2t} \\ \dots \\ -f_{nt} \end{pmatrix}$

  soit $A\omega = -I_t$ ou enfin $\begin{bmatrix} A & I_t \end{bmatrix} \begin{bmatrix} \omega \\ 1 \end{bmatrix} = 0$. On est en présence d'un système surdéterminé, l'estimation au sens des moindres carrés totaux donne : $\omega = -(A^T A - \sigma I)^{-1} A^T I_t$ où σ est la plus petite valeur singulière de $\begin{bmatrix} A I_t \end{bmatrix}$.

Cette méthode est séduisante car elle propose une alternative à la régularisation en proposant un système surdéterminé dans le domaine fréquentiel. Toutefois le filtrage excessif entraîne toujours une perte d'information préjudiciable, et ce pour les trois méthodes présentées dans ce paragraphe.

(b) Phase

Fleet & Jepson [34] :

On utilise des familles de filtres de Gabor orientés spatialement selon des directions différentes. Le résultat R de la convolution de la séquence avec un filtre est complexe, on peut donc l'écrire sous la forme $R = \rho e^{(i\Phi)}$ avec $\rho = |R| et \Phi = arg(R)$.

La vitesse normale au contour de phase est alors $v_n = s n$ avec $s = \frac{-\Phi_t}{|(\nabla \Phi)|} et \boldsymbol{n} = \frac{\nabla \Phi}{|\nabla \Phi|}$

où Φ_t est la dérivée temporelle de la phase.

Remarque : le calcul du gradient spatial se fait ainsi : $\nabla \Phi = \frac{\Im(R^* \nabla R)}{|(R)|^2}$.

On considère alors les surfaces espace-temps de phase constante, solutions de $\Phi = c$. En dérivant par rapport au temps, on a $\nabla \Phi . \omega_\perp = 0$

La solution est donc donnée par $\omega_\perp = \alpha . \boldsymbol{n} avec \alpha \in R$.

L'inconvénient de cette méthode est qu'elle ne fournit pas l'estimée dense du flot optique mais simplement une estimée de la vitesse normale.

(c) Corrélation de phase [35]

Dans le domaine de Fourier (fx et fy variables dans l'espace de Fourier), f1 et f2 étant deux blocs candidats à l'appariement, de même dimension, appartenant aux deux images successives que l'on considère : $\hat{f}_2(f_x, f_y) = \hat{f}_1(f_x, f_y) e^{-i(f_x v_x + f_y v_y)}$ puis $\frac{\hat{f}_2 \hat{f}_1^*}{|\hat{f}_2 \hat{f}_1^*|} = e^{-i(f_x v_x + f_y v_y)}$.

La solution du problème est donnée en considérant la surface de corrélation de phase :

$c_{t,t+1}(v_x, v_y) = F^{-1}\left(\frac{\hat{f}_2 \hat{f}_1^*}{|\hat{f}_2 \hat{f}_1^*|}\right)$ où F^{-1} désigne l'opérateur transformée de Fourier inverse.

Une estimée du mouvement est alors $[v_x, v_y] = argmax(\Re(c_{t,t+1}))$.

Cette méthode ne renvoie pas un résultat dense (seuls les maximas locaux sont considérés), et elle donne (dans sa version initiale) des déplacements entiers, ce qui va à l'encontre du cahier des charges en terme d'estimées subpixeliques. Ces méthodes fréquentielles sont intéressantes car elles permettent l'extraction de maximas non présents dans le domaine spatiotemporel, de plus l'algorithme de FFT (Fast Fourier Transform) est hautement parallélisable. Toutefois ces méthodes requièrent un certain nombre de prétraitements (filtrage successifs) et fournissent des résultats précis mais non denses (mouvements très bien estimés pour certains pixels). Les résultats denses qui pourraient être obtenus par ces méthodes sont malheureusement imprécis (petits déplacements non identifiés, mouvements globaux accentués).

(d) Approche Ondelettes

- *Méthode Bernard* [36]*:*

On repart de l'équation initiale du flot optique (1), on en fait le produit scalaire avec une famille d'ondelettes $(\Psi^n)_{n=1..N}$ translatée de U=(u1,u2) dans l'espace. Soit X = (x,y) Cela donne le système de N équations : $\int\int [\frac{\partial I}{\partial x} v_x + \frac{\partial I}{\partial y} v_y + \frac{\partial I}{\partial t}] \Psi^n(X-U)\, dxdy = 0 \; pour \; n = 1..N$ (N)

Si on pose $\Psi^n_u = \Psi^n(X-U)$ et $\langle\rangle$ le produit scalaire, (N) devient :

$\langle \frac{\partial I}{\partial x} v_x, \Psi^n_u \rangle + \langle \frac{\partial I}{\partial y} v_y, \Psi^n_u \rangle + \langle \frac{\partial I}{\partial t}, \Psi^n_u \rangle = 0$ pour n =1..N

On peut intégrer par parties chaque produit : $\int \frac{\partial I}{\partial x} \Psi^n_u dx = [I\, \Psi^n_u]_{-\infty}^{+\infty} - \int \frac{\partial \Psi^n_u}{\partial u} I\, dx = \langle I, \frac{\partial \Psi^n_u}{\partial u} \rangle$

car Ψ^n_u est nulle aux bornes infinies, et I est bornée.

On fait par ailleurs l'approximation que la vitesse v est constante sur la longueur de chaque ondelette (conservation locale).

On a donc finalement le système d'équations :

$$\begin{cases} \langle I, \frac{\partial \Psi^1_u}{\partial u} \rangle v_x + \langle I, \frac{\partial \Psi^1_u}{\partial u} \rangle v_y = \langle \frac{\partial I}{\partial t}, \Psi^1_u \rangle \\ .... \\ \langle I, \frac{\partial \Psi^N_u}{\partial u} \rangle v_x + \langle I, \frac{\partial \Psi^N_u}{\partial u} \rangle v_y = \langle \frac{\partial I}{\partial t}, \Psi^N_u \rangle \end{cases}$$

Approximations pour la résolution :

$\frac{\partial I(t+\frac{1}{2})}{\partial t} = I(t+1) - I(t)$ *et* $\frac{\partial \Psi}{\partial x}.\omega = \Psi(x+v_x, y+v_y) - \Psi(x,y)$

Cette méthode nécessite des calculs des différences entre images, des convolutions (utilisation de trois filtres) et hiérarchisation. Filtres utilisés : Deslauriers-Dubuc. Cette méthode est locale.

- *Décomposition des vitesses sur des ondelettes particulières*

Plutôt que de projeter les équations sur une base d'ondelettes, Szeliski propose de décomposer les vecteurs vitesse recherché vx et vy sur une base de fonctions, de la manière suivante :

$v_x(x_i, y_i) = \sum_j \hat{u}_j B_j(x_i, y_i)$, les Bj étant des fonctions non nulles sur un certain intervalle et nulles ailleurs, translatées spatialement les unes par rapport aux autres.

Le calcul du flot optique se résume alors à évaluer les $\hat{u}_j$ *et* $\hat{v}_j$ en minimisant le critère quadratique habituel (méthode de résolution Levenberg-Marquardt par exemple).

On couple à cela une technique pyramidale (estimation à chaque niveau et transmission au niveau de raffinement supérieur).

- *YuTeWu, Kanade, Cohn, ChungLi [37] :*

C'est le même style de décomposition que Szeliski, sur une autre B-spline.

La fonction d'échelle de base est $\phi(x)=\frac{1}{6}\sum_{j=0}^{4}\binom{4}{j}(x-j)_{+}^{3}, avec\ x_{+}^{n}=\begin{cases}x^{n}\ si\ x\geq 0\\ 0\ si\ x<0\end{cases}$

qui donne lieu aux ondelettes : $\Psi(x)=\frac{-3}{7}\phi(2x)+\frac{12}{7}\phi(2x-1)-\frac{3}{7}\phi(2x-2)$.

Les translations et dilatations se font comme suit :

$\phi_{j,k}(x)=\phi(2^{j}x-k),\ j\geq 0, k=-2..2^{j}L-2$

$\Psi_{j,k}(x)=\Psi(2^{j}x-k),\ j\geq 0, k=-1..2^{j}L-2$, j étant le niveau de raffinement.

On peut décomposer n'importe quelle fonction d(x) sur ces ondelettes de la manière suivante :

$$d(x)=d_{-1}(x)+d_{0}(x)+...+d_{j}(x) \quad \text{avec} \quad \begin{cases} d_{-1}(x)=\sum_{k=-2}^{L-2} c_{-1,k}\phi_{0,k}(x) \\ d_{j}(x)=\sum_{k=-2}^{2^{j}L-2} c_{j,k}\Psi_{j,k}(x) \end{cases} .$$

Il est possible d'étendre facilement cette décomposition en deux dimensions, pour décomposer vx et vy, en utilisant les produits tensoriels décrits en annexe 1.

Le problème revient alors à calculer les différents coefficients de ces projections, en minimisant un critère quadratique de la forme $E=\sum_{x,y}(f_{2}-f_{1})^{2}$ et en minimisant itérativement l'erreur quadratique. (Résolution : Levenberg-Marquardt).

Les méthodes basées sur les ondelettes sont très populaires en traitement d'image. Dans le cadre du flot optique, elles sont équivalentes aux décompositions sur des familles de filtres présentées précédemment et induisent donc un filtrage excessif en plus d'une paramétrisation a priori (choix des ondelettes) du calcul.

5. Approches basées sur la corrélation

Block Matching :

A partir d'un bloc de taille donnée de l'image, on cherche le déplacement de ce bloc (approximation de la vitesse) entre les deux images. On autorise pour cela un déplacement maximal et on cherche (différentes techniques d'exploration) le bloc qui correspond au mieux au bloc initial, de manière à minimiser un critère d'erreur (corrélation).

Les critères de corrélation les plus souvent employés sont les suivants :

- $\int_{\Omega} f_{1}(x+v_{x},y+v_{y})f_{2}(x,y)\,dxdy$ corrélation la plus simple
- $SSD=\sum\sum(I(i,j)-J(i+u,j+v))^{2}$ *Sum of Squared Differences*.
- $SAD=\sum\sum|I(i,j)-J(i+u,j+v)|$ *Sum of Absolute Differences*.

- Variante induisant un lissage local par une fonction : $\int_{\Omega} [\phi(f_{1,} f_{2}) + \lambda(\nabla \omega \nabla \omega^{T})] dx\, dy$

De nombreux algorithmes d'exploration ont été développés [24][25][26][27], le plus simple étant l'exploration de tous les blocs de la fenêtre de recherche.

L'inconvénient majeur des méthodes de block matching est qu'elles postulent un déplacement maximal sur une certaine fenêtre, on exclue ainsi les grands déplacements et le résultat est biaisé. Une solution consiste donc à utiliser une pyramide comprenant des niveaux de raffinement différents. C'est notamment ce qu'utilise Anandan [22], en appliquant la méthode d'implémentation pyramidale.

Le résultat (dense lorque l'on effectue la recherche en chaque pixel) fourni a une précision de l'ordre du pixel (on calcule une distance entière d'un pixel à l'autre), cela peut toutefois être amélioré en calculant le flot sur l'image sur-échantillonnée. Les méthodes de Block Matching sont donc à considérer avec intérêt, les résultats fournis étant denses, potentiellement subpixeliques et sans pré-traitement lourd. Enfin, l'algorithme est hautement parallélisable car chaque recherche est indépendante des recherches voisines, ce qui permet également d'identifier les faibles mouvements).

Autres méthodes de calcul du flot optique basées sur la corrélation :

- Modèles *contour* : après extraction d'éléments particuliers de l'image (côtés, coins) à l'aide de filtres, de manière pratique, on peut considérer par exemple que les coins sont des pixels dont l'intensité est très différente de celles des voisins. On cherche ensuite à localiser ces éléments dans l'image suivante à l'aide d'un calcul de corrélation. Les résultats peuvent être extrêmement faussés dans le cas d'occlusion de figures caractéristiques d'une image à l'autre, ou de trop nombreux éléments caractéristiques (qui ne le sont donc plus).

- Prise en compte d'un a priori sur le déplacement (flot affine, surface plane...) de manière à privilégier une direction de recherche.

Ces deux types de méthode dérogent clairement au cahier des charges car elles ne sont ni denses, ni dépourvues d'apriori.

6. Approches retenues

On rappelle que le cahier des charges impose comme résultat un flot dense (une estimation pour chaque pixel), le plus précis possible (subpixelique), dépourvu de paramétrisation du flot et de filtrages excessifs.

Du fait de ces contraintes, on ne retiendra pas les méthodes fréquentielles, qui donnent des pics locaux du flot et donc un résultat non dense. Les méthodes par filtrage (spatiotemporel, ondelettes...) sont également écartées du fait d'une part du lissage trop important qu'elles entraînent et d'autre part du nombre trop important de paramètres à régler.

Les approches répondant au cahier des charges sont donc les méthodes variationnelles (globales et locales) d'une part et les méthodes de block matching (corrélation) d'autre part, tout en analysant l'intérêt de l'information couleur.

III – Estimation du flot optique : Tests

Dans cette partie, il s'agit de tester les approches retenues de manière à affiner le choix jusqu'à obtenir un algorithme satisfaisant au mieux les conditions voulues.

1. Représentation du flot et références

Pour être à même de comparer les différentes estimations obtenues, il faut utiliser la même représentation visuelle du champ de vitesses pour toute les méthodes.

De nombreuses solutions sont possibles pour représenter visuellement le flot optique calculé. La plus couramment utilisée est le tracé du champ (en prenant 1 point sur 10 par exemple) des vecteurs vitesses superposé à l'image initiale, l'inconvénient étant que l'on ne visualise pas le champ dense, et donc que les points aberrants risquent d'être masqués ou au contraire mis en valeur, conduisant à une interprétation biaisée. On peut également représenter en niveaux de gris respectivement u et v. La solution la plus intéressante est l'utilisation d'une carte de couleurs, qui permet de représenter la direction du flot ainsi que son intensité de manière dense.

La carte des couleurs que nous utiliserons est la suivante (Figure 6). On présente également le tracé du champ de vitesse comme on le trouve dans la littérature, ce qui permet de bien comprendre la carte :

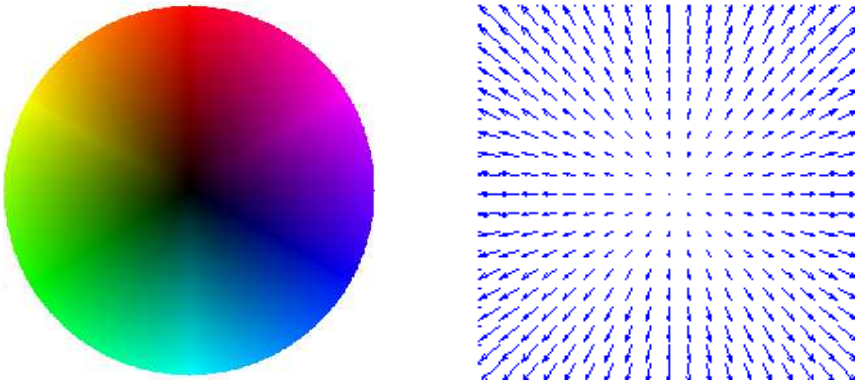


Figure 6. Colormap et affichage du même champ de vitesse sous forme de vecteurs

Les vecteurs vitesses sont représentés par les couleurs contenues dans ce cercle. Chaque vecteur vitesse est codé par la couleur qu'il indique en plaçant son origine au centre du cercle. En fonction de l'angle le point prend une couleur différente, et son intensité varie du noir à la couleur complète selon la norme du vecteur. Il faut également donner une vitesse maximale pour l'affichage au delà de laquelle l'affichage est alors un point blanc. Sinon, Cette cartographie revient finalement à construire une application qui associe à l'angle et à la norme du vecteur vitesse un point dans l'espace (R,G,B).

Ce type de représentation permet une représentation dense (une couleur pour chaque pixel), et permet de visualiser très rapidement la cohérence des résultats, la sensibilité au bruit de la méthode testée et l'aspect (lissage, discontinuités ...) du flot [code : Annexe I].
Le logiciel choisi pour effectuer les premiers tests est Matlab, principalement pour la facilité de manipulation des matrices (images 2D) et la toolbox très développée Image Processing.

La séquence-test utilisée est la séquence réelle sur laquelle on cherche à travailler, à savoir une vidéo (résolution : 640x480) prise dans le domaine de l'INRIA. On travaille plus particulièrement sur les deux images successives présentées ci-après.

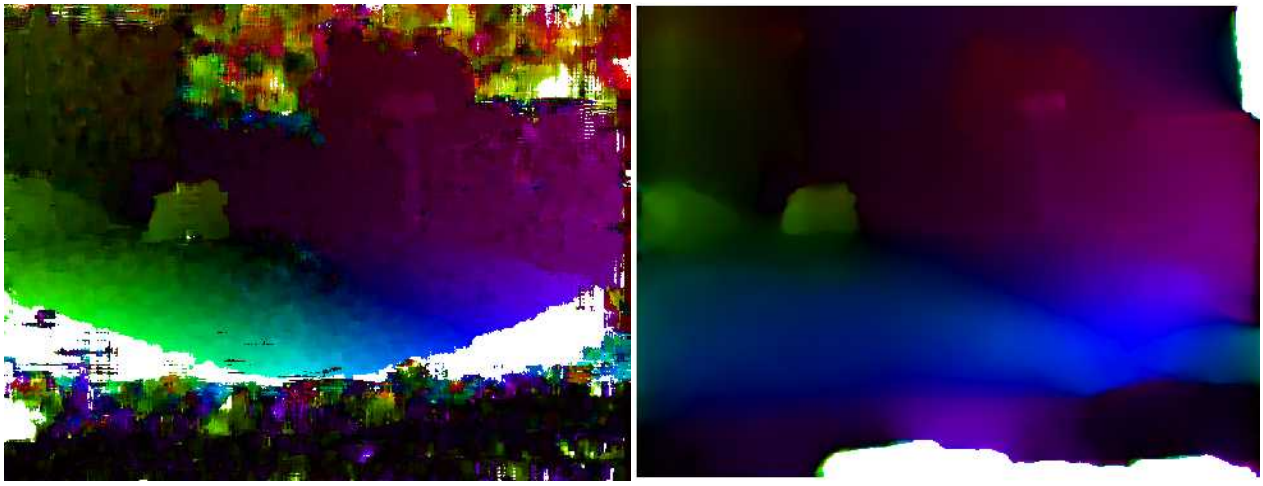
Image 1 Image2

Figure 7. Images de référence

Ces deux images ont l'avantage de présenter de nombreux points caractéristiques de l'application future du processus : l'environnement est urbain, il y a un véhicule arrivant en face qu'il faut détecter.

Les meilleures implémentations existantes (couramment utilisées) sont celles disponible dans la librairie OpenCV [12] codée en C d'une part et l'algorithme de Brox/Weickert (Combiné Horn Schunk / Lucas Kanade) d'autre part. Ces deux méthodes sont de type Lucas & Kanade :

On trace le flot pour une taille de patch 10x10, 5 niveaux, 10 itérations) :



OpenCV Brox/Weickert

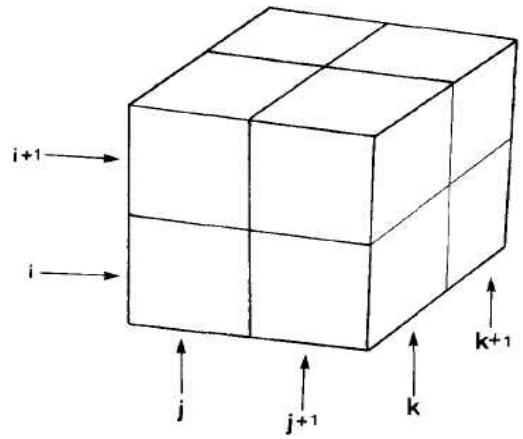
Figure 8. Flots optiques de référence

Le flot de Brox est volontairement très lissé et fournit un résultat globalement correct mais très lisse et imprécis. OpenCV fournit un résultat très bon, excepté quelques anomalies dues au calcul des dérivées et au mode d'interpolation. Ce sera notre image de référence.

2. Tests : Méthodes Variationnelles

(a) Calcul des dérivées

Les méthodes variationnelles sont basées sur l'équation du flot optique et utilisent donc les dérivées temporelles et spatiales de l'image, qu'il faut estimer numériquement. De nombreux masques de convolution sont possibles pour calculer ces dérivées, les masques les plus couramment utilisés sont ceux de Sobel ou Prewitt.
On utilisera, dans tous les tests, le masque proposé par Horn and Schunck, mieux adapté au calcul du flot optique, qui considère un pixel centré dans l'espace et le temps.



Ix = ¼ [I(i,j+1,k) – I(i,j,k) + I(i+1,j+1,k) – I(i+1,j,k) + I(i,j+1,k+1) – I(i,j,k+1) + I(i+1,j+1,k+1) – I(i+1,j,k+1)]

Iy = 1/4[I(i+1,j,k) – I(i,j,k) + I(i+1,j+1,k) – I(i,j+1,k) + I(i+1,j,k+1) – I(i,j,k+1) + i(i+1,j+1,k+1) – I(i,j+1,k+1)]

It = 1/4[I(i,j,k+1) – I(i,j,k) + I(i+1,j,k+1) – I(i+1,j,k) + I(i,j+1,k+1) – I(i,j+1,k) + I(i+1,j+1,k+1) – I(i+1, j+1, k)]

Figure 9. Dérivées selon Horn & Schunck

(b) Méthode d'Horn et Schunck

La première méthode testée est la méthode « historique » de calcul du flot optique, proposée par Horn et Schunck [7] en 1980.
On cherche à minimiser $J_{HS}=\int\int[((\nabla I)^T\omega+I_t)^2+\alpha((\nabla v_x)^2+(\nabla v_y)^2)]dx\,dy$ sur toute l'image.
En pratique, la résolution se fait par la méthode de Gauss-Siedel, qui revient à résoudre les équations d'Euler-Lagrange équivalentes.

Le résultat est alors donné de manière itérative :

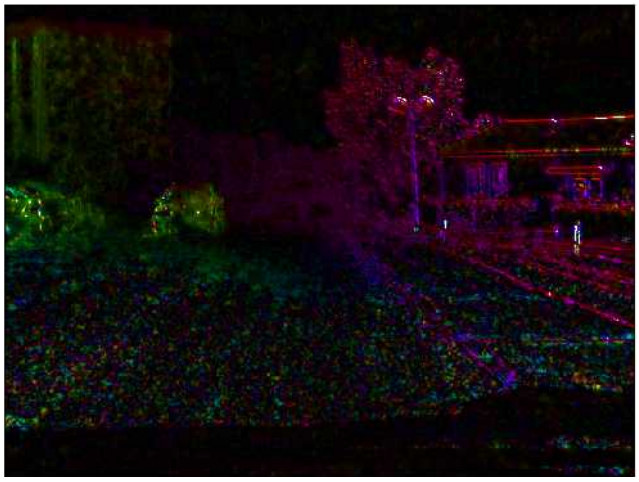
$$\begin{cases} u_{n+1}=\bar{u}_n-\dfrac{I_x(I_x\bar{u}_n+I_y\bar{v}_n+I_t)}{\alpha^2+I_x^2+I_y^2} \\ v_{n+1}=\bar{v}_n-\dfrac{I_y(I_x\bar{u}_n+I_y\bar{v}_n+I_t)}{\alpha^2+I_x^2+I_y^2} \end{cases}$$

avec

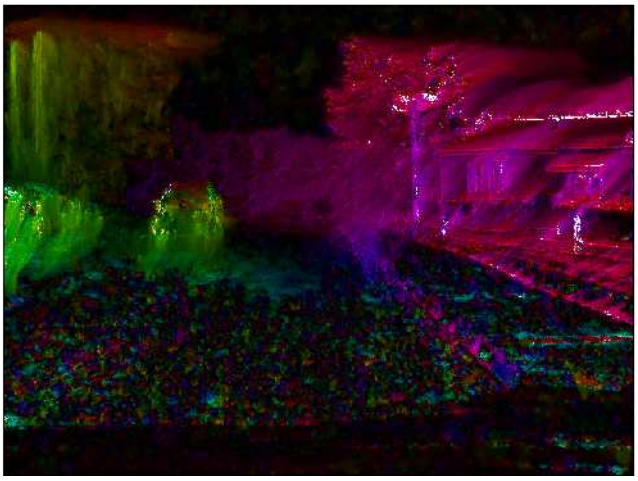
$$\bar{u}_n=\frac{1}{6}(u_{i-1,j}+u_{i,j-1}+u_{i,j+1}+u_{i+1,j})+\frac{1}{12}(u_{i-1,j-1}+u_{i+1,j-1}+u_{i-1,j+1}+u_{i+1,j+1})$$
$$\bar{v}_n=\frac{1}{6}(v_{i-1,j}+v_{i,j-1}+v_{i,j+1}+v_{i+1,j})+\frac{1}{12}(v_{i-1,j-1}+v_{i+1,j-1}+v_{i-1,j+1}+v_{i+1,j+1})$$

α étant le coefficient de régularisation qu'il faut régler, de même que le nombre d'itérations de la méthode de résolution. Pour améliorer l'estimation, il est possible d'utiliser une implémentation pyramidale et d'effectuer des itérations de raffinement au sein d'un même niveau de pyramide.

Les résultats, avec l'utilisation de la carte de couleurs définie précédemment sont :



HS simple (α = 5, 30 itérations)



HS raffiné (α= 5, 30 itér. avec raffinement)

HS Pyramidal (3 niveaux, α = 13, 100 itérations)

Figure 10. Résultats Horn & Schunck

Les résultats fournis par Horn & Schunck non pyramidal sont très sensibles au bruit (sur la route notamment) et on ne dispose finalement d'information que sur les gradients spatiaux. Le caractère global de la méthode lui confère un caractère très lisse, ce qui est encore plus exacerbé par l'utilisation de plusieurs niveaux de résolution (pyramide). Les résultats sont imprécis et difficilement exploitables, en effet les petits mouvements sont ignorés du fait de la globalité de la méthode et le lissage est très important, avec finalement peu d'information utile : le véhicule à détecter est confondu avec le mouvement radial de la route, la segmentation a posteriori risque donc d'échouer.

(c) Méthode de Lucas & Kanade

On fait toujours l'hypothèse de conservation de la luminance entre deux images successives, à laquelle on ajoute une hypothèse (forte mais locale) de constance du flot sur un voisinage Ω centré sur le pixel dont on veut calculer le déplacement.

On est donc conduit à minimiser la fonctionnelle suivante : $J_{LK} = \sum_{\Omega} [\nabla I . \vec{\omega} + I_t]^2$.

La solution la plus simple (maximum de vraisemblance) consiste à réaliser une estimation au sens des moindres carrés, c'est à dire à résoudre de la façon suivante sur le voisinage Ω :

$$Soient\ A=\begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \dots & \dots \\ I_{xn} & I_{yn} \end{bmatrix}, b=\begin{bmatrix} I_{t1} \\ I_{t2} \\ \dots \\ I_{tn} \end{bmatrix}, alors \begin{bmatrix} u \\ v \end{bmatrix}=(A^T A)^{-1} A^T b \quad ,$$

ce qui est également équivalent à : $G=\begin{bmatrix} \sum_{\Omega} I_x^2 & \sum_{\Omega} I_x I_y \\ \sum_{\Omega} I_x I_y & \sum_{\Omega} I_y^2 \end{bmatrix}, \eta=\begin{bmatrix} \sum_{\Omega} I_t I_x \\ \sum_{\Omega} I_t I_y \end{bmatrix} alors \begin{bmatrix} u \\ v \end{bmatrix}=G^{-1}\eta$.

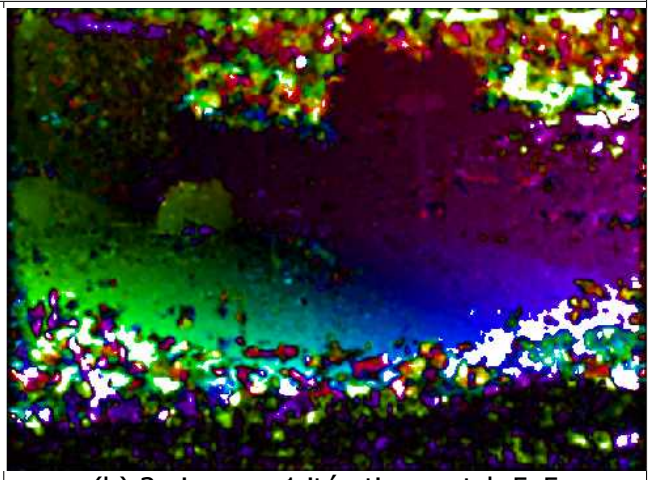
Il peut arriver que la matrice A^TA (ou G, de manière équivalente) soit mal conditionnée (déterminant quasi nul, donc non inversible). Cela provient du fait que le pixel considéré se trouve dans une zone où l'intensité lumineuse est constante sur le domaine, d'où un gradient nul (problème de l'ouverture). Le calcul par les moindres carrés risque alors de fournir une estimation aberrante. Par ailleurs, l'estimateur des moindres carrés a la particularité d'être très sensible au bruit et aux erreurs de mesure.

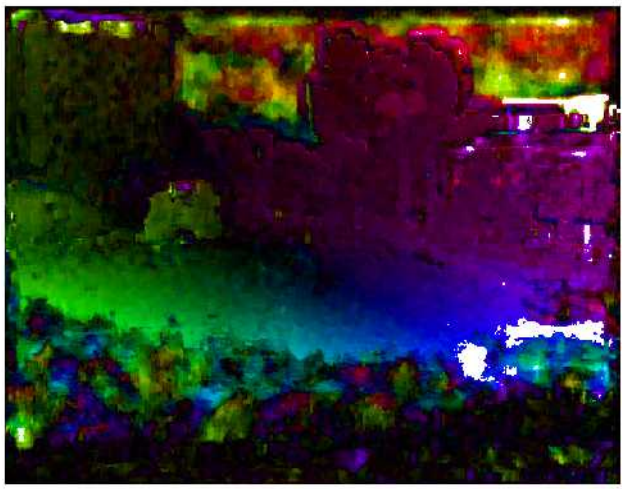
On peut remédier à ces problèmes en utilisant une technique de régularisation, comme la régularisation quadratique l_2l_2. La nouvelle fonctionnelle s'écrit : $J_{LKL2}=\sum_{\Omega}[\nabla I.\vec{\omega}+I_t]^2+\alpha|\omega|^2$

Cet estimateur est toujours linéaire, et s'exprime tout calcul fait de la manière suivante : $\begin{bmatrix} u \\ v \end{bmatrix}=(A^T A+\alpha I)^{-1} A^T b$, avec α réglable, représentant la régularité de la solution [Richard].

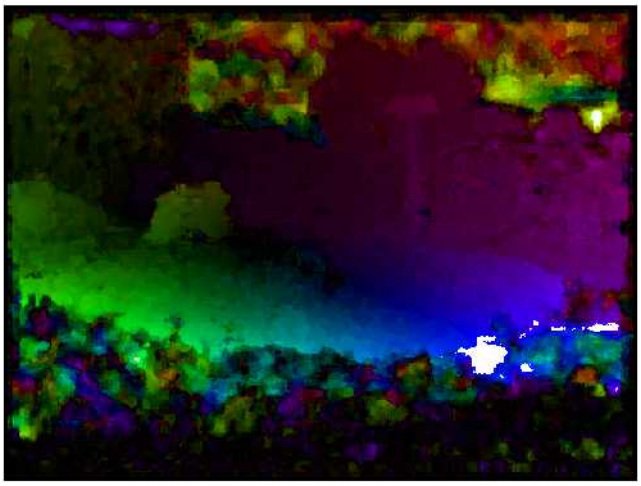
De même que pour la méthode d'Horn & Schunck, il est possible de réaliser une implémentation pyramidale [12] et d'utiliser le raffinement itératif à chaque niveau. Les paramètres à régler sont alors la taille du patch, le nombre de niveaux de pyramide et le nombre d'itérations.

Les résultats fournis par cette méthode sont détaillés dans le tableau ci-après.

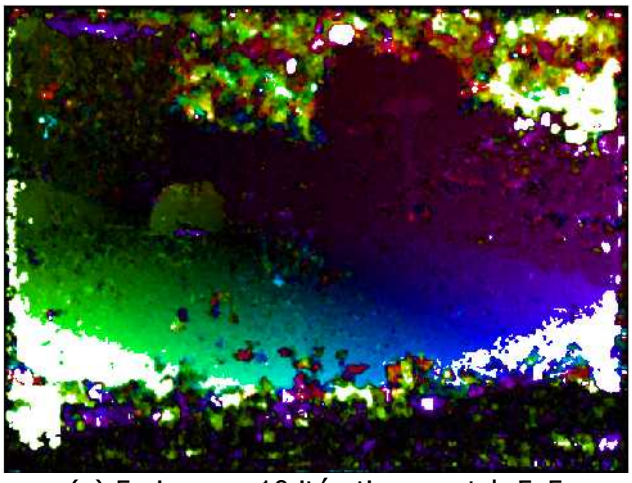
Résultats Lucas & Kanade	
	
(a) 1 niveau, 1 itération, patch 9x9	(b) 3 niveaux, 1 itération, patch 5x5



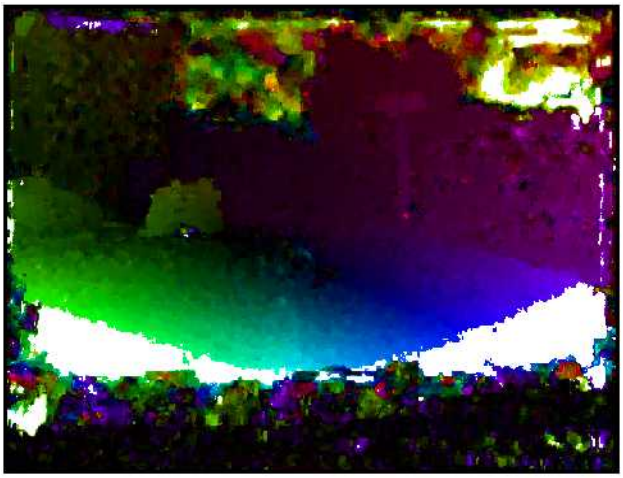
(c) 3 niveaux, 1 itération, patch 9x9



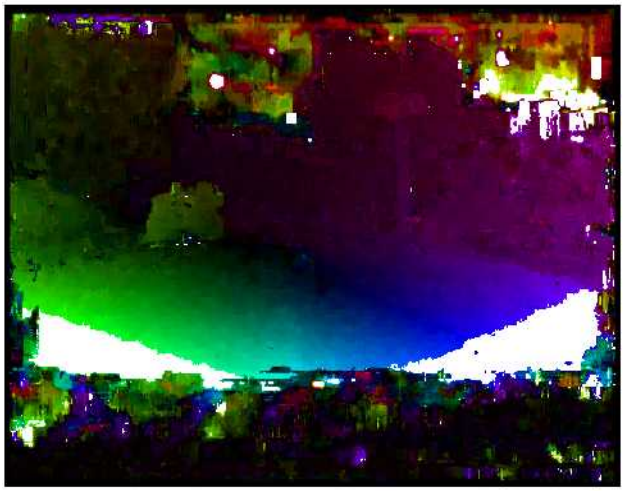
(d) 3 niveaux, 1 itération, patch 9x9, régularisé



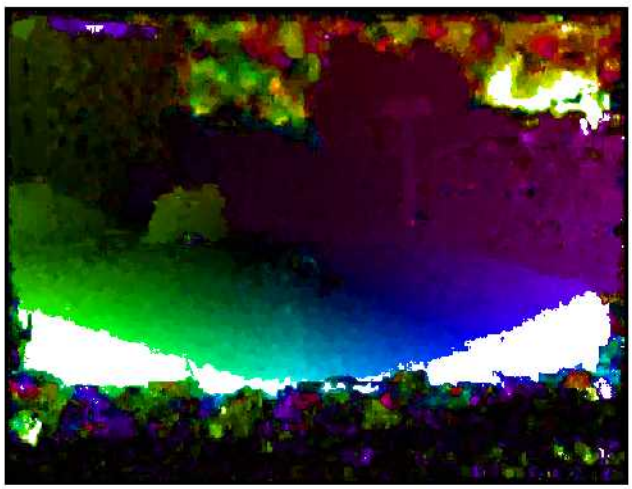
(e) 5 niveaux, 10 itérations, patch 5x5



(f) 5 niveaux, 10 itérations, patch 9x9



(g) 5 niveaux, 10 itérations, patch 11x11



(h) 5 niveaux, 10 itérations, patch 9x9, régularisé

Figure 11. Résultats Lucas & Kanade

Le meilleur résultat obtenu est celui pour 5 niveaux de pyramide, 10 itérations, une taille de patch de 9x9 et un calcul par moindres carrés régularisés (figure 11 h). Sur ces images réelles, le patch 5x5 se révèle trop sensible au bruit et donc une hypothèse de régularité trop faible.

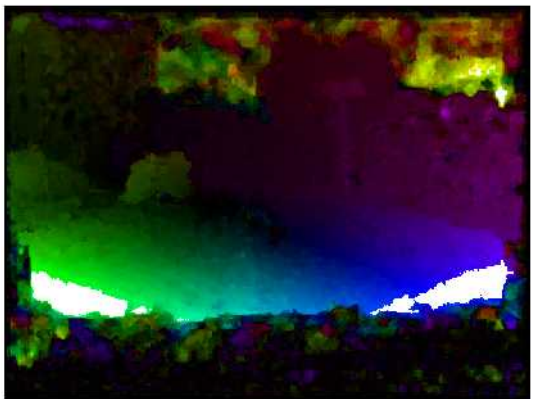
De même, trois niveaux de pyramides sont insuffisants pour repérer les mouvements radiaux importants. Le patch 11x11 rajoute des calculs supplémentaires tout en n'améliorant pas fondamentalement le résultat et en lissant le champ.

Quelle que soit la méthode employée, les bords ainsi que le bas de l'image (« essuie-glace ») comporteront toujours des valeurs aberrantes car il s'agit de points de l'image qui disparaissent au cours du mouvements, il est donc impossible « d' inventer » leur déplacement.

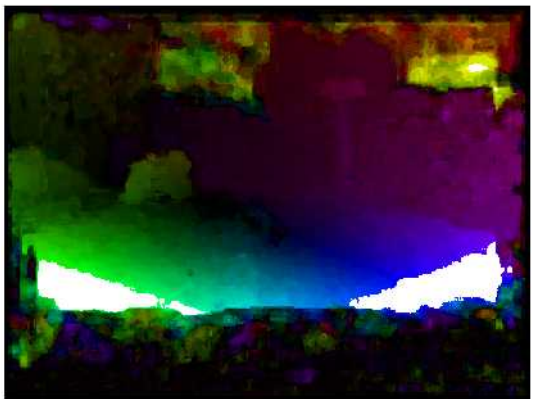
Par ailleurs, le ciel, qui est uni, perturbe considérablement tous les algorithmes. Toutefois, on constate lorsque l'on effectue la reconstruction de l'image (déplacement des pixels par la vitesse calculée point à point) que l'image déplacée est bonne, même dans le ciel, ce qui signifie que l'algorithme a bien retrouvé le pixel correspondant, mais les mouvements sont erronés.

L'important est toutefois le respect des contours des objets. On retrouve bien un flot majoritairement radial (avancée du véhicule en ligne droite), et le véhicule arrivant en sens inverse (objet à détecter principalement) est bien segmenté. Enfin, le flot est dense et le plus exact possible.

Remarque : implémentation non itérative (= une seule itération) :



5 niveaux, une itération, 9x9



5 niveaux, une itération, 11x11

Figure 12. Résultats Lucas Kanade Pyramidal non itératif

Les résultats obtenus avec l'implémentation pyramidale non itérative sont très satisfaisants, bien que d'un grain moins fin. L'étude de l'influence du paramètre itération sera étudié sur la séquence de référence ultérieurement.

On remarque surtout que les résultats obtenus sont analogues à ceux fournis par OpenCV, mais en s'affranchissant des erreurs (traits horizontaux et verticaux) engendré par cette méthode, ce qui place donc la méthode utilisée comme un très sérieux candidat pour l'algorithme final.

(d) Méthode de Lucas & Kanade : Prise en compte de la couleur

Les caméras récentes fournissant des images couleur, on est en droit de s'interroger sur la manière d'utiliser ces informations pour le calcul du flot optique.

Soient R, G et B les valeurs des intensités correspondant aux composantes rouge, verte et bleue

de nos images. Alors on peut écrire l'équation du flot optique [17] comme suit :

$$\begin{cases} \frac{\partial R}{\partial x} v_x + \frac{\partial R}{\partial y} v_y + \frac{\partial R}{\partial t} = 0 \\ \frac{\partial G}{\partial x} v_x + \frac{\partial G}{\partial y} v_y + \frac{\partial G}{\partial t} = 0 \\ \frac{\partial B}{\partial x} v_x + \frac{\partial B}{\partial y} v_y + \frac{\partial B}{\partial t} = 0 \end{cases}$$

L'avantage de ce système est qu'il est bien posé : on a plus d'équations que d'inconnues, on peut donc résoudre simplement par une technique de type moindres carrés ou pseudo inverse. Une variante pouvant accélérer l'algorithme et conduire à un système carré (résolution exacte) est d'effectuer la moyenne de deux plans (R et G par exemple), et de considérer les deux équations restantes.

Les invariants photométriques :

Soit la couleur c = [R,G,B]. Il existe plusieurs façons de concevoir des invariants photométriques[18] :

- normalisation : moyenne, racine cubique des trois informations (équivalent à l'intensité lumineuse en niveau de gris)
- Dérivée du logarithme des intensités respectives : invariance par rapport à l'intensité.
- Transformées sphériques et coniques :
  - HSV (Teinte, Saturation, Valeur):

$$\begin{cases} H = \begin{cases} \frac{G-B}{M-m}.60° \; si \; R \geq G, B \\ (2 + \frac{B-R}{M-m}).60° \; si \; G \geq R, B \\ (4 + \frac{R-G}{M-m}).60° \; si \; B \geq R, G \end{cases} \\ S = \frac{M-m}{M} \\ V = M \end{cases}$$

avec M=max(R,G,B) et m=min(R,G,B)

  - r, θ, φ : $\begin{cases} r = \sqrt{R^2 + G^2 + B^2} \\ \theta = \arctan\left(\frac{G}{R}\right) \\ \phi = \arcsin\left(\frac{\sqrt{R^2 + G^2}}{\sqrt{R^2 + G^2 + B^2}}\right) \end{cases}$

On intègre ainsi la couleur dans le calcul du flot optique par Lucas & Kanade : on fait l'hypothèse de conservation de la luminance sur chaque plan couleur, et on a alors le système surdéterminé suivant : $\begin{cases} \frac{\partial R}{\partial x}v_x+\frac{\partial R}{\partial y}v_y+\frac{\partial R}{\partial t}=0 \\ \frac{\partial G}{\partial x}v_x+\frac{\partial G}{\partial y}v_y+\frac{\partial G}{\partial t}=0 \\ \frac{\partial B}{\partial x}v_x+\frac{\partial B}{\partial y}v_y+\frac{\partial B}{\partial t}=0 \end{cases}$, soit les matrices $A=\begin{bmatrix} \frac{\partial R}{\partial x} & \frac{\partial R}{\partial y} \\ \frac{\partial G}{\partial x} & \frac{\partial G}{\partial y} \\ \frac{\partial B}{\partial x} & \frac{\partial B}{\partial y} \end{bmatrix}, b=-\begin{bmatrix} \frac{\partial R}{\partial t} \\ \frac{\partial G}{\partial t} \\ \frac{\partial B}{\partial t} \end{bmatrix}$

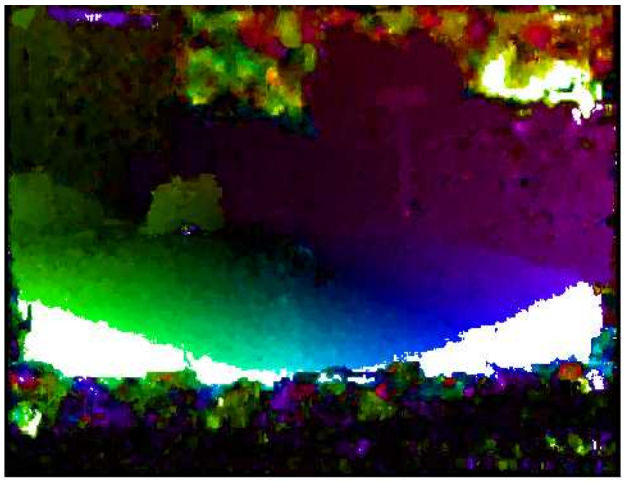
alors on obtient : $\omega=(A^T A)^{-1} A^T b$

Examinons les résultats obtenus avec comme paramètres : 5 niveaux, patch 9x9, 10 itérations.

(R, θ, φ)

RGB

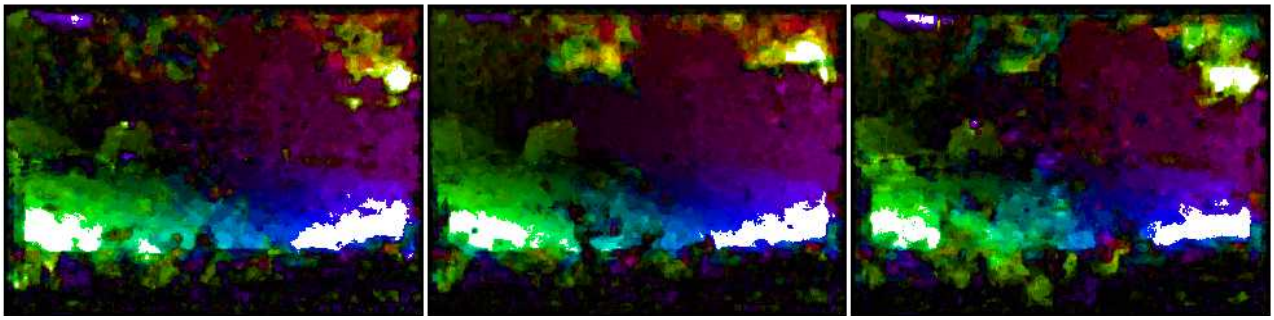
H,S,I



Référence : Niveaux de Gris

Figure 13. Résultats Couleur

Les résultats obtenus sont très peu satisfaisants, avec notamment l'apparition de nombreux points aberrants. Il est possible de mettre en évidence ces problèmes sur l'espace RGB par exemple en examinant le résultat du calcul sur chaque composante (figure 14).



Rouge Vert Bleu

Figure 14. Calcul du flot optique sur les trois composantes distinctes

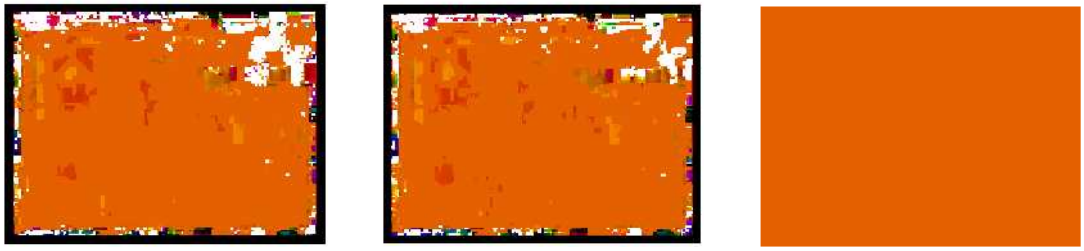
Chaque plan est perturbé (bruit, artefacts, espaces unis) à certains endroits, de manière différente car les canaux ne sont pas sensibles aux mêmes variations, ce qui fait que l'estimation « au mieux » (moindres carrés) utilisant les trois plans est finalement très perturbée.

Remarque : exemple d'images de synthèse

Pour la séquence de synthèse suivante (couleur puis passée en noir et blanc) :

(déplacement constant de 4 pixels en hauteur et 2 pixels en largeur)

Les résultats sont les suivants (dans les deux cas, 3 niveaux, patch 9x9 et 10 itérations :



Flot Noir et Blanc Flot Couleur Flot réel

Les deux techniques donnent un résultat cohérent (globalement le vrai mouvement est retrouvé). Localement, la technique couleur donne moins de résultats aberrants que son homologue noir et blanc, et ce dans une image où le contraste est faible. Cela donne une indication sur la sensibilité des méthodes couleurs au bruit. En synthèse (application à l'estimation de mouvement dans des vidéos de synthèse par exemple) où l'on s'affranchit du bruit, les résultats sont convaincants.

Commentaires sur les méthodes de type Lucas & Kanade, sur les espaces de couleur :

Barron & Klette [17] concluent à la fin de l'article « Quantitative Optical Flow' » que les méthodes couleur « semblent » meilleures. L'utilisation de la saturation (H de HSV) considérée a priori comme devant fournir les meilleurs résultats donne en fait des résultats catastrophiques (du fait de la sensibilité exacerbée aux variations d'illumination).

Cet article est représentatif des travaux sur les méthodes utilisant divers espaces de couleurs : les résultats sont donnés sur des images synthétiques, avec le plus souvent l'utilisation de modèles paramétriques.

Weickert et al. [18] utilisent des images réelles pour tester leur méthode, les résultats sont convaincants mais le principal inconvénient de leur code est un lissage excessif du résultat : certes, l'objet en mouvement est bien détecté, mais les mouvements dans le reste de l'image sont lissés au maximum, donnant finalement un flot globalement erroné.

Les explications que l'on peut avancer concernant l'échec des méthodes couleurs sont les suivants :

- Si l'un des trois plans couleur risque de fournir des valeurs aberrantes en un point donné, le résultat sera biaisé par cette valeur, par le fait même des moindres carrés, comme on peut le voir sur l'exemple simple suivant (figure 15):

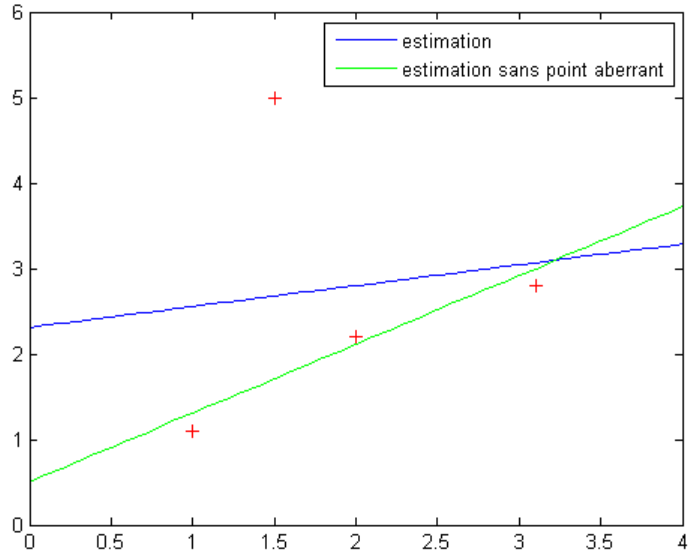


Figure 15. Influence de points aberrants sur les moindres carrés

- Le fait de moyenner (passage de RGB à niveaux de gris, quelque soient les coefficients retenus) le signal obtenu par le capteur couleur nous affranchit finalement du bruit propre à chaque canal de couleur. Cela revient finalement à utiliser l'information des trois canaux pour approcher au mieux la valeur vraie de l'intensité lumineuse, on est ainsi raisonnablement beaucoup moins sensible au bruit qu'en utilisant un capteur CCD noir et blanc classique.

- L'utilisation des moindres carrés comme on l'a fait présuppose l'indépendance des trois canaux couleurs, apportant des informations supplémentaires les uns par rapport aux autres. On peut alors expliquer les erreurs rencontrées par le fait que les trois plans couleurs ne sont en réalité pas indépendants (mesure de la même scène au même instant, avec un même capteur).

3. Block matching

Les techniques de Block Matching sont couramment utilisées dans les algorithmes de compression d'image, pour l'estimation du mouvement et donc le codage relatif qui en découle (norme MPEG notamment).

Le principe est le suivant : on considère la vitesse constante par blocs et on autorise un domaine de recherche autour du bloc considéré (figure 16). Dans ce domaine, on cherche le meilleur score de corrélation entre le patch initial et un autre patch. La différence des coordonnées entre le patch initial et le meilleur candidat nous donne alors le déplacement.

Les critères de corrélation les plus fréquemment employés, car les plus simples à implémenter, sont $SSD = \sum \sum (I(i,j) - J(i+u, j+v))^2$ et $SAD = \sum \sum |I(i,j) - J(i+u, j+v)|$.

Un des principaux inconvénients des techniques de Block Matching est le choix des paramètres, notamment celui du domaine de recherche. En effet, celui-ci nous imposera la vitesse maximale recherchée, alors que dans le cas d'une méthode variationnelle la « valeur vraie » est conditionnée simplement par les gradients. Le champ obtenu est borné par le choix de mb et p.

Par ailleurs, le Block Matching donne des déplacements entiers tandis que l'estimateur des moindres carrés nous donne par conception des valeurs subpixeliques. Pour obtenir des déplacements subpixeliques à l'aide de techniques de type Block Matching, il nous faut sur-échantillonner l'image puis remettre les estimées à la taille originale par le biais d'une interpolation.

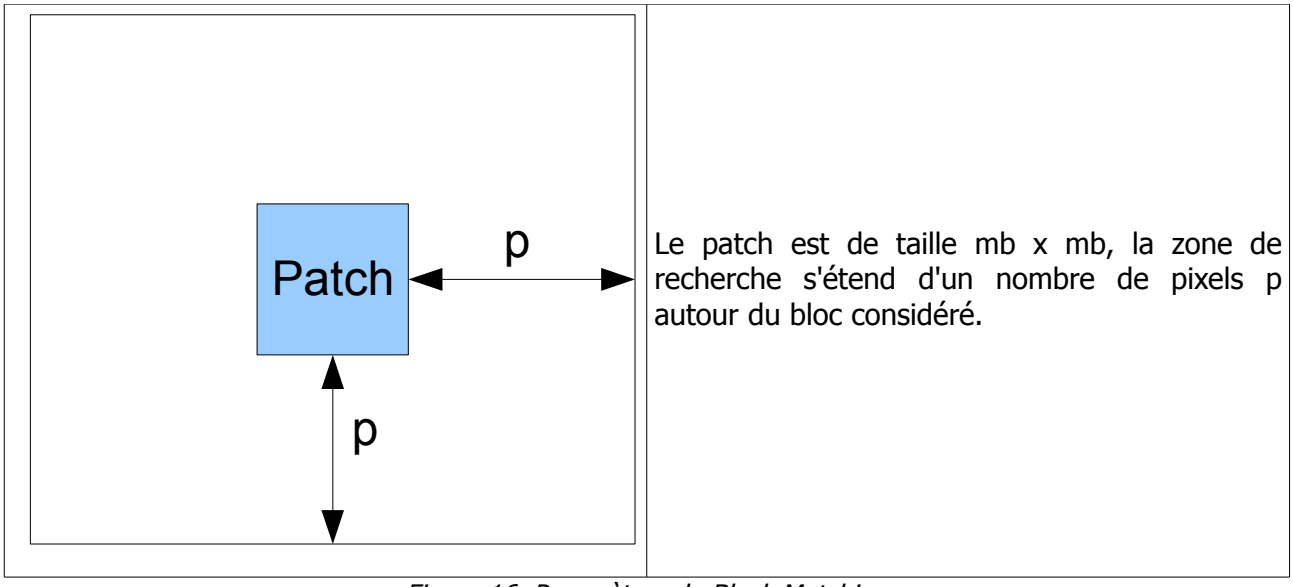


Figure 16. Paramètres du Block Matching

Les algorithmes sont conçus à l'origine pour fournir une estimée par patch, à savoir, tous les pixels du patch considéré se verront attribuer le même déplacement. On peut toutefois, de manière analogue à ce que l'on fait en Lucas & Kanade, attribuer une vitesse différente à chaque pixel : il suffit d'effectuer le même calcul, mais en chaque pixel plutôt que toutes les tailles de bloc. Cela revient à ajouter au Block Matching une contrainte de continuité, qui nous renvoie un résultat (certes toujours entier, non subpixelique, cela est corrigé par l'exécution du calcul sur l'image sur-échantillonnée) plus proche de Lucas & Kanade.

Il existe de nombreux algorithmes d'exploration de la zone de mouvement autorisée autour du patch considéré. L'explication des techniques les plus couramment utilisée est donnée ci-après :

- *Full Search* ou *Exhaustive Search* : le plus simple que l'on puisse concevoir, la fonction de coût est calculée pour chaque pixel. Malgré le coût de calcul important qu'il implique, cet algorithme est celui qui fournit les meilleures estimées.

- ***Three Step Search*** **(TSS)** [25] **:**

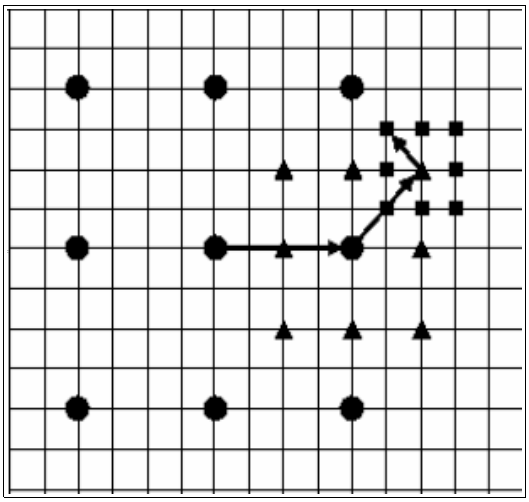


Figure 17. Three Step Search

La recherche, comme le nom de la méthode l'indique, se fait en trois étapes :

- Lors de la première étape, on compare le pixel central aux pixels situés à la distance S du centre (ronds)
- On effectue la même recherche autour du meilleur candidat retenu lors de la première étape, avec une distance S/2
- De même lors de la troisème étape autour du meilleur candidat de la seconde étape avec la distance S/4.

- ***Diamond Search*** **(DS**) [30]:

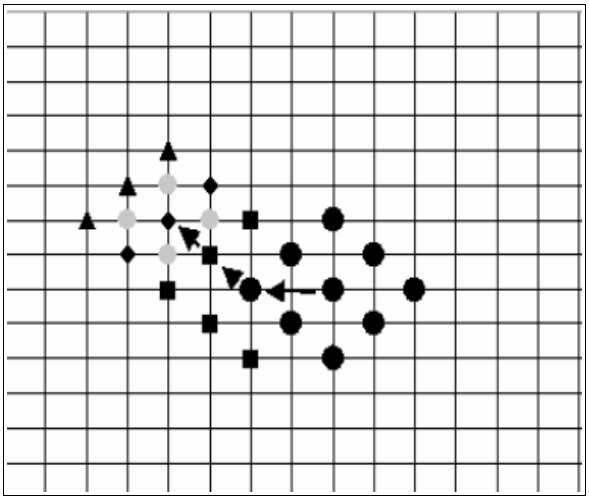


Figure 18. Diamond Search

Le motif de recherche est cette fois-ci un losange, et le pas de recherche est constant. De la même manière qu'avec le TSS, on considère le meilleur candidat sur le tour du losange, puis on recommence le processus jusqu'à converger vers un minimum. Le nombre d'itérations n'est pas fixé, le critère d'arrêt étant la convergence de la recherche, d'où une délicate estimation des performances car l'exécution de l'algorithme dépendra des caractéristiques de la séquence d'image utilisée.

- ***Adaptative Rood Pattern Search*** (ARPS) [26] :

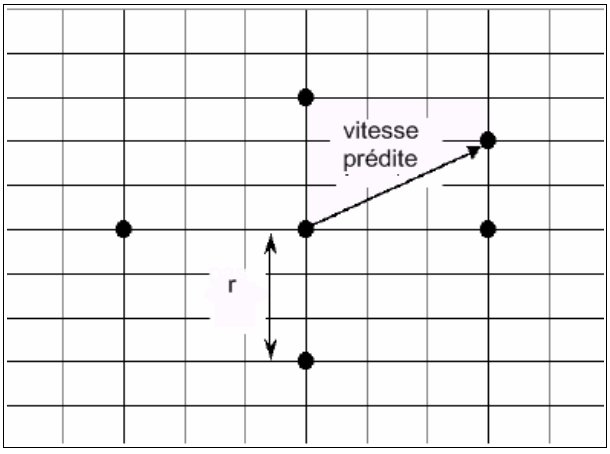
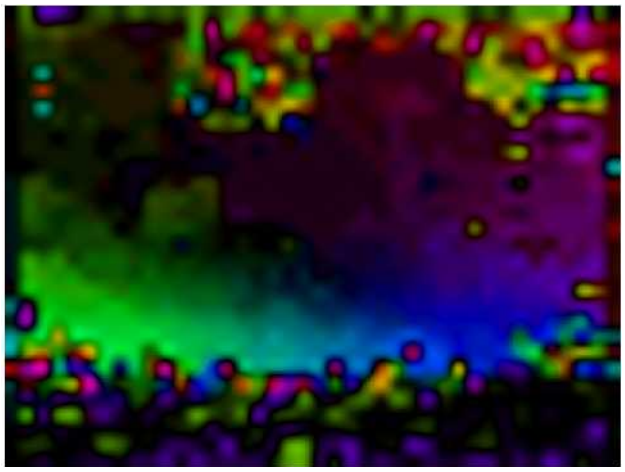


Figure 19. ARPS

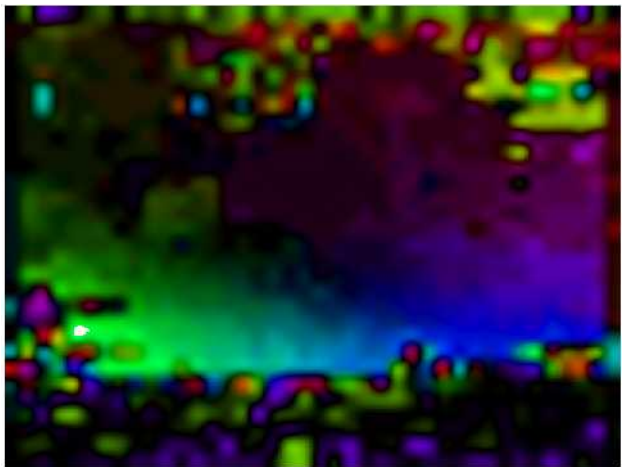
Cet algorithme consiste à utiliser la vitesse prédite par les voisins du pixel considéré comme *a priori* de la vitesse à estimer. On effectue la recherche sur le bloc où pointe la vitesse prédite d'une part, au centre lui-même et aux quatre coins de la croix représentée ci-contre d'autre part.

Cet algorithme est celui dont le rapport signal sur bruit se rapproche le plus de la recherche exhaustive (Full Search). On l'utilisera pour tester de manière rapide les implémentations mises en oeuvre.

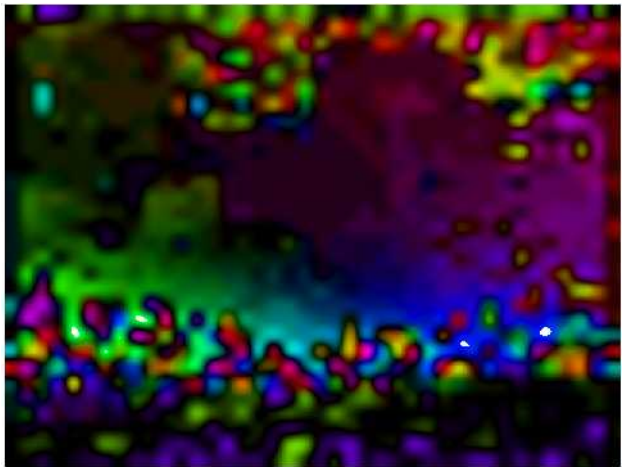
Le tableau suivant contient les principaux résultats obtenus avec les différentes techniques de Block Matching :



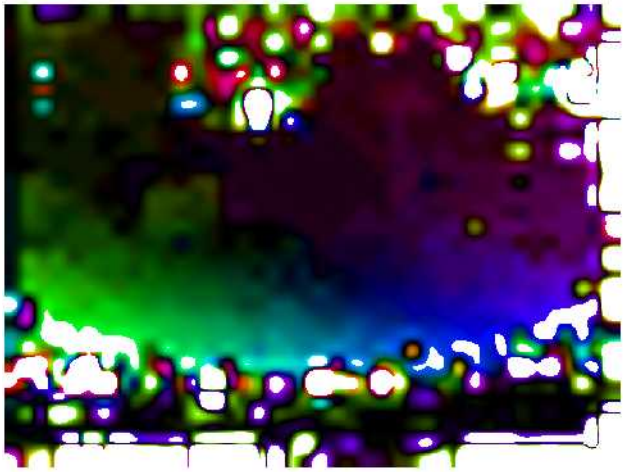
(a) Full Search, mb=16,p=7



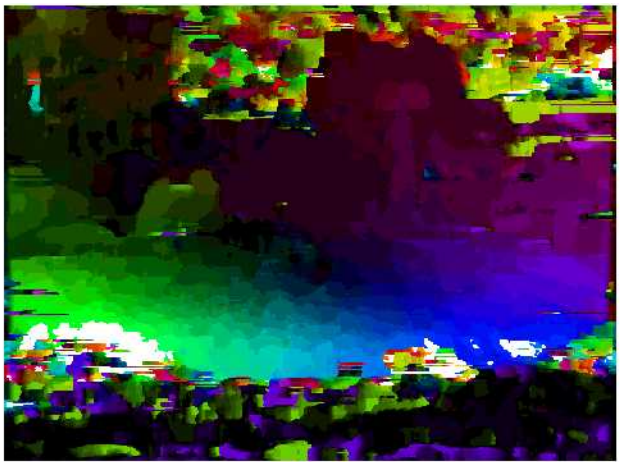
(b) ARPS, mb=16,p=7



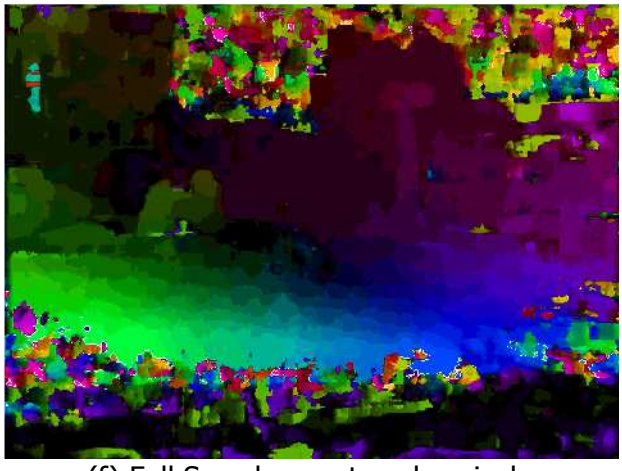
(c) DS, mb=16, p=7



(d) Full Search, raffinement 100 iterations



(e) ARPS pour tous les pixels
mb=16, p=7



(f) Full Search pour tous les pixels
mb=16, p=7

Figure 20. Calcul du flot optique par techniques de Block Matching

La dernière estimation (Figure 20 f) fournit un résultat cohérent analogue à celui fourni par OpenCV et Lucas & Kanade. Il est toutefois assez difficile de régler les paramètres et la sensibilité au bruit est assez forte (il est difficile de réduire la taille du bloc). Par ailleurs, le résultat est pixelique, problème dont on peut s'affranchir en suréchantillonnant l'image, toutefois le temps de calcul est considérablement augmenté (quatre fois plus de calcul à chaque niveau supplémentaire de pyramide...). Ce que l'on peut conclure est la grande utilité du block matching pour estimer rapidement des grands mouvements, mais une plus grande difficulté à effectuer un calcul précis subpixelique avec l'identification de tous les mouvements des objets de l'image. Ce type d'estimation reste toutefois un bon point de comparaison pour l'algorithme que nous allons définir.

4. Equivalence Block Matching / Méthodes Variationnelles

Il est intéressant d'établir un parallèle entre les méthodes de Block Matching et les méthodes variationnelles qui ont été décrites et testées précédemment. En effet, si la manière d'effectuer l'estimation est différente, la « philosophie » des deux approches est sensiblement la même.

(cf CQ Davis, ZZ Karu & DM Freeman *Equivalence of Subpixel Motion Est. Based on OF & BM*) [51]

On peut résumer le calcul du flot optique à l'aide de méthodes de type moindres carrés sous la forme suivante : $[u,v]=\arg\min_{u,v}\sum_{i,j}[I_x u+I_y v+I_t]^2$.

De la même manière, les algorithmes de Block Matching, fondés sur l'hypothèse que, sur un même voisinage dans deux images successives I(x,y,1)=I(x-u,y-v,0), reviennent à chercher :

$$[u,v]=\arg\min_{u,v}\sum_{i,j}\left|I(i+u,j+v,1)-I(i,j,0)^q\right|$$ avec q = 1 (SAD), 2 (SSD) etc...

Dans le cadre d'une estimation subpixelique , on calcule le patch déplacé à l'aide d'une interpolation bilinéaire, de la forme suivante :

$$I(i+u,\ j+v)=(1-u)(1-v)I(i,\ j)+u(1-v)I(i+1,j)+(1-u)vI(i,j+1)+uvI(i+1,j+1)$$

avec $0\le u<1\ et\ 0\le v<1$ (déplacements subpixeliques)

Montrons sur un exemple simple, une ligne 1D, que le block matching utilisant une fonction de coût SAD revient à utiliser l'équation du flot optique :

Soit $u_{BM}=\arg\min_{u}\sum_{i}[I(i+u,1)-I(i,0)]^2$.

L'interpolation bilinéaire donne : $I(i+u)\simeq(1-u)I(i)+uI(i+1)$,

on remplace donc dans la somme : $u_{BM}=\arg\min_{u}\sum_{i}[(1-u)I(i,1)+uI(i+1,1)-I(i,0)]^2$,

puis en développant : $u_{BM}=\arg\min_{u}\sum_{i}[I(i,1)-I(i,0)+uI(i+1,1)-uI(i,1)]^2$.

Or, l'approximation la plus simple de l'opérateur dérivée est la simple différence, on peut donc réécrire le dernier résultat obtenu comme : $u_{BM}=\arg\min_{u}\sum_{i}[I_t+uI_x]^2$, ce qui équivaut à l'équation du flot optique en dimension 1.

On peut effectuer des calculs identiques en dimension 2 :

$[u,v]_{BM} = \arg\min_{u,v} \sum_{i,j} [I(i+u, j+v, 1) - I(i,j,0)]^2$. En remplaçant avec l'interpolation bilinéaire, on a le résultat suivant :

$$[u,v]_{BM} = \arg\min_{u,v} \sum_{i,j} [(1-u)(1-v)I(i,j,1) + u(1-v)I(i+1,j,1) + (1-u)vI(i,j+1,1) + uvI(i+1,j+1,1) - I(i,j,0)]^2$$

que l'on peut ordonner de la manière suivante :

$$(1-u)(1-v)I(i,j,1) + u(1-v)I(i+1,j,1) + (1-u)vI(i,j+1,1) + uvI(i+1,j+1,1) - I(i,j,0)$$

$$= I(i,j,1) - I(i,j,0) + u[I(i+1,j,1) - I(i,j,1)] + v[I(i,j+1,1) - I(i,j,1)]$$
$$+ uv[I(i,j,1) - I(i+1,j,1) + I(i+1,j+1,1) - I(i,j+1,1)]$$

On reconnaît en première ligne l'équation du flot optique $I_t + I_x u + I_y v$. Le terme de la deuxième ligne correspond à une « dérivation croisée », à l'aide d'un masque de la forme : [1 -1;-1 1], cela ne correspond à rien de physique et n'apparaît pas dans les méthodes classiques de calcul du flot optique.

On peut toutefois remarquer que l'on extrait ce terme lorsque les mouvements u et v sont petits (inférieurs à un pixel), le produit u.v est donc lui très petit, et l'on peut raisonnablement négliger ce terme devant les précédents, et on a donc finalement, avec une *interpolation bilinéaire* en *subpixelique* : $\sum_{i,j} [I(i+u, j+v, 1) - I(i,j,0)]^2 \simeq \sum_{i,j} [I_x u + I_y v + I_t]^2$.

On a donc montré que l'équivalence est vraie pour les déplacements subpixeliques. Or tout mouvement peut être décomposé en une série de déplacements subpixeliques, d'où par extension le théorème :

Tout algorithme de Block-Matching 2D utilisant une interpolation bilinéaire et une fonction de coût SSD peut être remplacé de manière équivalente par un algorithme variationnel utilisant les dérivées du premier ordre, et réciproquement. (Davis, Karu, Freeman, 1995).

L'équivalence de ces deux classes d'algorithmes vient du fait que l'interpolation bilinéaire utilise finalement la différence centrée, qui peut également être utilisée comme approximation de la dérivée.
L'utilisation de fonctions de coût différentes ou d'autres masques de convolutions entraînent des équivalences différentes (des fonctions de coût utilisant une puissance plus élevée seront équivalents à des algorithmes variationnels utilisant les dérivées d'ordre supérieur pour la régularisation).

5. Algorithme choisi et optimisations

(a) Description de l'implémentation

Compte tenu des résultats obtenus et du cahier des charges, le choix final de l'algorithme de calcul du flot optique est une implémentation pyramidale de la méthode de Lucas & Kanade, avec l'introduction des moindres carrés régularisés. Le schéma suivant (figure 21) donne une description précise de l'implémentation de cet algorithme.

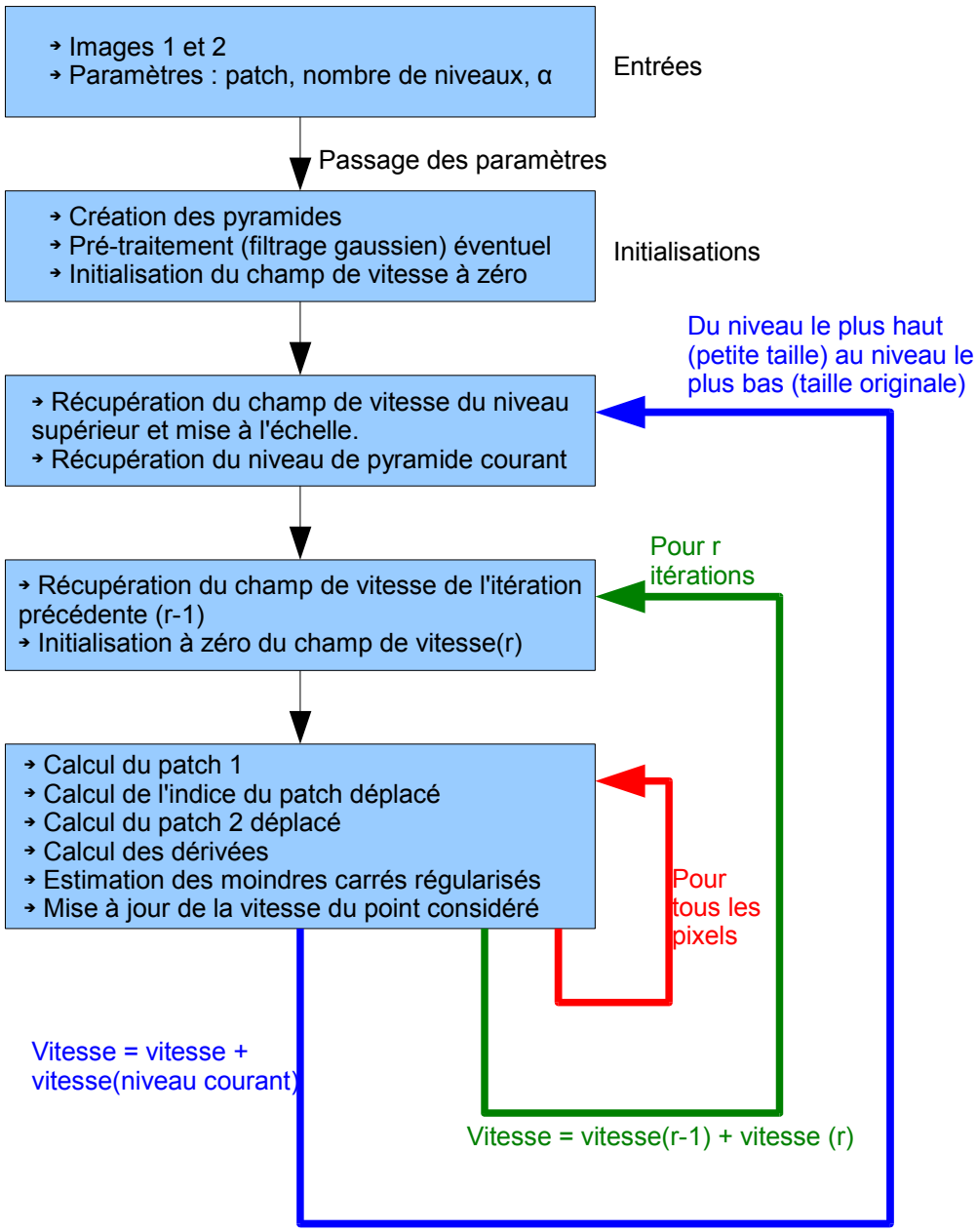


Figure 21. Description de l'implémentation de l'algorithme utilisé

A partir de deux images successives, il s'agit tout d'abord de créer les pyramides gaussiennes (sous échantillonnage) pour chaque image. L'algorithme de calcul en lui même commence alors avec l'interpolation du champ de vitesse du niveau supérieur, puis le calcul pour chaque pixel.

(b) Optimisations

Il est possible d'optimiser le code décrit précédemment en remarquant qu'il est équivalent de calculer les dérivées sur chaque patch et de calculer les dérivées puis de considérer le patch. Cette dernière version est même conseillée car l'on s'affranchit ainsi des effets de bord et des calculs redondants sont ainsi évités.

Par ailleurs, le calcul de l'indice déplacé est pour le moment entier, ce qui est inexact car cela revient à arrondir le champ calculé au niveau précédent et donc à perdre le subpixelique. Cela peut être résolu en effectuant l'interpolation des images dérivées de la manière décrite ci-après.

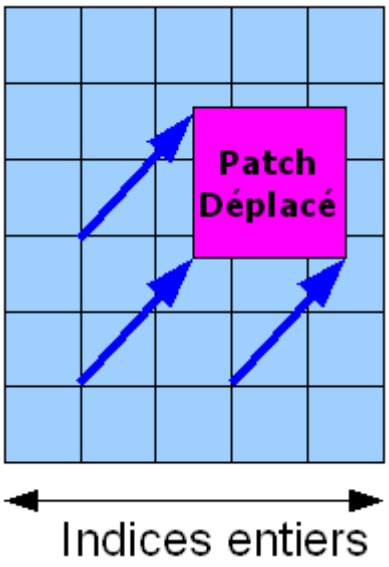


Figure 22. Interpolation

Soient ux et uy les vitesses réelles , soient ux_0 et uy_0 leurs parties entières et αx, αy les parties décimales telles que :

$ux = ux_0 + \alpha x$; $uy = uy_0 + \alpha y$;

Alors le point interpolé I(ux,uy) a pour valeur :

$$I(ux,uy) = (1-\alpha x).(1-\alpha y) \cdot I(ux_0, uy_0) + \alpha x.(1-\alpha y) \cdot I(ux_0+1, uy_0) + (1-\alpha x).\alpha y \cdot I(ux_0, uy_0+1) + \alpha x.\alpha y \cdot I(ux_0+1, uy_0+1) ;$$

De la même manière, interpoler les dérivées et dériver l'interpolation revient au même. On modifie donc l'algorithme en calculant les images dérivées en même temps que l'on calcule l'interpolation du champ de vitesse précédent. Dans le calcul pour chaque pixel, on effectue simplement l'interpolation du patch de la deuxième image à l'aide des indices déplacés.

Le meilleur résultat que l'on obtient est le suivant : (4 niveaux, patch 10x10, 3 itérations) :

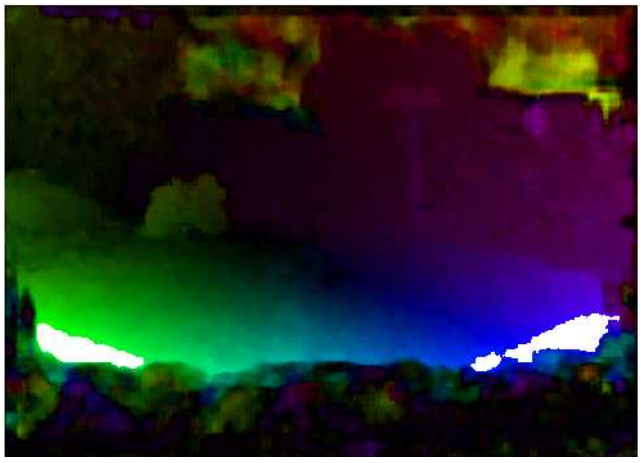
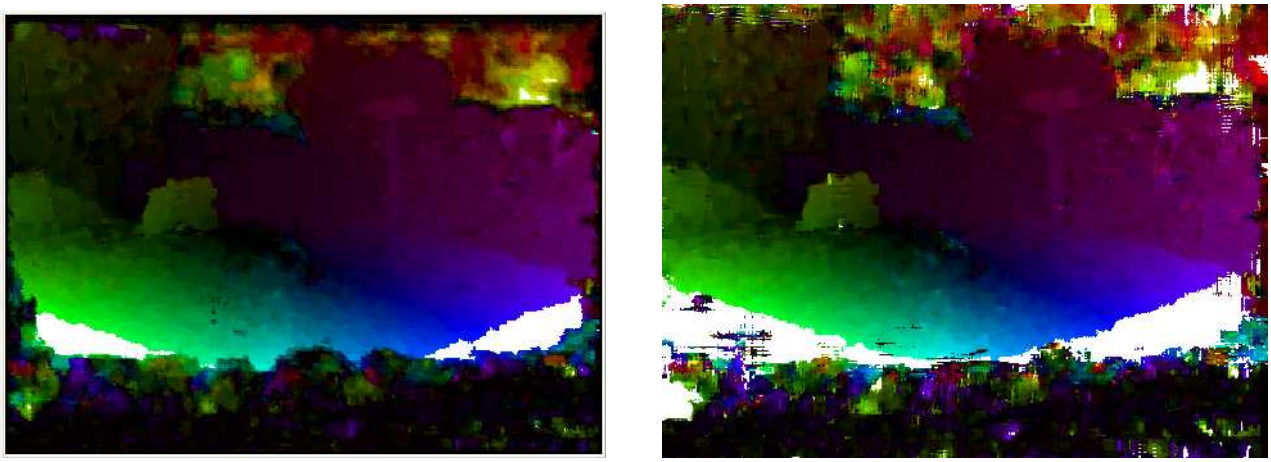


Figure 23. Résultat du calcul du flot optique

En ce qui concerne le temps d'exécution de l'algorithme, l'implémentation Matlab est très lente (ce qui était attendu), il faut ainsi (avec un pentium D) 200 secondes (résolution 640x480) pour effectuer le calcul sur 4 niveaux avec un patch 10x10 et 3 itérations.

Le langage CUDA, sur lequel l'implémentation parallèle se fait est dérivé du C, ainsi la transition passe par un codage en C de l'algorithme. Le code est disponible en annexe IV.

On compare les résultats obtenus (paramètres identiques), disponibles en figure 24.



Résultat code C Code OpenCV

Figure 24. Code C et référence OpenCV

Le code C est fidèle au code Matlab préalablement défini, tout en étant plus performant - notamment en terme de régularité - que le code openCV de référence. Par ailleurs, son exécution (pour ces paramètres) s'effectue en 7 secondes, soit 30x plus vite que la version Matlab (valeur classique lorsque l'on traduit un code Matlab en C).

IV – Validation de l'algorithme et Tenseur Voting

Une fois l'algorithme clairement défini, il faut évaluer clairement les performances obtenues et se doter d'indicateurs permettant le réglage précis des paramètres de l'algorithme. C'est ce qui est développé dans cette partie.

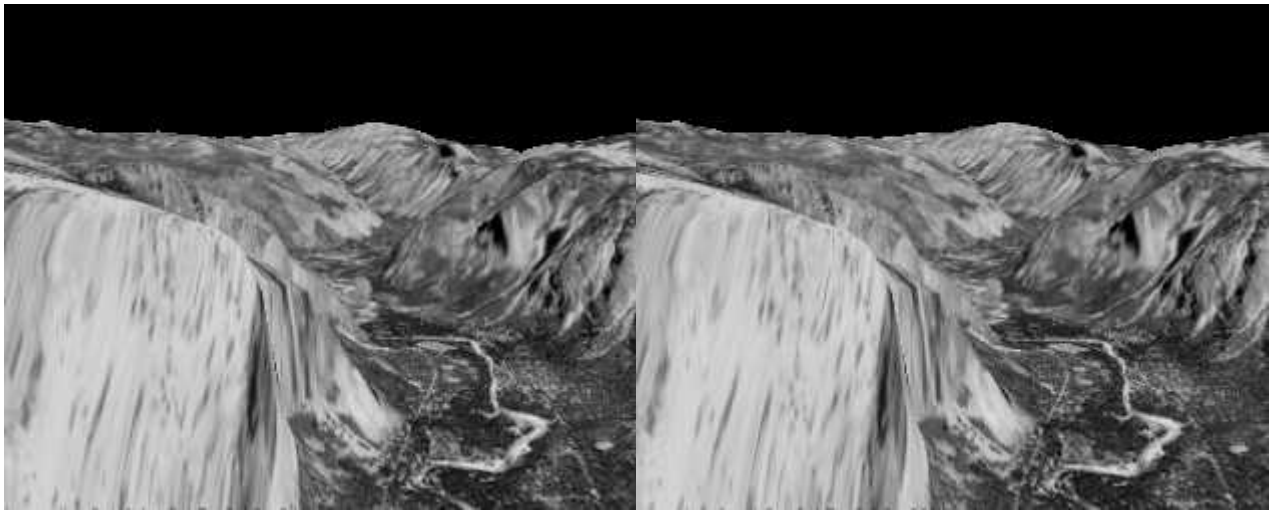
1. Validation : Ground Truth Yosemite

(a) *Contexte*

Pour pouvoir comparer deux algorithmes d'estimation du flot optique, il faut disposer d'images de référence comme toujours en traitement d'image. On doit disposer du flot réel, c'est pourquoi ces images sont nécessairement synthétiques. Une des séquences les plus utilisées est celle nommée Yosemite, du nom du parc américain contenant la vallée présentée figure 25. L'avantage de cette séquence est d'être composée de mouvements complexes (composés) et non de simples rotations ou translations.

A partir du flot réel, on peut donc d'une part évaluer les performances de l'algorithme développé et d'autre part étudier l'influence des paramètres sur l'erreur, de manière à les choisir le plus judicieusement possible.

Séquence test Yosemite :



Frame 10 Frame 11

Figure 25. Séquence de référence Yosemite

(b) Calcul de l'erreur angulaire et de l'erreur en norme

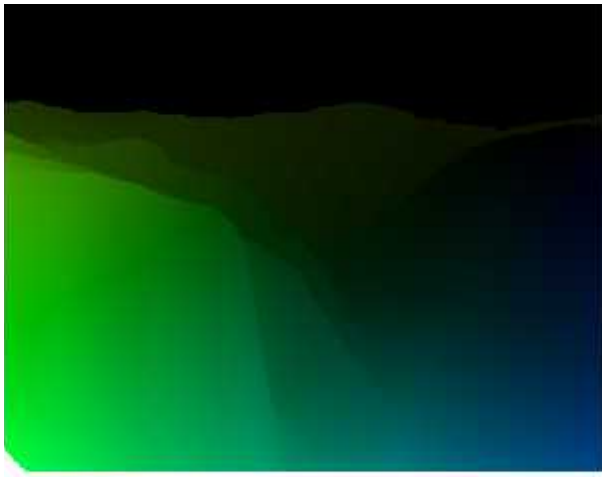
Les formules classiquement admises et utilisées dans la littérature [5] pour le calcul de l'erreur angulaire et de la norme de l'erreur sont les suivantes :

Soient (u_c,v_c) le flot calculé et (u_r, v_r) le flot réel, alors :

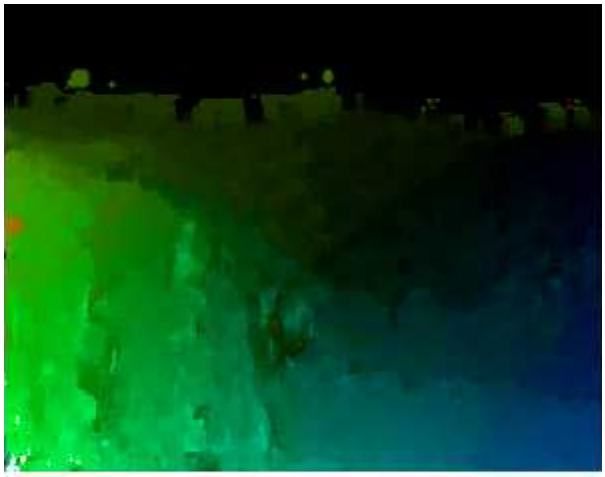
$$AAE = \frac{1}{(N.M)} \sum_{i=1}^{N} \sum_{j=1}^{M} \arccos\left(\frac{u_r u_c + v_r v_c + 1}{\sqrt{(u_r^2 + v_r^2 + 1)(u_c^2 + v_c^2 + 1)}} \right) \quad \text{et} \quad Norm = \sqrt{((u_r - u_c)^2 + (v_r - v_c)^2)}$$

2. Etude de l'influence des paramètres de l'algorithme retenu (Type Lucas Kanade)

Exemple de flot calculé pour 3 niveaux, 1 itération, taille de patch = 10x10:



Flot réel



Flot calculé

(a) Nombre de niveaux de pyramide

On conserve fixe le nombre d'itérations (1) et la taille du patch (10x10) et l'on fait varier le nombre de niveau de la pyramide. Le nombre maximal de niveaux autorisé pour Yosemite est de 3, car l'image initiale est de taille 252x316, et donc le troisième niveau a pour taille 63x79. Le niveau suivant aurait une taille non entière, ce qui pose des problèmes numériques. Ce qui apparaît clair sur les résultats (synthétiques ou réels) est qu'à partir de 3 niveaux, l'estimation varie peu. 3 ou 4 niveaux semble donc tout à fait indiqués. Par ailleurs, le temps de calcul des « petits » niveaux est infime.

(b) Taille du patch

On garde fixe le nombre de niveaux de la pyramide (3), le nombre d'itérations (1 seule), et l'on fait varier la taille du patch sur lequel la vitesse est considérée constante. On obtient :

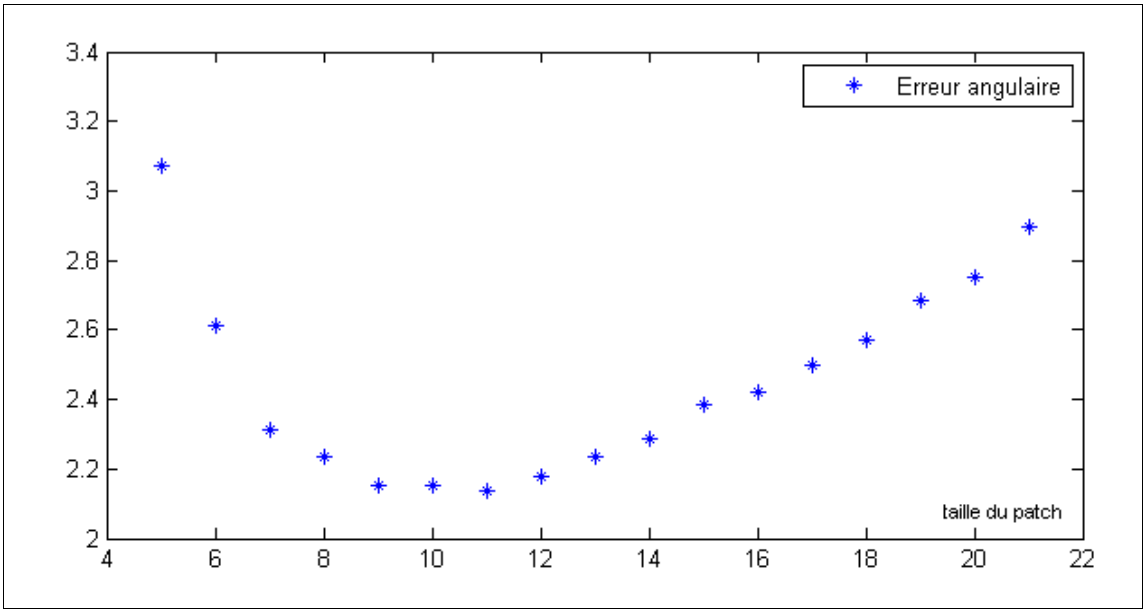
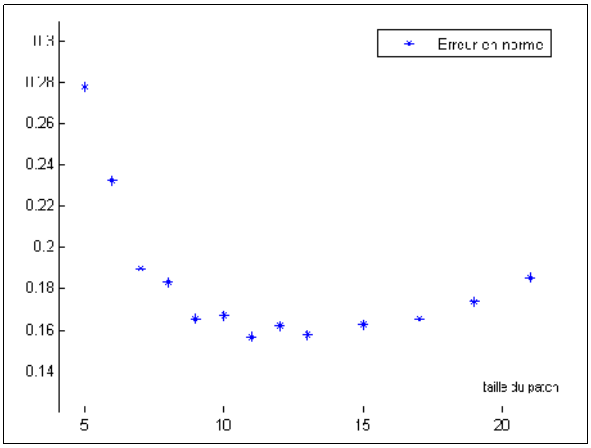


Figure 26. Courbes des erreurs en fonction de la taille du patch, en Lucas Kanade Pyramidal, avec 3 niveaux



Sur cette séquence, les tailles de patchs minimisant au mieux les erreurs sont 9x9, 10x10 et 11x11. Cela conforte ce que l'on avait remarqué lors des premiers tests de l'algorithme sur les images de la séquence réelle.

Par ailleurs, l'erreur angulaire et l'erreur en norme sont clairement corrélées.

Taille de patch retenue = 9x9, 10x10 ou 11x11

(c) Nombre d'itérations

On conserve désormais fixe le nombre de niveaux (3) et la taille du patch (10x10), et l'on fait varier le nombre d'itérations que l'on fait à chaque niveau. On observe la convergence de l'erreur comme suit :

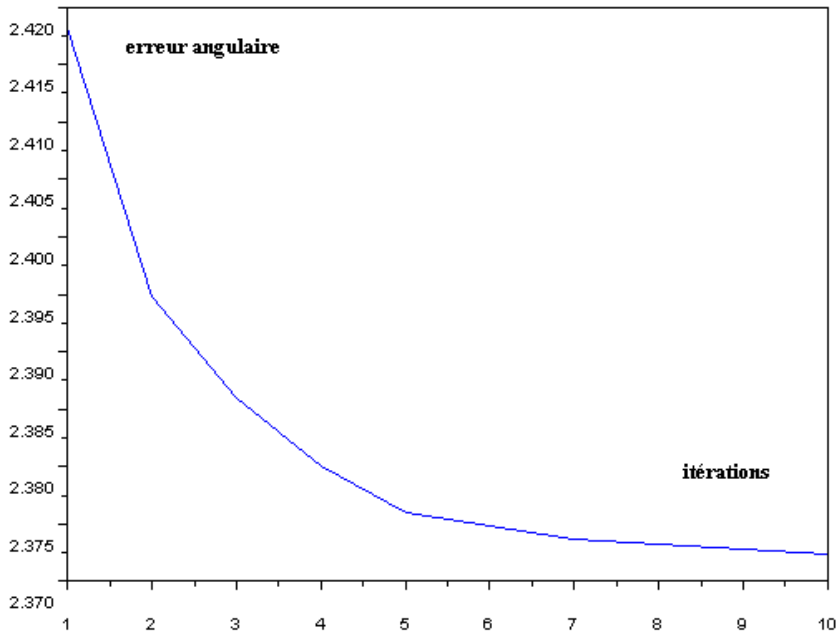


Figure 27. Évolution de l'erreur angulaire en fonction du nombre d'itérations

La convergence est rapide (au bout de 5 itérations, la valeur finale est quasiment atteinte). L'utilisation d'itérations augmente considérablement le temps de calcul (il est tout simplement multiplié par le nombre d'itérations réalisées). Le meilleur compromis temps de calcul/résultat est donc entre 2 et 4 itérations.

(d) Coefficient de Régularisation

Le coefficient de régularisation sert simplement à éviter la nullité du déterminant lors de l'estimation des moindres carrés, sa valeur doit donc être proche de zéro, entre 10^{-3} et 10^{-6}. On n'observe pas de variation de l'erreur et aucune inverse de matrice nulle, le coefficient remplissant son office.

(e) Retour sur la méthode d'Horn et Schunck

Revenons sur la méthode d'Horn & Schunck (méthode globale) que nous avons écartée mais qu'il est intéressant d'évaluer à titre de comparaison dans l'optique de la validation.

On fait varier le paramètre α et on trace l'erreur :

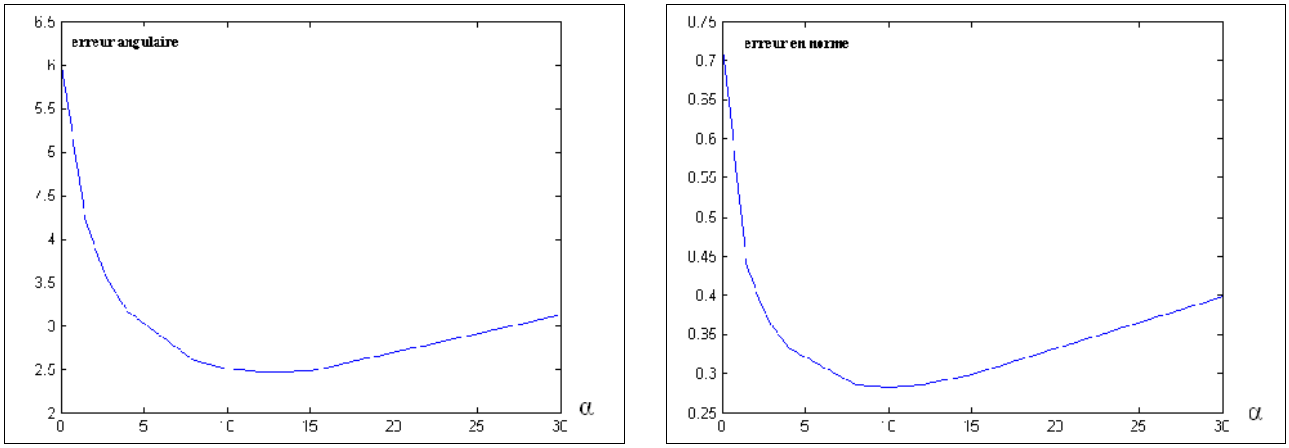


Figure 28. Erreur angulaire et erreur en norme en fonction de α pour la séquence Yosemite avec la méthode Horn Schunck Pyramidale

On constate donc que l'on atteint un minimum pour la valeur $\alpha = 13$.

Variation du nombre d'itérations, pour la valeur « optimale » $\alpha = 13$ et 3 niveaux de pyramide :

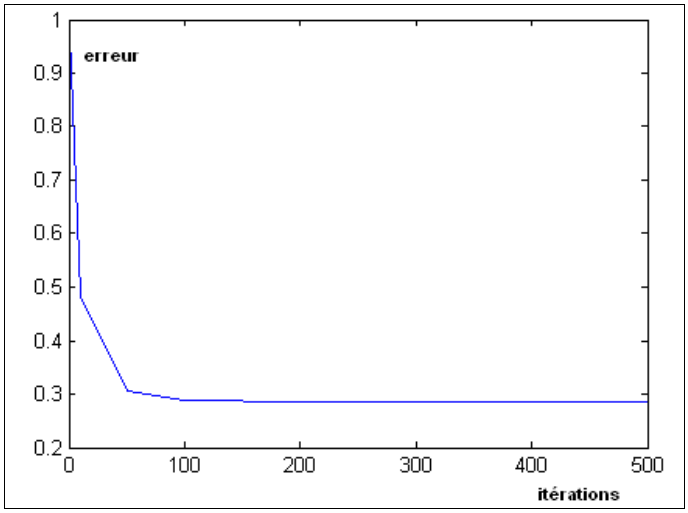


Figure 29. Erreur angulaire et erreur en norme en fonction du nombre d'itérations

On observe une convergence rapide de l'erreur, et le choix de 50 ou 100 itérations semble approprié.

Avec le réglage plus précis des paramètres, on obtient des résultats convenables avec la méthode d'Horn & Schunck, toutefois bien inférieurs à ceux obtenus avec l'algorithme choisi.

(f) Block Matching

Une autre technique dense jugée moins efficace que l'algorithme retenu est le Block Matching. On peut confirmer le choix qui a été fait au regard des deux résultats suivants :

	Erreur Angulaire	Erreur en norme
Block Matching, mb=16, r=8 pixelique	4.5001	0.3410
Block Matching, mb=16, r=8 subpixelique (suréchantillonné une fois)	3.1126	0.3889

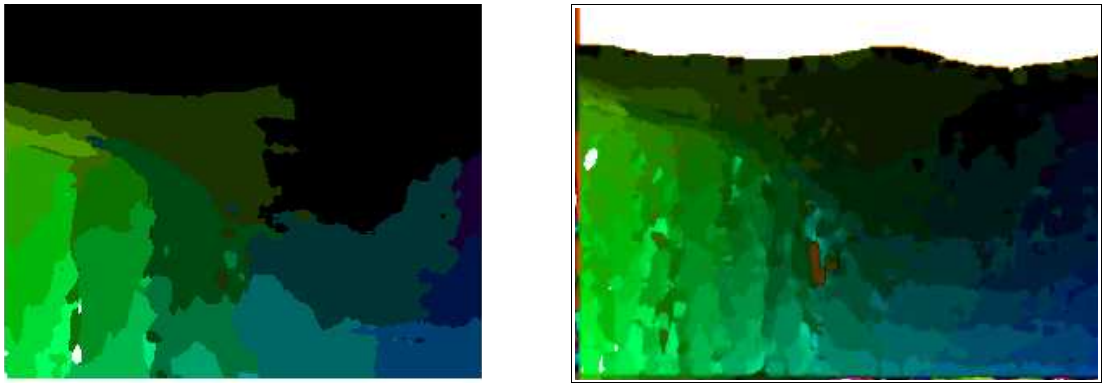


Figure 30. Flot optiqueYosemite par technique de Block Matching (droite : suréchantillonnée)

Les résultats obtenus (figure 30) sont de très faible qualité (+ 150% d'erreur par rapport à l'algorithme choisi). L'amélioration possible est d'effectuer l'estimation sur une image sur-échantillonnée, de taille deux fois supérieure à l'image originale, mais le temps de calcul supplémentaire est très long et le résultat encore médiocre (+ 73% d'erreur).

Je vais maintenant présenter ce qu'est le Tensor Voting et en quoi ce processus peut améliorer l'estimation du flot optique.

3. Le Tensor Voting

Le *Tensor Voting* [39] est un formalisme introduit à la fin des années 90 par Gérard Médioni pour identifier les structures géométriques auxquelles appartient tout point d'un espace de dimension quelconque, en fonction de l'agencement de son voisinage. Des tenseurs symétriques du second ordre codent, en chaque point, l'information structurelle acquise au cours d'un processus de communication, à l'issu duquel leur décomposition en tenseurs élémentaires décrit la géométrie induite par le jeu de données original.

Un tenseur 2D peut être représenté géométriquement par une ellipse (les axes de celle-ci étant orientés selon les vecteurs propres). Il se décompose de la manière suivante :
$T=\lambda_1 \hat{e}_1 \hat{e}_1^T+\lambda_2 \hat{e}_2 \hat{e}_2^T=(\lambda_1-\lambda_2)\hat{e}_1 \hat{e}I^T+\lambda_2(\hat{e}_1 \hat{e}_1^T+\hat{e}_2 \hat{e}_2^T)$ où λ_1,λ_2 sont les valeurs propres et $\hat{e}_1,\hat{e}_2$ les vecteurs propres correspondants.

Les tenseurs dégénérés « boule » et « stick » sont caractérisés par chacun des termes. Lorsque $\lambda_1=\lambda_2$, le tenseur est isotrope (boule) et lorsque $\lambda_2=0$, l'ellipse est réduite à son grand axe: tenseur stick (figure 31).

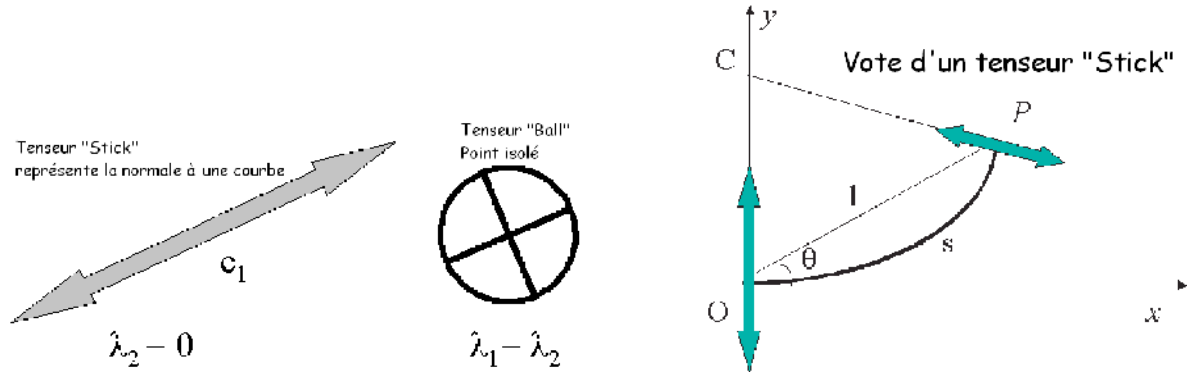


Figure 31. Tenseurs stick et boule, vote d'un tenseur stick

Les tenseurs boule isotropes permettent de coder les points isolés, les tenseurs stick permettent de coder les normales en chaque point de la courbe.

Le processus de vote s'effectue par accumulation, de manière à spécialiser chaque tenseur en fonction de la disposition des tenseurs de son voisinage selon un profil gaussien d'écart-type σ. Le vote d'un tenseur boule est obtenue en intégrant sur 2π le champ de vote d'un tenseur stick. Pour de plus amples informations, se reporter au livre de G.Medioni [39].

Ce processus décrit en 2-D se généralise aisément en dimension N. En effet, la décomposition 4-D d'un tenseur s'écrit :

$$T = (\lambda_1 - \lambda_2)\hat{e}_1\hat{e}_1^T + (\lambda_2 - \lambda_3)(\hat{e}_1\hat{e}_1^T + \hat{e}_2\hat{e}_2^T) + (\lambda_3 - \lambda_4)(\hat{e}_1\hat{e}_1^T + \hat{e}_2\hat{e}_2^T + \hat{e}_3\hat{e}_3^T) + \lambda_4(\hat{e}_1\hat{e}_1^T + \hat{e}_2\hat{e}_2^T + \hat{e}_3\hat{e}_3^T + \hat{e}_4\hat{e}_4^T)$$

Dans notre cas, on s'intéresse à l'espace 4-D (x, y, ux, uy) formé par les coordonnées d'un pixel de l'image et des vitesses selon les axes qui lui sont associées. Dans cet espace, les objets (régions de vitesse homogène) forment des surfaces identifiées par le tenseur élémentaire $e_1 e_1^T + e_2 e_2^T$ caractérisé par le coefficient $\lambda_2 - \lambda_3$.

Il est ainsi possible d'améliorer le calcul du flot optique en réalisant plusieurs estimations (taille de patch différente notamment ce qui donne un grain plus ou moins fin à l'estimation), puis de confier au Tensor Voting le nuage de point 4-D formé par ces deux estimations. Il s'agira ensuite à l'issue du traitement, de prendre pour chaque point l'estimée qui maximise le coefficient de surface $\lambda_2 - \lambda_3$ pour obtenir un flot respectant au mieux en chaque point la continuité imposée par son voisinage et donc les objets connexes.

4. Utilisation de plusieurs estimations

On effectue quelques tests préliminaires sur la séquence Yosemite avec notre algorithme :

Méthode	AAE (°)	Ecart en norme (pixel)
LK 3 niveaux, patchs 9 et 15	2.1232	0.1490
Patchs 7,13,19	2.4518	0.1614
Patchs 9, 10 et 11	2.0183	0.1501
Patchs 9 et 12	2.0045	0.1474
Patchs 9,10,11 et 12	2.0293	0.1484

L'utilisation des tailles de patch donnant les erreurs minimales suivi d'un traitement Tensor Voting améliore la précision de l'estimation.

Pour les tailles de patch donnant les meilleures estimations (9,10,11) :

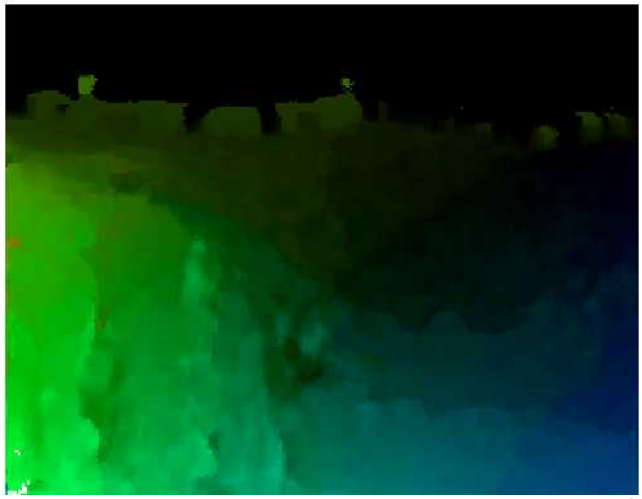


Figure 32. Tensor Voting sur les trois meilleures estimées

Il est ensuite possible d'étudier l'Influence du paramètre d'échelle σ du Tensor Voting sur l'erreur mesurée pour choisir au mieux celui-ci:

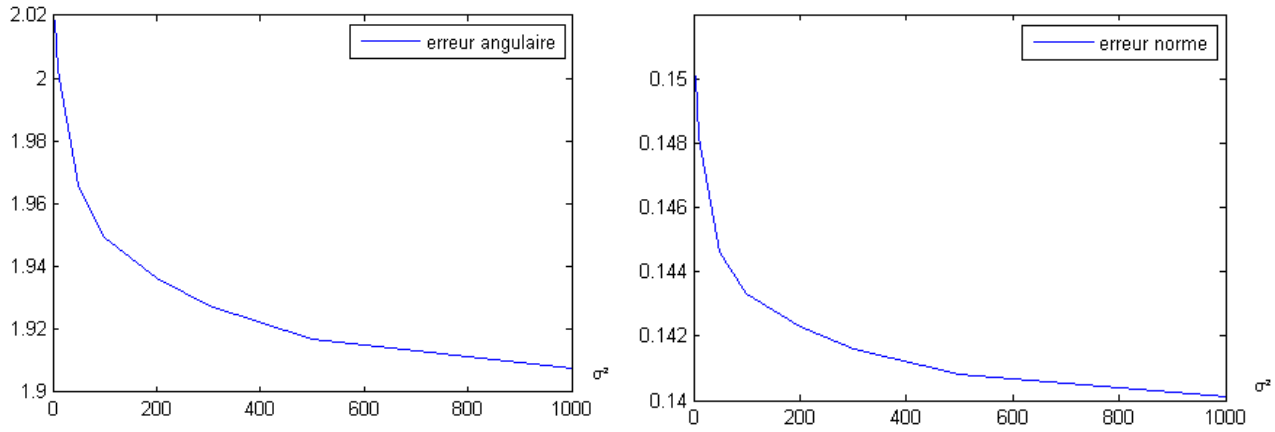


Figure 33. Evolution de l'erreur en fonction du paramètre d'échelle du Tensor Voting

Le changement de pente se trouve à la valeur $\sigma^2 = 150$, qui semble donc être la valeur optimale pour le paramètre d'échelle.

Pour cette valeur, l'amélioration en terme d'erreur par rapport au résultat obtenu directement avec un patch 10x10 est de **10 %**.

Remarque : à titre d'extension, le même raisonnement est conduit avec la méthode d'Horn & Schunck en annexe V.

Il est donc intéressant de recourir à cette technique, ne serait-ce que sur deux estimées du flot, l'amélioration étant significative. Il faut bien sûr réaliser le meilleur compromis précision / temps de calcul, et cela va dépendre des résultats obtenus en terme de temps de calcul à l'aide de l'implémentation parallèle sur GPU qui est détaillée dans toute la partie suivante.

V – GPU, CUDA, Implémentation Temps réel

1. GPU et CUDA

(a) Utilisation des architectures parallèles en calcul scientifique

L'objectif du calcul parallèle est la réduction significative du temps de calcul du processus considéré, le plus souvent pour rendre le processus viable en temps réel. Historiquement, les logiciels étaient écrits pour un traitement séquentiel et pour être exécuté sur une seule machine avec une seule unité de calcul. Le développement des approches parallèles est donc assez récent et ouvre de nombreuses possibilités.

Quatre types de fonctionnement sont possibles concernant les calculateurs, selon la classification de Flynn [47] :

	Single Data	**Multiple Data**
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

- SISD (Single Instruction, Single Data), c'est le mode de fonctionnement classique des ordinateurs, traitant un seul processus à un instant donné s'exécutant sur un seul canal de données.

- SIMD (Single Instruction, Multiple Data), c'est l'architecture des GPU que nous utiliserons, à savoir l'exécution de plusieurs threads en parallèle contenant la même instruction s'appliquant à des jeux de données différents.

- MISD (Multiple Instructions, Single Data), flux d'instructions différentes sur un même flux de données, cette structure est peu pratique et donc peu utilisée.

- MIMD (Multiple Instructions, Multiple Data), niveau maximal de parallélisme : plusieurs instructions sur des données différentes.

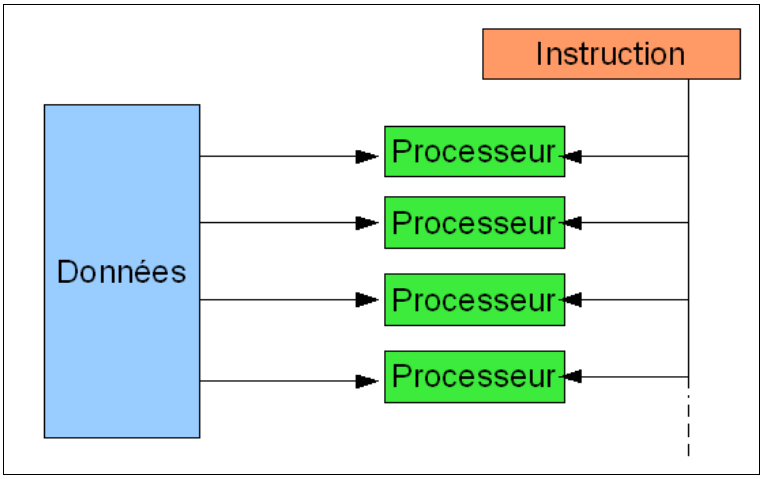


Figure 34. Architecture SIMD

(b) Récentes avancées : des processeurs graphiques dédiés au calcul

Les processeurs de carte graphique (GPU : Graphics Processing Unit), dédiés à l'affichage ont évolués très rapidement, boostés notamment par l'industrie du jeu vidéo et des univers 3D. Leur développement a ainsi été beaucoup plus rapide que celui des processeurs classiques, ces derniers étant également limités en fréquence et donc proches de leur maximum.

Les processeurs classiques sont conçus pour traiter un flux d'instructions, tandis que les GPU établissent un lien entre un ensemble de polygones et un ensemble de pixels, chaque élément étant dissociable.

Les premières tentatives de programmation scientifique à l'aide de processeurs de carte graphique (GPGPU : General-Purpose computing on GPU) utilisaient les structures existantes servant à l'affichage pour effectuer des calculs parallèles, et donc pas un outil dédié au calcul.

La démocratisation de ce type d'approche a amené un constructeur de cartes graphiques, NVIDIA, à produire des chipsets graphiques, paradoxalement sans sortie vidéo, dédiées au calcul scientifique et donc menant à des performances beaucoup plus intéressantes que l'utilisation des cartes graphiques standard.

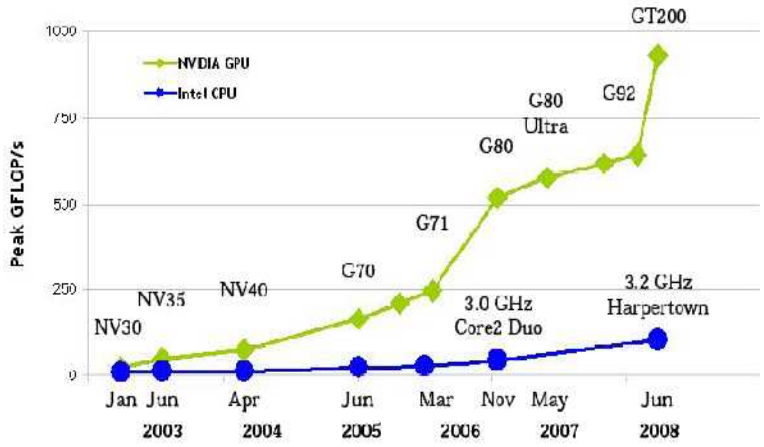


Figure 35. Progrès récents en terme de capacité de calcul des GPU NVIDIA

La véritable révolution ayant amené le monde de la recherche à s'intéresser à la programmation parallèle sur GPU est la sortie en 2007 d'une API pour la programmation de ces cartes : CUDA.

(c) CUDA (Compute Unified Device Architecture)

CUDA [42] est une interface et un langage de programmation dérivé du C qui permettent d'utiliser les GPU (de marque NVIDIA, G80 et suivants) pour effectuer du calcul massivement parallèle.

Le matériel utilisé au cours du stage consiste en deux cartes Nvidia Tesla C870 (figure 36) (chipset de type G80) qui peuvent collaborer.

Les cartes (*device*) sont vues comme des co-processeurs par l' *Host*, c'est à dire le cpu. Les programmes sont écrits en C, éxécutés sur le CPU et font appel aux GPU par le biais de CUDA.

Le langage et l'approche se veulent génériques, ainsi les programmes écrits en CUDA pour les cartes existantes seront compatibles avec les futurs GPU, le dernier en date et le plus performant étant le GT200 (deux fois plus performant que les Tesla utilisées)(voir figure 35).

Figure 36. Carte NVIDIA Tesla C870

Les GPU sont constitués d'un assemblage de multiprocesseurs indépendants, formant un total de 128 multiprocesseurs par unité de calcul pour l'architecture qui nous occupe.

CUDA demande au développeur d'organiser la masse de travail à exécuter par le GPU en un assemblage de blocs de threads. Chaque bloc peut contenir jusqu'à 512 threads. Plusieurs blocs peuvent être actifs en même temps dans un même multiprocesseur.

De manière à ce que l'approche soit générique, la parallélisation est organisée sur trois niveaux :

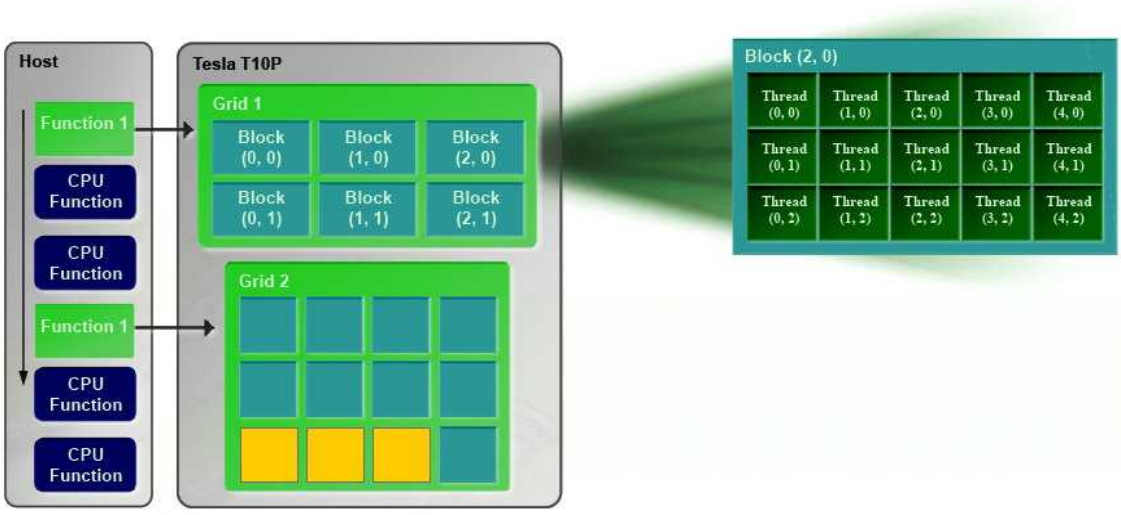


Figure 37. CUDA : Trois niveaux d'abstraction pour la programmation parallèle

Chaque thread contient la même instruction. Chaque bloc de thread est exécuté sur un multiprocesseur, les blocs étant contenus dans une grille de blocs de threads : si le nombre de blocs contenus dans la grille est supérieur au nombre de multiprocesseurs, les blocs sont placés en file d'attente. Une fois que l'exécution de la fonction GPU est terminée, le CPU reprend la main.

En ce qui concerne la mémoire, chaque thread dispose de 32 registres de 4 octets permettant de stocker les variables temporaires ou d'écriture. En plus de cela, chaque bloc de thread dispose d'une « shared memory » de 16 Ko. Au niveau supérieur, la carte Tesla dispose d'une mémoire physique (DRAM) appelée « global memory » où sont stockées les éléments calculés ou nécessaires au calcul. Il existe également une mémoire particulière, la « texture memory » qui permet de manipuler aisément des tableaux (l'interpolation bilinéaire est réalisée en quelques cycles d'horloge). [44]

Remarque concernant la mémoire : les transferts de mémoire en interne sur la carte sont très rapides. Par contre les transferts de mémoire du CPU vers le GPU sont très lents. Il est ainsi conseillé d'effectuer le maximum de calcul en restant sur le GPU. Ceci est possible car la « global memory » est de taille conséquente : 1,5 Go.

Les fonctions CUDA sont appelées « Kernel ». Pour pouvoir les lancer, il faut préalablement calculer le nombre de threads par bloc et le nombre de blocs par grille que la carte va devoir gérer. Un certain nombre de contraintes s'appliquent sur ces paramètres, par exemple le nombre maximal de threads par bloc est de 512. Le kernel se lance depuis le programme C selon la syntaxe suivante :

```
int main()
{
//appel du kernel KernelSample

SampleKernel<<<dimGrid, dimBlocks>>>( );

}
```

Une contrainte majeure est l'unicité d'exécution simultanée des kernels : lorsqu'un kernel est actif, aucun autre appel de kernel n'est possible. Un kernel ne peut donc appeler un autre kernel de manière à avoir une dimension de parallélisme supplémentaire. Cela sera sans doute possible dans des versions ultérieures.

Maintenant que l'environnement a été décrit dans ses grandes lignes, je vais présenter plus précisément l'usage qui en a été fait pour rendre temps réel l'estimation du flot optique.

2. Détail de la parallélisation de l'algorithme

(a) Décomposition de l'algorithme

L'idée selon laquelle l'algorithme utilisé est parallélisable part du simple constat suivant : l'estimation en chaque pixel est indépendante des estimations voisines. A partir de là, il est possible de confier l'estimation en chaque pixel à un kernel, de manière à ce que chaque thread s'occupe de l'estimation en un pixel donné.

Plus précisément, les étapes parallélisables sont :

- **La construction des pyramides :** ce calcul est à la fois séquentiel et parallélisable. En effet, s'il faut avoir calculé le niveau inférieur pour calculer le niveau courant, la valeur de chaque pixel ne dépend pas des valeurs alentour, il est donc possible de confier le calcul d'un niveau de pyramide à un Kernel et d'exécuter celui-ci dans une boucle pour construire la pyramide entière.

- **Le calcul des images dérivées :** ce calcul se fait à chaque changement de niveau de pyramide, une seule fois pour chaque image. La dérivée en chaque pixel est indépendante des dérivées voisines, cela ne pose aucun problème de parallélisation.

- **L'interpolation (doublement de la taille)** des vitesses du niveau supérieur : de la même manière, la vitesse en un point donné est indépendante des vitesses voisines.

- **Le calcul en lui même :** à l'aide des objets définis précédemment, le calcul en chaque pixel est indépendant du reste de l'image, chaque élément allant accéder (à l'aide de son numéro de thread) aux informations nécessaires au calcul qui auront été définies en amont.

Les parties qui restent immuablement séquentielles sont :

- L'initialisation des objets utilisés pour le calcul
- La boucle sur le nombre de niveaux
- Le raffinement itératif dans un même niveau (cela sera toutefois inclus dans le kernel de calcul)

(b) Choix des objets

Pour mener à bien le calcul, il nous faut stocker :

- les pyramides des deux images successives utilisées
- les dérivées (spatiales et temporelles) de chaque image, au niveau courant
- les champs de vitesse du niveau courant et précédent

CUDA est optimisé pour manipuler des *texture* en lecture et de la mémoire linéaire alignée (tableaux de *float*) en écriture. Ainsi, avant l'appel de chaque Kernel, il faut lier (fonction dédiée *cudaBindTextureToArray*) les tableaux utilisés à des textures de manière à faire appel à ces textures dans le kernel et optimiser la durée de lecture.

En revanche, l'écriture se fait dans de la mémoire linéaire alignée, il faut donc à la sortie des kernels (lorsque tous les threads ont été exécutés) copier cette mémoire dans les tableaux dédiés à la lecture, ces transferts de mémoire sont immédiats. Il y a donc une double allocation mémoire pour les éléments décrits ci-dessus, qui se justifie par le gain de temps qu'elle procure.

(c) Structure du programme

Le programme C/CUDA final comporte en premier lieu une fonction main (dans un fichier .c) qui fait appel aux fonctions d'initialisation, de calcul et qui réalise la mesure du temps d'exécution. Toutes les fonctions C réalisant les appels de Kernel sont rangées dans un « .cu », type propre à CUDA, qui nécessite le compilateur nvcc dédié. Finalement, tous les Kernel appelés sont dans un .cu qui n'est pas compilé avec le reste du programme, le fichier précédent réalisant les liens nécessaires.

Il faut noter que l'étape d'initialisation (allocation mémoire) qui prend un temps certain, ne se compte pas dans le temps de calcul total, car cette phase ne s'effectue qu'une seule fois au démarrage du véhicule, les mêmes objets sont toujours réutilisés par la suite, avec simplement de nouvelles images qui sont acquises au fur et à mesure.

Le listing commenté des différents kernels est disponible en annexe, ainsi que le détail du programme séquentiel. L'implémentation CUDA est résumée dans le schéma ci-après (figure 38).

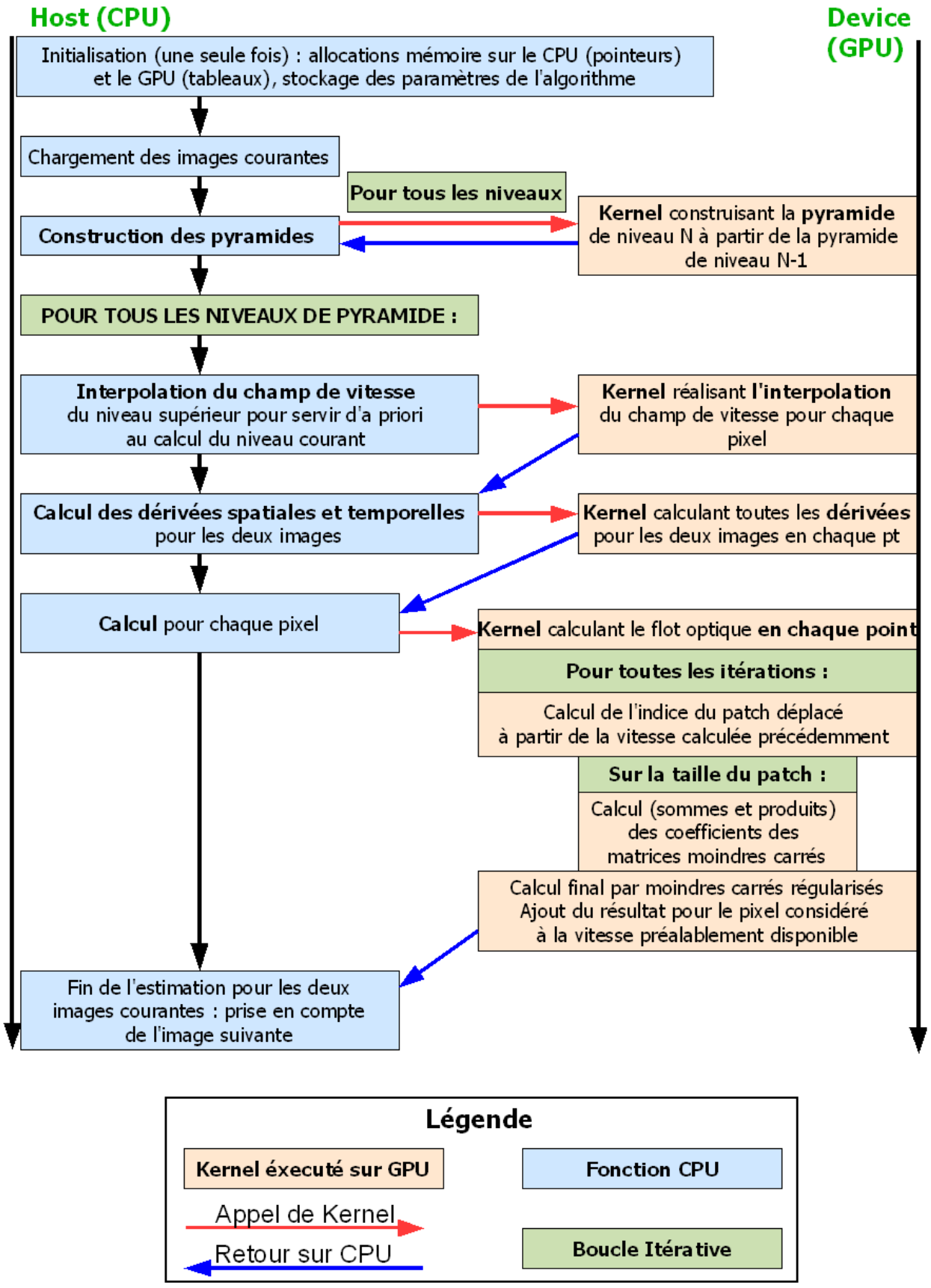


Figure 38. Description de l'implémentation CUDA de l'algorithme

3. Optimisations

(a) Gestion de la mémoire

Une des principales améliorations préconisée par Nvidia est la gestion de la mémoire. En effet, le nombre de registres pour chaque Kernel étant limité à 32 float, il faut prendre garde à limiter les variables intermédiaires. Il est possible de compiler le fichier contenant les Kernel avec une option renvoyant un fichier texte contenant les occupations mémoire de chaque Kernel. Par exemple, pour le Kernel de calcul :

```
code  {
      name = _Z13LKCudaLKerneljjjfPfS_jjj
      lmem = 0    place mémoire occupée sur la mémoire de la carte
      smem = 52   place mémoire occupée en shared memory (commune à un bloc)
      reg = 28    place mémoire occupée dans les registres (32 bytes max)
      bar = 0
      bincode  {[...]}
```

Cela nous informe que 52 float sont utilisés en shared memory ainsi que 28 registres, pour ce kernel. Il est possible d'utiliser plus de shared memory et moins de registres mais le gain en temps n'est pas évident. Une Tesla garantit 8192 registres de 32 bits. Il faut donc limiter le nombre de threads actifs en même temps à 8192.

Une fois que l'on s'est assuré que tous les kernels n'utilisent pas plus de ressource mémoire qu'il ne doivent, il faut régler le nombre de threads par blocs pour chaque kernel. Celui-ci doit être compris entre 64 et 512. La taille de la shared memory est de 16 KB, il faut ainsi respecter la contrainte suivante : $Nb_{threads} * smem < 16\,KB$.

Par exemple, dans le cas du Kernel développé ci-dessus, une taille de bloc de 16x16 (256 threads) mène au calcul suivant $Nb_{threads} * smem = 256 * 52 = 13\,312\,bytes < 16\,KB$. Le choix de 256 threads par bloc est donc possible pour ce kernel, une autre puissance de 2 dépasserait largement la valeur permise.

Par ailleurs, à l'aide des informations fournies par le fichier et les valeurs limites qu'il est possible d'allouer pour chaque kernel, NVIDIA propose sur son site (http://www.nvidia.com/cuda) un outil permettant de mesurer l'*occupation* des multiprocesseurs en fonction de la mémoire utilisée à la fois en shared memory et en terme de registres ainsi que des tailles de bloc retenues.

L'*occupation* d'un multiprocesseur est le rapport entre le nombre de groupes de 32 threads (appelés « warp ») actifs et le nombre maximal de warps suppportés par un multiprocesseur du GPU. Cela donne une bonne idée de l'utilisation du processeur, même si maximiser l'occupation ne revient pas toujours à maximiser les performances.

Le calculateur prend en compte le modèle de la carte utilisée (G80 dans le cas présent), et calcule le nombre de warps actifs en fonction des valeurs de mémoire renseignées.

Pour le kernel de calcul utilisé, qui utilise beaucoup de ressources mémoire (irréductibles), l'occupation ne peut être réduite en dessous de 33%, comme le montre le tableau et les courbes disponibles en page suivante .C'est donc la mémoire qui limite l'utilisation des processeurs, ce qui est logique au vu du Kernel (annexe VIII) : il s'agit principalement d'écriture/lecture en mémoire globale de la carte.

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	28
Shared Memory Per Block (bytes)	13312
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**256**
Active Warps per Multiprocessor	**8**
Active Thread Blocks per Multiprocessor	**1**
Occupancy of each Multiprocessor	**33%**
Maximum Simultaneous Blocks per GPU	**16**
Physical Limits for GPU:	**G80**
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384

Figure 39. Calculateur d'occupation

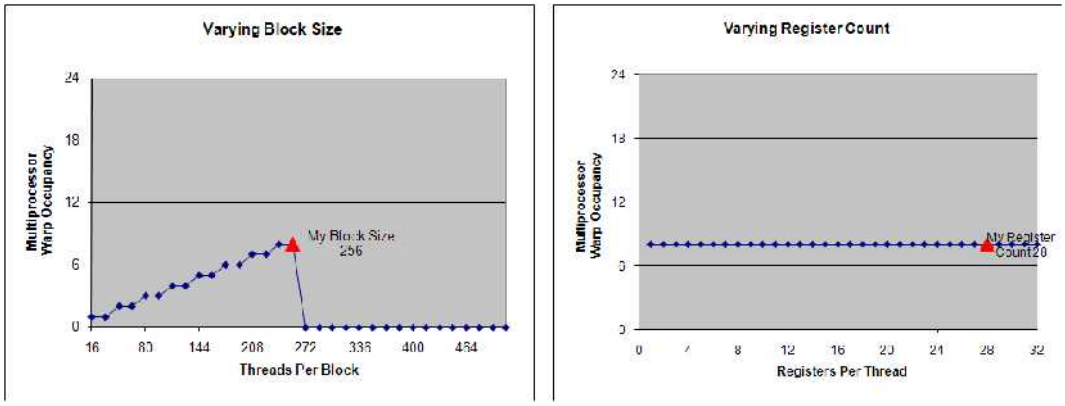


Figure 40. Courbes d'occupation pour le kernel Calcul

Le pic d'occupation est clairement atteint pour la mémoire occupée par ce kernel. Une diminution de la taille du bloc ne fera que diminuer l'occupation des multiprocesseurs.

Pour les Kernels utilisant moins de mémoire, comme ceux effectuant la construction des niveaux de pyramide ou encore l'interpolation du champ de vitesse du niveau supérieur, il est plus aisé d'optimiser l'occupation, par exemple pour le kernel LKCudaPyrKernel, smem = 36 et reg = 13 :

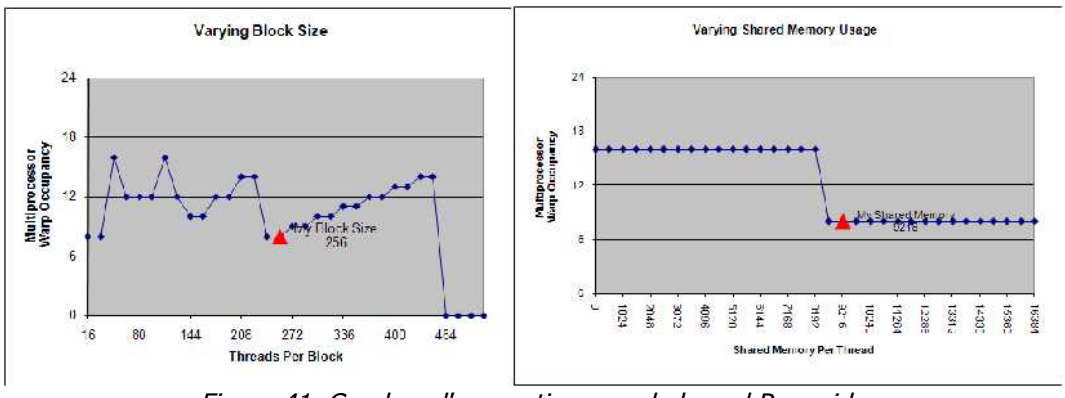


Figure 41. Courbes d'occupation pour le kernel Pyramide

On constate donc qu'il est possible d'augmenter l'occupation en diminuant la taille du bloc. En effet, pour 100 threads par bloc l'occupation monte à 67%. Le temps d'exécution de ce Kernel était déjà très court (de l'ordre de 0.6 ms), il chute à 0.4 ms.

On procède de même pour les autres kernels.

(b) Utilisation du Profiler

NVIDIA met également à disposition avec la version bêta de CUDA 2.0 un profiler qui permet de voir le temps passé dans chacun des kernels. Par exemple, pour une exécution comprenant un seul niveau et une seule itération, on peut analyser le temps passé dans chaque kernel pendant toute l'exécution :

	Method	#Calls	GPU usec	CPU usec	%GPU time	gst coalesced	branch	divergent branch	instructions	cta launched
1	LKCudaLKernel	1	16578	16986,4	87,6	19200	72643	121	1511947	150
2	LKCudaDerivKernel	1	513,12	643,478	2,71	57600	600	0	57001	150
3	LKCudaResetKernel	1	59,648	73,724	0,31	19200	600	0	6923	150
4	memcopy	4	1771,78		9,36					

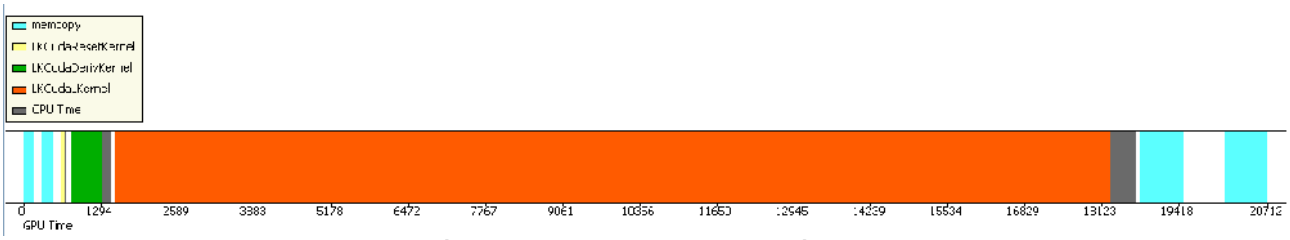


Figure 42. Résultats du Profiler pour une itération, un niveau

Ce qui prend clairement le plus de temps est le Kernel de calcul (87,6%), ce qui était attendu car celui-ci contient une boucle itérative sur le nombre de points du patch (calcul des éléments des matrices de calcul des moindres carrés : somme et produits sur 100 points pour un patch 10x10).

On constate également que le programme reste très peu de temps sur CPU, ce qui confirme que la majeure partie du programme s'effectue sur GPU, ce qui est recommandé. Les copies mémoire présentes au début et à la fin (bleu clair sur le graphique) représente le chargement des images depuis le CPU vers le GPU et la copie des résultats pour affichage depuis le GPU vers le CPU.

Pour une exécution plus complète (4 niveaux, 3 itérations, patch 10x10), on visualise l'histogramme des Kernels :

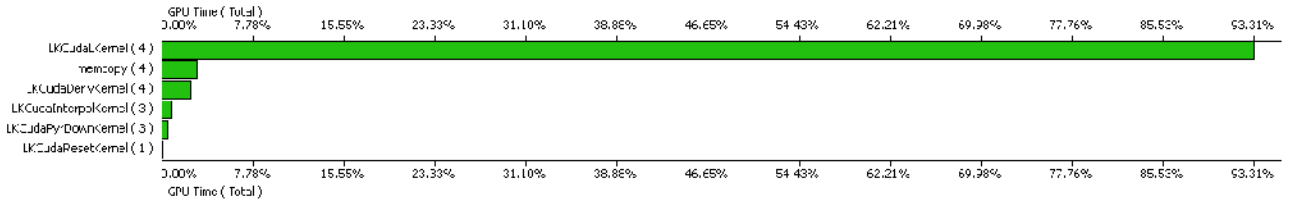


Figure 43. Histogramme relatif des kernels

Cet histogramme relatif permet de vérifier que les Kernels se sont bien éxecutés le nombre de fois prévus par l'algorithme. On retrouve les résultats précédents en terme de temps d'occupation relatif.

Il est également possible de visualiser l'enchaînement des Kernels et leurs temps d'exécution respectifs.

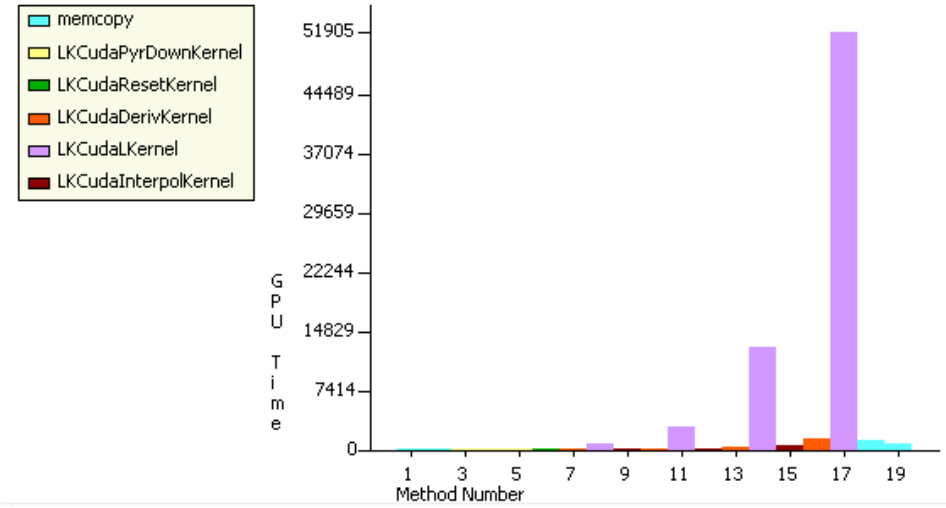


Figure 44. Histogramme absolu fourni par le profiler

Cela permet de détecter les anomalies d'exécution, d'une part au niveau de l'enchaînement des fonctions et d'autre part au niveau des temps d'exécution.

(c) Calcul de la bande passante et de la puissance de calcul

Une dernière vérification de la bonne programmation de l'algorithme et de son optimisation est la mesure de deux grandeurs caractéristiques.

La bande passante (quantité de mémoire manipulée par seconde) théorique est de l'ordre de 4 Go/s pour l'interface GPU-CPU. Concernant l'interface GPU-GPU (copie mémoire sur la carte), la bande passante est de 80 Go/s (valeur pic).[43]

Comme la totalité des calculs des kernels s'effectue sur la carte, ce calcul devrait donner une valeur de l'ordre de 50 à 70 Go/s.

Faisons le calcul pour le kernel LKCudaLKernel (calcul en chaque point). Le nombre de lectures et d'écritures effectuées dans ce kernel est égal à 813, le temps d'exécution mesuré est de 15,5 millisecondes. Par ailleurs, la résolution de l'image est 640x480, soit 307200 threads (un thread par pixel pour le calcul). On obtient alors la valeur de la bande passante par le calcul suivant :

$$Bande\ passante = \frac{Nb_{read/write} * taille_{octets} * nb_{points}}{tps_{exec} * taille_{Go}} = \frac{813 * 4 * 307200}{15{,}5.10^{-3} * 1024^3} = \mathbf{60\ Go/s}$$

La valeur trouvée est tout à fait cohérente avec la valeur attendue, cela confirme qu'il n'y a pas de problème de conception d'accès mémoire : chaque thread écrit bien dans la case de mémoire alignée qui lui est attribué.

Enfin, il est intéressant de regarder la valeur du nombre de calcul par seconde effectués par la carte. La valeur théorique pour les Tesla G80 est de 500 Gflop/s (valeur pic), mais la valeur couramment admise pour une application avec des lecture/écriture en mémoire est de 50 Gflop/s.
Le Kernel effectue 2250 opérations par thread, la valeur de la puissance de calcul est donc :

$$Puissance_{calc} = \frac{Nb_{operations} * nb_{points}}{tps_{exec} * taille_{Go}} = \frac{2250 * 307200}{15{,}5.10^{-3} * 1024^3} = \mathbf{41{,}53\ GFlop/s}$$

Cette valeur correspond tout à fait à ce qui est attendu, cela conclut la validation de l'implémentation. Il est maintenant temps d'examiner les résultats finaux.

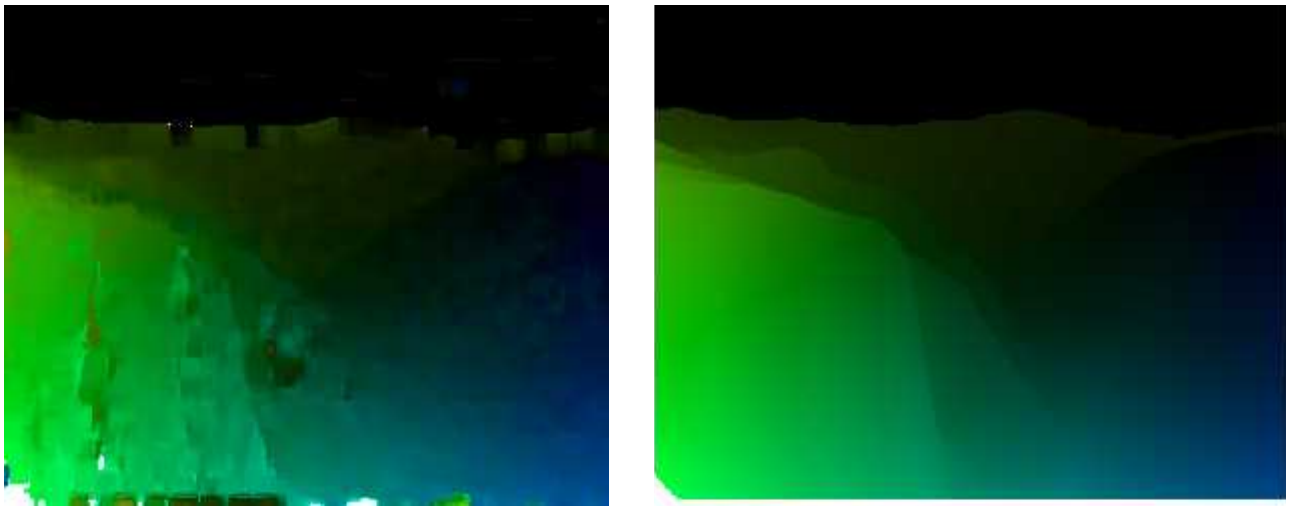
4. Résultats

Avec les mêmes paramètres que précédemment, 4 niveaux de pyramide, patch 10x10 et 3 itérations, on obtient le résultat suivant sur la séquence réelle de référence :

Flot calculé avec CUDA *Flot de référence OpenCV, mêmes paramètres*

Figure 45. Résultat final CUDA sur les images réelles

Sur la séquence-test Yosemite, on obtient :



Flot calculé avec CUDA *Flot Réel*

Figure 46. Résultat final CUDA sur Yosemite

Les résultat sont tout à fait satisfaisants et conformes au cahier des charges. Par ailleurs, le temps d'exécution observé sur la séquence 640x480 pour 4 niveaux, 3 Itérations et une taille de patch de 10x10 est de **67 millisecondes,** soit **15 Hz.** Le temps d'initialisation (allocations mémoire effectuées une seule fois) est quant à lui de 45 millisecondes.

Le challenge temps réel est donc tout à fait rempli, après passage par une phase méthodique d'optimisation du code.

Remarque : la mesure du temps s'effectue avec les fonctions de timer fournies par CUDA.

5. Extensions

(a) Utilisation de plusieurs GPU

CUDA dispose de fonctions pour gérer plusieurs cartes de même type. Il est ainsi intéressant de construire un programme permettant d'effectuer l'estimation en utilisant plusieurs cartes. Les fonctions utilisés sont toujours les mêmes, ce qui change dans le programme est le pré-traitement : il faut diviser l'image de base en autant de morceaux que de GPU, en ajoutant une bande de la taille d'un demi patch pour « recoller » ensuite les estimations.

L'utilisation de plusieurs GPU passe par la création de processus (threads) CPU, il faut donc autant de cœurs de processeurs que de GPU.

Cet aspect n'a pas pu être testé faute de temps, on peut toutefois raisonnablement espérer un gain de temps de 40% (un thread CPU nécessitant environ une milliseconde pour se lancer), soit un temps d'exécution pour les paramètres définis précédemment de 40 ms, soit 25 Hz.

(b) Production de deux estimations

Pour en revenir au Tensor Voting, vu les temps d'exécution, il est possible de construire un programme renvoyant deux estimations effectuées avec des tailles de patch différentes. Cela peut se faire soit en séquentiel, le kernel de calcul étant appelé deux fois successivement, soit de manière plus efficace, dans le même kernel en prenant garde à ne pas dépasser la mémoire autorisée.

Par ailleurs, l'utilisation de 2 GPU pour calculer les deux estimations est tentante, dans un premier temps en lançant chaque exécution séparément sur un GPU, mais cela confère du temps d'inactivité à l'un des deux GPU, en effet les deux estimations ne durent pas aussi longtemps car une taille de patch est inférieure à l'autre. Il est donc beaucoup plus intéressant de scinder l'image en 2, comme en 5.(a) et d'effectuer les deux estimations dans le même kernel.

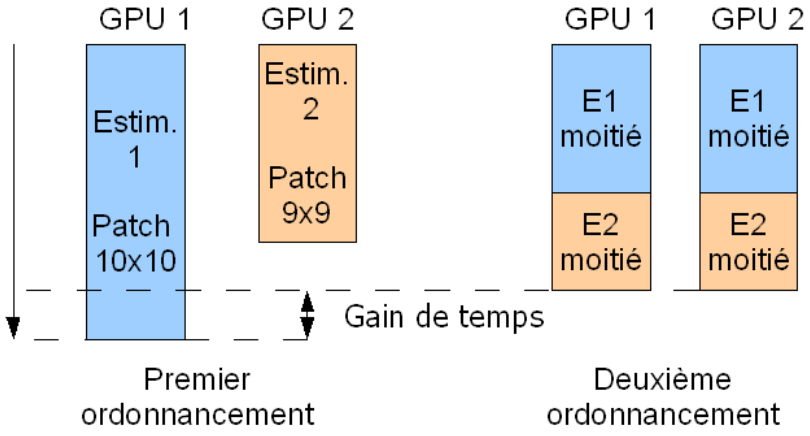


Figure 47. Deux ordonnancements possibles pour deux estimations avec deux GPU

Conclusion

Les principales difficultés rencontrées au cours de ce projet ont été de deux sortes. Tout d'abord, lors de l'étude bibliographique préliminaire, certaines publications présentaient des résultats biaisés ou peu différents d'autres papiers. Faire la synthèse a ainsi pris du temps, et il a souvent fallu tester une première implémentation Matlab pour se faire une idée de la méthode.

L'autre type de difficulté rencontrée est liée à des problèmes inhérents à la programmation, tout d'abord en C, plus exigeant que Matlab au niveau du choix des outils, puis en CUDA où il a fallu appréhender une nouvelle architecture. Toutefois la résolution des problèmes pratiques de programmation m'a permis de m'améliorer considérablement en développement.

Une fois ces difficultés surmontées, l'algorithme de calcul du flot optique a été clairement défini et implémenté sur diverses plate-formes (Matlab et langage C). L'évaluation sur les séquences synthétiques classiques a confirmé les choix effectués préalablement.

La réflexion sur la parallélisation a ensuite été menée avec méthode ainsi que l'apprentissage de l'architecture particulière de programmation sur GPU. Il était ainsi très enrichissant de pouvoir apprendre un nouveau mode de pensée et d'appréhender les problématiques de calcul parallèle, solution d'avenir pour l'optimisation du calcul scientifique (depuis la sortie de CUDA en 2007, de nombreux chercheurs programment avec cet outil [46]) .

Le projet complet de recherche (depuis l'étude bibliographique jusqu'à la production finale) que ce stage constitue a donc abouti au résultat souhaité : 15 estimations précises répondant au cahier des charges du flot optique sont fournies par seconde.

En définitive, le résultat étant convaincant et d'un intérêt scientifique certain, les résultats donneront lieu à une publication dans une conférence ainsi qu'à la mise à disposition d'un exécutable visible dans la CUDA zone du site NVIDIA [48].

--- Bibliographie ---

Généralités et comparaisons :

[1] E. Mémin, *Estimation du flot-optique : contributions et panorama de différentes approches,* habilitation à diriger des recherches de l'Université de Rennes 1, Juillet 2003

[2] B. McCane, K. Novins, D. Crannitch, B. Galvin, *On Benchmarking Optical Flow,* Computer Vision and Image understanding Volume 84, Number 1, October 2001 , pp. 126-143(18)

[3] S. S. Beauchemin, J. L. Barron, *The Computation of Optical Flow,* ACM Computing Surveys, Vol 27, No. 3, pp 433-467, September 1995

[4] C. Q.David, Z.Z. Karu, D.M. Stevenson, *Equivalence of Subpixel Motion Estimators Based on Optical Flow and Block Matching,* IEEE Proceedings of the International Symposium on Computer vision , pp. 7-12, 1995.

[5] J.L. Barron, D.J. Fleet, S.S. Beauchemin, T.A. Burkitt, *Performance of Optical Flow Techniques,* International Journal of Computer Vision (IJCV), 12(1):43-77 1994

[6] F.-B. Lauze, P. Kornprobst, C. Lenglet, R. Deriche, M. Nielsen, *Sur Quelques Méthodes de Calcul de Flot Optique à partir du Tenseur de Structure : Synthèse et Contribution,* 14ème Congrès Francophone AFRIF-AFIA de Reconnaissance des Formes et Intelligence Artificielle, 2004

Méthodes variationnelles :

Horn & Schunck

[7] B.K.P. Horn, B. G. Schunck, *Determining Optical Flow,* Artificial Intelligence, 16(1--3):185--203, August 1981.

[8] A.Bruhn, J.Weickert, *Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow* Method, IJVC(61), No. 3, February-March 2005, pp. 211-231.

Lucas & Kanade

[9] B.D. Lucas, T. Kanade, *An Iterative Image Registration Technique with an Application to Stereo Vision* In IJCAI81, pages 674--679, 1981.

[10] S. Baker, I. Matthews, *Lucas-Kanade 20 Years On: A Unifying Framework*, tech. report CMU-RI-TR-02-16, Robotics Institute, Carnegie Mellon University, July, 2002.

[11] T. Amiaza, E. Lubetzkyb, N. Kiryatia, *Coarse to over-fine optical flow estimation*, PR(40), No. 9, September 2007, pp. 2496-2503.

[12] J.-Y. Bouguet, *Pyramidal Implementation of the Lucas Kanade Feature Tracker*, Intel Corporation, Microprocessor Research Labs (2000)

Weickert et al. (Formalisation)

[13] J. Weickert, C. Schnörr, *Variational Optic Flow Computation with a Spatio-Temporal*

Smoothness Constraint, Journal of Mathematical Imaging and Vision, v.14, n.3, pp. 245-255, May 2001.

[14] T. Brox, A. Bruhn, N. Papenberg, J.Weickert, *High Accuracy Optical Flow Estimation Based on a Theory for Warping*, ECCV 2004, LNCS 3024, pp. 25–36, 2004.

Modèles paramétriques

[15] A. Verri, F. Girosi, V. Torre, *Mathematical Properties of the 2D Motion Field:from Singular Points to Motion Parameters,* J. Opt. Soc. Am. A Vol 6, 698-712 (1989)

[16] T. Nir, A.M. Bruckstein, R. Kimmel, *Over-Parameterized Variational Optical Flow* In B.C. Lovell and A.J. Maeder, Proceedings Workshop on Digital Image Computing, pages 135–139, 2003.

Couleur

[17] J.Barron, R.Klette, *Quantitative Color Optical Flow*, In 16th International Conference on Pattern Recognition, volume 4, pages 251--255, 2002.

[18] Y. Mileva, A. Bruhn, J. Weickert, Illumination-Robust Variational Optical Flow with Photometric Invariants, DAGM 2007, LNCS 4713, pp. 152–162, 2007.

[19] R.J. Andrews, B.C. Lovell, *Color Optical Flow* In Eds. Proceedings Workshop on Digital Image Computing, Brisbane, 2003

[20] P. Golland and A. M. Bruckstein, *Motion from Color,* Computer vision and image understanding Vol. 68, No. 3, December, pp. 346–362, 1997

[21] K. R. T. Aires, A. M. Santana, A. D. Medeiros, *Optical flow using color information : preliminary results*

Autres

[22] M.J Black, P. Anandan, *Robust Dynamic Motion Estimation Over Time,* In Proc. 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, IEEE Computer Society Press: Maui, HI, 1991, 292-302.

[23] A. Bab-Hadiashar, D.Suter, *Robust Optic Flow Computation,* International Journal of Computer Vision 29(1), 59–77, 1998.

Block Matching

[24] W. Li and E. Salari, *Successive Elimination Algorithm for Motion Estimation,* IEEE Transactions on image processing Vol 4, No. 1, january 1995

[25] X. Jing, L.P Chaud, *An Efficient Three-Step Search Algorithm for Block Motion Estimation,* IEEE Transactions on multimedia, Vol 6, No. 3, june 2004

[26] Y. Nie, K.K. Ma, *Adaptive Rood Pattern Search for Fast Block-Matching Motion Estimation,* IEEE Transactions on image processing , Vol. 11, No. 12, december 2002

[27] F. Essannouni, R.O.H. Thami, A. Salam, D. Aboutajdine, *An efficient fast full search block matching algorithm using FFT algorithms,* IJCSNS International Journal of Computer Science and Network Security, Vol .6 No.3B, March 2006

[28] Y.-S. Chen, Y.-P. Hung, C.S. Fuh, *Fast Block Matching Algorithm Based on the Winner-Update Strategie,* IEEE Transactions on image processing, Vol. 10, No. 8, august 2001

[29] Aroh Barjatya, *'Block matching algorithms for motion estimation,'* Tech. Rep., Utah State University, April 2004

[30] J.J. Little, A.Verri, *Analysis of Differential and Matching Methods for Optical flow,* Proceedings of workshop on Visual Motion, ISBN: 0-8186-1903-1, Mar. 1989.

Méthodes fréquentielles

[31] J.R. Bergen, P. Anandan, K.J. Hanna, and R.Hingorani, *Hierarchical Model-Based Motion Estimation,* ECCV 1992: 237-252.

[32] A. Spinéi, D. Pellerin, J. Hérault, *Estimation rapide de vitesse à base de triades de filtres de Gabor,* 16ème colloque GRETSI, pp 905-908, 15-19 septembre 1997.

[33] D.J. Fleet, *Disparity from Local Weighted Phase- Correlat ion,* IEEE International Conference on Systems, Man and Cybernetics, 1, 48-54, 1994.

[34] D.J. Fleet, A.D. Jepson, *Computation of Component Image Velocity from Local Phase Information,* IJCV(5:1), pp 77-104, 1990.

[35] V. Argyriou and T. Vlachos, *A study of sub-pixel motion estimation using phase correlation,* Centre for Vision, Speech and Signal Processing University of Surrey, 2004

Ondelettes

[36] C. Bernard *Ondelettes et problèmes mal posés : la mesure du flot optique et l'interpolation irrégulière, Thèse* Centre de Mathématiques Appliquées, Ecole Polytechnique, 1999

[37] Y.T Wu,T.Kanade,J.Cohn, C-C. Li, *Optical Flow Estimation Using Wavelet Motion Model,* ICCV 98 pp 992-998, 1998

[38] J.Magarey, N. Kingsbury, *Motion Estimation Using a Complex-Valued Wavelet Transform,* IEEE Transactions on signal processing, vol 46, no 4, April 1998.

Tensor Voting

[39] G. Medioni, M.S. Lee, C.K. Tang. *A Computational Framework for Segmentation and Grouping,* Elsevier Science, ISBN-13: 978-0444503534, 2000

[40] P. Mordohai, G.Medioni, Tensor Voting : *A perceptual Organization Approach to compute vision and machine learning,* Morgan & Claypool Publishers, ISBN:1598291009, 2007

[41] Y. Dumortier, I. Herlin, A. Ducrot, *4-D Tensor Voting Motion Segmentation for Obstacle Detection in Autonomous Guided Vehicle,* IEEE Intelligent Vehicle Symposium Eindhoven, 4-

6 juin 2008

GPU, CUDA

[42] J. Nickolls, I. Buck, M. Garland (Nvidia) et K.Skadron (University of Virginia), *Scalable Parallel Programming,* ACM QUEUE Mars/avril 2008

[43] Tom R. Halfhill, *Parallel Processing With CUDA*, Microprocessor Report, 28 janvier 2008

[44] *NVIDIA CUDA (Compute Unified Device Architecture) Programming guide*, NVIDIA, juin 2008.

[45] *CUDA Technical Training Vol. 1 et 2*, NVIDIA, 2008

[46] Y. Mizukami and K. Tadamura*, Optical Flow Computation on Compute Unified Device Architecture, 14th* ICIAP, 2007

[47] W. Stallings, Computer Organization and Architecture: Designing for Performance, Ed. Pearson, ISBN-13: 9780131856448, 2005

[48] Obtenir CUDA : http://www.nvidia.com/cuda

Compléments scientifiques :

[49] A. Richard, *Traitement statistique du signal*, cours de 3ème année ENSEM 2007-2008

[50] S.J Sangwine, R.E.N Horne et al., *The colour image processing handbook,* Ed. Chapman & Hall, ISBN : 0412806207, 1998

[51] C.Q. Davis, Z.Z Karul, D.M Freeman, *Equivalence of subpixel Motion Estimators based on optical flow and block matching,* iscv, p. 7, International Symposium on Computer Vision, 1995

ANNEXES

I - Table des figures

II - code Matlab colormap

```
function RGB = showmap(u,v,vmax)

anglestep = pi/3;
colorstep=1/anglestep;

RGBpart=[   0   1   255 255 -1  0;
            -1  0   0  1    255    255;
            255 255 -1  0   0   1];

RGB=zeros(size(u,1),size(u,2),3);

for i=1:size(u,1)
    for j=1:size(u,2)

    angle=atan2(u(i,j),v(i,j));

    if  (angle < 0)
        angle = angle+2*pi;
    end

    part=angle/anglestep;
    if ( part >= 6.0 )
        part = 0.0;
    end

    length=sqrt( u(i,j)*u(i,j) + v(i,j)*v(i,j) );

        if(length>vmax)
            RGB(i,j,1)=1;
            RGB(i,j,2)=1;
            RGB(i,j,3)=1;
        else

            for k=1:3

                switch RGBpart(k,fix(part+1))
                    case 0
                        RGB(i,j,k)=0;

                    case 1
                        RGB(i,j,k)=colorstep*mod(part,1)*length/vmax;

                    case -1
                        RGB(i,j,k)=(1-colorstep*mod(part,1))*length/vmax;

                    case 255
                        RGB(i,j,k)=length/vmax;

                end

            end

        end

    end
end
end
```

III – code Matlab final de l'algorithme

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%LUCAS KANADE PYRAMIDAL + RAFFINEMENT ITERATIF + INTERPOLATION
%J.Marzat - code final juillet 2008
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Acquisition des donnees
im1=single(rgb2gray(imread('obs0059.png')));
im2=single(rgb2gray(imread('obs0060.png')));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%parametres : nombre de niveaux, taille du patch, nombre d'iterations,
%regularisation
numLevels=4;
window=9;
iterations=1;
alpha = 0.000001;
hw = floor(window/2);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%creation des pyramides
pyramid1 = im1;
pyramid2 = im2;
%calcul et stockage de tous les niveaux
for i=2:numLevels
    im1 = impyramid(im1, 'reduce');
    im2 = impyramid(im2, 'reduce');
    pyramid1(1:size(im1,1), 1:size(im1,2), i) = im1;
    pyramid2(1:size(im2,1), 1:size(im2,2), i) = im2;
end;
%initialisation des patchs temporaires
Fx2=zeros(window);
Fy2=zeros(window);
Ft2=zeros(window);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcul à tous les niveaux
for p = 1:numLevels

    %extraction de la pyramide courante
    im1 = pyramid1(1:(size(pyramid1,1)/(2^(numLevels - p))), 1:
(size(pyramid1,2)/(2^(numLevels - p))), (numLevels - p)+1);
    im2 = pyramid2(1:(size(pyramid2,1)/(2^(numLevels - p))), 1:
(size(pyramid2,2)/(2^(numLevels - p))), (numLevels - p)+1);

    %lors de la premiere iteration au niveau le plus haut, on initialise a 0 le
champ de vitesse
    if p==1
    u=zeros(size(im1));
    v=zeros(size(im1));
    else
    %redimensionnement et interpolation du champ de vitesse
    u = 2 * imresize(u,size(u)*2,'bilinear');
    v = 2 * imresize(v,size(v)*2,'bilinear');

    end

    %calcul des dérivées pour le niveau courant
    fx1 = conv2(im1, 0.25* [-1 1; -1 1],'same');
    fx2 = conv2(im2, 0.25*[-1 1; -1 1],'same');
    fy1 = conv2(im1, 0.25* [-1 -1; 1 1],'same');
    fy2 = conv2(im2, 0.25*[-1 -1; 1 1],'same');
    ft1 = conv2(im1, 0.25*ones(2),'same');
    ft2 = conv2(im2, -0.25*ones(2),'same');
```

```
    %boucle de raffinement
    for r = 1:iterations

    %boucle sur tous les pixels
        for i = 1+hw:size(im1,1)-hw
            for j = 1+hw:size(im2,2)-hw

            %calcul de l'indice du patch deplace et des parties decimales
            lr = i-hw+v(i,j);
            hr = i+hw+v(i,j);
            lc = j-hw+u(i,j);
            hc = j+hw+u(i,j);

            px= lr-floor(lr);
            py= lc-floor(lc);

                  if (lr <= 1)||(hr >= size(im1,1))||(lc <= 1)||(hc >=
size(im1,2))
                  %on ne fait rien, c'est a dire qu'on garde la valeur
                  else

                  %interpolation : calcul des patchs derives dans l'image 2
                  for a=1:window
                  for b=1:window

                  Fx2(a,b)= (1-px)*(1-py)*fx2(floor(lr)+a-1,floor(lc)+b-1) +
px*(1-py)*fx2(floor(lr)+a,floor(lc)+b-1) + (1-px)*py*fx2(floor(lr)+a-1,floor(lc)
+b) + px*py*fx2(floor(lr)+a,floor(lc)+b);

                  Fy2(a,b)= (1-px)*(1-py)*fy2(floor(lr)+a-1,floor(lc)+b-1) +
px*(1-py)*fy2(floor(lr)+a,floor(lc)+b-1) + (1-px)*py*fy2(floor(lr)+a-1,floor(lc)
+b) + px*py*fy2(floor(lr)+a,floor(lc)+b);

                  Ft2(a,b)= (1-px)*(1-py)*ft2(floor(lr)+a-1,floor(lc)+b-1) +
px*(1-py)*ft2(floor(lr)+a,floor(lc)+b-1) + (1-px)*py*ft2(floor(lr)+a-1,floor(lc)
+b) + px*py*ft2(floor(lr)+a,floor(lc)+b);

                  end
                  end

                  %calcul des derivees courantes
                  Fx = fx1(i-hw:i+hw,j-hw:j+hw) + Fx2;
                  Fy = fy1(i-hw:i+hw,j-hw:j+hw) + Fy2;
                  Ft = ft1(i-hw:i+hw,j-hw:j+hw) + Ft2;

                  %calcul par moindres carres
                  A = [Fx(:) Fy(:)];
                  G=A'*A;
                  %regularisation
                  G(1,1)=G(1,1)+alpha; G(2,2)=G(2,2)+alpha;
                  U=1/(G(1,1)*G(2,2)-G(1,2)*G(2,1))*[G(2,2) -G(1,2);-G(2,1)
G(1,1)]*A'*-Ft(:);
                  u(i,j)=u(i,j)+U(1); v(i,j)=v(i,j)+U(2);

                  end
            end
        end
    end
end
```

IV – code C

Fonction de calcul du flot :

```
void LKPR(IplImage* img1,IplImage* img2, int numLevels, int window, int iterations, float
alpha,float* vx1, float* vy1)
//images 1 et 2, nombre de niveaux de la pyramide, taille de la fenetre, nombre d'iterations,
constante de regularisation, tableaux de sortie a remplir, de la taille de l'image
{
	//calculs preliminaires
	int hw = (int)(window/2);
	const int width = img1->width;
	const int height = img1->height;

	//allocation pour la creation des pyramides
	IplImage** buffer = (IplImage **)malloc( 2 * (numLevels) * sizeof(IplImage*));
	IplImage** imgI = (IplImage **)buffer;
	IplImage** imgJ = imgI+numLevels;

	//tableau pour stocker les champs de vecteur aux differents niveaux
	float** buffer1 = (float **)malloc( 2 * (numLevels) * sizeof(float*));
	float** vx = (float **)buffer1;
	float** vy = vx+numLevels;

	//initialisation
	imgI[0] = img1;
	imgJ[0] = img2;
	CvSize levelSize = cvSize(width,height);
	vx[0] = new float[width*height];
	vy[0] = new float[width*height];

	for (int i=1;i<numLevels;i++)
	{
		levelSize.width = (levelSize.width+1)/2;
		levelSize.height = (levelSize.height+1)/2;
		//niveau de la pyramide
		imgI[i] = cvCreateImage( levelSize, 8, 1 );
		imgJ[i] = cvCreateImage( levelSize, 8, 1 );
		cvPyrDown(imgI[i-1],imgI[i]);
		cvPyrDown(imgJ[i-1],imgJ[i]);

		//champs de vitesses
		vx[i] = new float[levelSize.width*levelSize.height];
		vy[i] = new float[levelSize.width*levelSize.height];

		//initialisation à zero
		for(int ik=0;ik<levelSize.width*levelSize.height;ik++)
	    {
			vx[i][ik]=0;
			vy[i][ik]=0;
		}
	}

	//declaration des variables intermediaires utilisees ulterieurement
	int lr = 0;int hr = 0;int lc = 0;int hc = 0;float vxi= 0;float vyi= 0;
	uchar *patch1=new uchar[window*window]; uchar *patch2=new uchar[window*window];
	float Dinv=0;float ux=0;float uy=0;int w = levelSize.width;int h = levelSize.height;
	float G11=0;float G12=0;float G22=0;float bx=0;float by=0;float It=0;float Ix=0;float Iy=0;
	float vtempx;float vtempy;

	//boucle globale, du plus haut niveau au plus bas
	for (int l=numLevels-1; l>=0; l--)
	{
		int size = h*w;

		//interpolation du champ précédent, sauf à la première itération
		if (l<numLevels-1)
		{

		//interpolation bilineaire

		for (int ib =1; ib<(h+1)/2-1;ib++)
			for(int jb = 1;jb<(w+1)/2-1;jb++)
			{
vx[l][(2*ib)*w + 2*jb]		=	2*(vx[l+1][(w/2)*(ib) + jb] + vx[l+1][(w/2)*(ib-1) + jb] +
vx[l+1][(w/2)*(ib) + jb-1] + vx[l+1][(w/2)*(ib-1) + jb-1])/4;
```

```
vx[l][(2*ib+1)*w + 2*jb]      =       2*(vx[l+1][(w/2)*(ib) + jb] + vx[l+1][(w/2)*(ib) + jb-1] +
                                      vx[l+1][(w/2)*(ib+1) + jb+1] + vx[l+1][(w/2)*(ib+1) + jb])/4;
vx[l][(2*ib)*w + 2*jb+1]      =       2*(vx[l+1][(w/2)*(ib) + jb] + vx[l+1][(w/2)*(ib-1) + jb] +
                                      vx[l+1][(w/2)*(ib) + jb+1] + vx[l+1][(w/2)*(ib-1) + jb+1])/4;
vx[l][(2*ib+1)*w + 2*jb+1]    =       2*(vx[l+1][(w/2)*(ib) + jb] + vx[l+1][(w/2)*(ib+1) + jb] +
                                      vx[l+1][(w/2)*(ib) + jb+1] + vx[l+1][(w/2)*(ib+1) + jb+1])/4;

vy[l][(2*ib)*w + 2*jb]        =       2*(vy[l+1][(w/2)*(ib) + jb] + vy[l+1][(w/2)*(ib-1) + jb] +
                                      vy[l+1][(w/2)*(ib) + jb-1] + vy[l+1][(w/2)*(ib-1) + jb-1])/4;
vy[l][(2*ib+1)*w + 2*jb]      =       2*(vy[l+1][(w/2)*(ib) + jb] + vy[l+1][(w/2)*(ib) + jb-1] +
                                      vy[l+1][(w/2)*(ib+1) + jb+1] + vy[l+1][(w/2)*(ib+1) + jb])/4;
vy[l][(2*ib)*w + 2*jb+1]      =       2*(vy[l+1][(w/2)*(ib) + jb] + vy[l+1][(w/2)*(ib-1) + jb] +
                                      vy[l+1][(w/2)*(ib) + jb+1] + vy[l+1][(w/2)*(ib-1) + jb+1])/4;
vy[l][(2*ib+1)*w + 2*jb+1]    =       2*(vy[l+1][(w/2)*(ib) + jb] + vy[l+1][(w/2)*(ib+1) + jb] +
                                      vy[l+1][(w/2)*(ib) + jb+1] + vy[l+1][(w/2)*(ib+1) + jb+1])/4;

                        }}

                for (int r = 0;r<iterations;r++) //boucle de raffinement
                {
                        for (int idi = 0; idi<h; ++idi)
                                for(int idj= 0 ; idj<w; ++idj)       //boucle sur tous les pixels

                        {if ((idi>=hw)&&(idi<h-hw)&&(idj>=hw)&&(idj<w-hw))
                                vxi = (int)(vx[l][w*idi + idj]+0.5);
                                vyi = (int)(vy[l][w*idi + idj]+0.5);

                                //calcul des indices du patch déplacé
                                lr = (idi-hw+vyi);
                                hr = (idi-hw+vyi);
                                lc = (idj-hw+vxi);
                                hc = (idj-hw+vxi);

                                        if((lr>=0)&&(hr<)&&(lc>=0)&&(hc<w))
                                        {
                                                //recuperation des patchs
                                                for (int i_r=0;i_r<window;++i_r)
                                                for(int i_c=0;i_c<window;++i_c)
                                                {
patch1[window*i_r + i_c]=((imgI[l])->imageData)[imgI[l]->widthStep*(idi-hw+i_r) + (idj - hw +i_c) ];
patch2[window*i_r + i_c]=((imgJ[l])->imageData)[imgJ[l]->widthStep*(lr + i_r) + (lc + i_c)];
                                                }

                                                //reinitialisation des matrices de calcul
                                                G11=0;G12=0;G22=0;bx=0;by=0;

                                                //calcul des derivees et des éléments des matrices
                                                for (int ki = 1;ki<window;ki++)
                                                for (int kj = 1;kj<window;kj++)
                                                {

                                                //derivee temporelle
                                                It=  (float)(0.25*(patch1[kj-1 + window*(ki-1)] -
patch2[kj-1 + window*(ki-1)] + patch1[kj + window*(ki)] - patch2[kj + window*(ki)] +patch1[kj +
window*(ki-1)] - patch2[kj + window*(ki-1)]  +patch1[kj-1 + window*(ki)] - patch2[kj-1 +
window*(ki)]));
                                                //derivees spatiales
                                                Ix = (float)(0.25*(patch1[kj-1+ window*(ki-1)] +
patch2[kj-1+ window*(ki-1)] + patch1[kj-1+ window*(ki)] + patch2[kj-1+ window*(ki)] - patch1[kj+
window*(ki-1)] - patch2[kj+ window*(ki-1)] - patch1[kj+ window*(ki)] - patch2[kj+ window*(ki)]));

                                                Iy = (float)(0.25*(patch1[kj-1+ window*(ki-1)] +
patch2[kj-1+ window*(ki-1)] - patch1[kj-1+ window*(ki)] - patch2[kj-1+ window*(ki)] + patch1[kj+
window*(ki-1)] + patch2[kj+ window*(ki-1)] - patch1[kj+ window*(ki)] - patch2[kj+ window*(ki)]));

                                                //calcul des coefficients de la matrice G et du vect b
                                                G11 = G11 + Ix*Ix;
                                                G22 = G22 + Iy*Iy;
                                                G12 = G12 + Ix*Iy;
                                                bx = bx - It*Ix;
                                                by = by - It*Iy;
                                                }

                                                //regularisation
                                                G11 = G11 + alpha;
                                                G22 = G22 + alpha;
```

```
                                        //determinant et inverse du determinant
                                        Dinv = G11 * G22 - G12 * G12;
                                        Dinv = (float)(1./Dinv);

                                        //inverse matrice
                                        ux = (float)(Dinv * (G22 * bx - G12 * by));
                                        uy = (float)(Dinv * (G11 * by - G12 * bx));

                                        //mise a jour du champ de vitesse
                                        vx[l][w*idi + idj]= float(vxi) + ux;
                                        vy[l][w*idi + idj]= float(vyi) + uy;

                                    } //fin calculs pour les pixels ne sortant pas de l'image

                                } //fin calculs pour les indices hors tour
                            } //fin boucle sur tous les pixels
                        } //fin boucle de raffinement itératif


        //mise a jour des tailles pour le niveau suivant
                h=2*h;
                w=2*w;
        } //fin boucle sur le nombre de niveaux

        //remplissage des tableaux de sortie
        for (int idx=0;idx<width*height;idx++)
        {
        vx1[idx]= vx[0][idx];
        vy1[idx]= vy[0][idx];
        }
}
```

Méthode Main :

```
int main()

{       IplImage *img1 = cvLoadImage( "obs0059.png", CV_LOAD_IMAGE_GRAYSCALE);
        IplImage *img2 = cvLoadImage( "obs0060.png", CV_LOAD_IMAGE_GRAYSCALE);

        float* vx1=new float[img1->width*img1->height];
        float* vy1=new float[img1->width*img1->height];

        //parametres
        int numLevels = 4;
        int window = 10;
        int iterations = 3;
        float alpha = (float)0.001;
        //calcul du flot
        LKPR(img1,img2,numLevels,window,iterations,alpha,vx1,vy1);

        //affichage
        IplImage *result =cvCreateImage( cvSize(img1->width,img1->height), 8, 3 );

        OpticalFlow_draw(result,vx1,vy1,img1->width*img1->height,12,0);

        cvNamedWindow( "colormap",0);
        cvShowImage( "colormap",result);

        cvWaitKey();
        return 0;
}
```

V - Tensor Voting sur la méthode d'Horn & Schunck

Les résultats étant assez convaincants, étudions le résultat du Tensor Voting sur les deux meilleures estimées que l'on peut obtenir, pour 100 itérations et 3 niveaux :

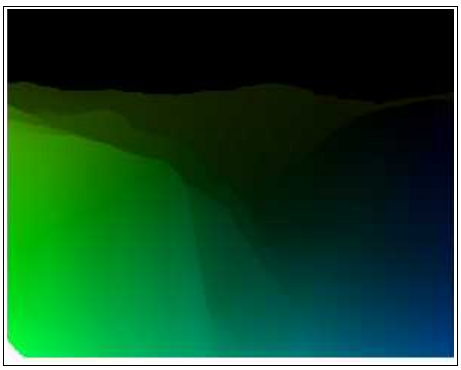
Paramètre	α = 12,5	α = 13	α = 13,5
AAE	2.4630	2.4633	2.4659
Norme	0.2877	0.2895	0.2915

Flots correspondants :

Appliquons le Tensor Voting sur ces 3 estimées, avec $\sigma^2 = 50$.

On obtient le résultat suivant :

Flot raffiné après TV



Rappel : flot réel

AAE = 2.4583°
Norm Error = 0.2901 px

(amélioration très faible)

L'inconvénient majeur de la méthode d'Horn et Schunck est que malgré le raffinement pyramidal, le tracking des grands mouvements (supérieurs à 5 ou 6 pixels) est très mauvais. La vitesse maximale dans la séquence Yosemite est de 5 pixels c'est pour cela que les résultats observés sont plutôt bons. Pour notre approche, les résultats ne seront pas suffisants (la vitesse de déplacement du véhicule en mouvement radial engendre des mouvements jusqu'à 20 pixels). Il faut également remarquer que les tests se font sur séquence synthétique (car l'on connaît ainsi le mouvement réel), et donc nuancer ces résultats puisque la méthode Horn & Schunck est très sensible au bruit.

VI – Code CUDA : Fonctions C

```
#define CB_TILE_W  16
#define CB_TILE_H  16

#define CB_TILE_W1  10
#define CB_TILE_H1  10

//objets
static cudaArray** reimgI, **reimgJ, **revx, **revy, **reDer_t1, **reDer_t2, **reDer_xI, **reDer_xJ,
** reDer_yI, **reDer_yJ;
static float* wrvx, *wrvy, *wrDer_t1, *wrDer_t2, *wrDer_xI, *wrDer_xJ, *wrDer_yI, *wrDer_yJ;
static unsigned char *wrimgI, *wrimgJ;
static size_t* pitch, *pitch_char;

//Fonction arrondi à l'entier supérieur
int iDivUp(int a, int b)
{
    return (a % b != 0) ? (a / b + 1) : (a / b);
}

//Fonction d'initialisation :allocations mémoire avant le lancement de l'algo
void
LKCudaInit( unsigned nLevels,
                  unsigned width,
                     unsigned height )
.NON DETAILLEE.

//Fonction d'appel du kernel de construction du niveau de pyramide supérieur
void
LKCudaPyrDown(unsigned level,
                       unsigned int Width,
                       unsigned int Height)
{
  dim3 grid(iDivUp(Width, CB_TILE_W1), iDivUp(Height, CB_TILE_H1));
  dim3 threads(CB_TILE_W1, CB_TILE_H1);


  // Lien entre les tableaux et les textures
  _levelTextureImg1.filterMode = cudaFilterModePoint;
  _levelTextureImg2.filterMode = cudaFilterModePoint;

  CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTextureImg1, reimgI[level-1]) );
  CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTextureImg2, reimgJ[level-1]) );

  LKCudaPyrDownKernel<<< grid , threads >>>( wrimgI, wrimgJ, pitch_char[level], Width, Height);

  CUDA_SAFE_CALL( cudaUnbindTexture(_levelTextureImg1));
  CUDA_SAFE_CALL( cudaUnbindTexture(_levelTextureImg2));
  CUDA_SAFE_CALL( cudaThreadSynchronize());
}

//Fonction de construction des pyramides
void
LKCudaPyr( unsigned char* image1,
                 unsigned char* image2,
                 unsigned width,
                 unsigned height,
                 unsigned step,
                 unsigned nLevels,
                 unsigned* w,
                 unsigned* h)
{
       //initialisation : copie dans le niveau 0 des images avec les CUDA ARRAY
       CUDA_SAFE_CALL(       cudaMemcpy2DToArray(reimgI[0], 0, 0, image1, step, width, height,
cudaMemcpyHostToDevice) );
       CUDA_SAFE_CALL(       cudaMemcpy2DToArray(reimgJ[0], 0, 0, image2, step, width, height,
cudaMemcpyHostToDevice) );

       *w = width;
       *h = height;
```

```
        //boucle de creation des pyramides
        for(unsigned i = 1; i < nLevels; i++)
        {
                *w=((*w)+1)/2;
                *h=((*h)+1)/2;

                LKCudaPyrDown(i, *w, *h);


        CUDA_SAFE_CALL(       cudaMemcpy2DToArray(reimgI[i], 0, 0, wrimgI, pitch_char[i], (*w),*h,
cudaMemcpyDeviceToDevice) );
    CUDA_SAFE_CALL(   cudaMemcpy2DToArray(reimgJ[i], 0, 0, wrimgJ, pitch_char[i], (*w),*h,
cudaMemcpyDeviceToDevice) );
        }
}

//Fonction d'appel du kernel de dérivation
void
LKCudaDeriv(unsigned level,
                  unsigned int Width,
                  unsigned int Height)
{
        dim3 grid(iDivUp(Width, CB_TILE_W1), iDivUp(Height, CB_TILE_H1));
        dim3 threads(CB_TILE_W1, CB_TILE_H1);

  // Lien entre les tableaux et les textures
  _levelTexture.filterMode = cudaFilterModePoint;
  _levelTexture2.filterMode = cudaFilterModePoint;

  CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTextureImg1, reimgI[level]));
  CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTextureImg2, reimgJ[level]));

  LKCudaDerivKernel<<< grid , threads >>>(wrDer_xI, wrDer_yI, wrDer_xJ, wrDer_yJ, wrDer_t1,
        wrDer_t2, pitch[level]/sizeof(float),Width, Height);

  CUDA_SAFE_CALL( cudaUnbindTexture(_levelTextureImg1));
  CUDA_SAFE_CALL( cudaUnbindTexture(_levelTextureImg2));
  CUDA_SAFE_CALL( cudaThreadSynchronize());

  //copie dans les cudaArray
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(reDer_xI[level], 0, 0, wrDer_xI, pitch[level],
Width*sizeof(float),Height, cudaMemcpyDeviceToDevice) );
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(reDer_yI[level], 0, 0, wrDer_yI, pitch[level],
Width*sizeof(float),Height, cudaMemcpyDeviceToDevice) );
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(reDer_xJ[level], 0, 0, wrDer_xJ, pitch[level],
Width*sizeof(float),Height, cudaMemcpyDeviceToDevice) );
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(reDer_yJ[level], 0, 0, wrDer_yJ, pitch[level],
Width*sizeof(float),Height, cudaMemcpyDeviceToDevice) );
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(reDer_t1[level], 0, 0, wrDer_t1, pitch[level],
Width*sizeof(float),Height, cudaMemcpyDeviceToDevice) );
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(reDer_t2[level], 0, 0, wrDer_t2, pitch[level],
Width*sizeof(float),Height, cudaMemcpyDeviceToDevice) );

}

//Fonction d'appel du kernel d'interpolation du champ du niveau supérieur
void
LKCudaInterDown(unsigned level,
                            unsigned int Width,
                            unsigned int Height)
{
  dim3 grid(iDivUp(Width, CB_TILE_W1), iDivUp(Height, CB_TILE_H1));
  dim3 threads(CB_TILE_W1, CB_TILE_H1);

  // Lien entre les tableaux et les textures
  _levelTexture.filterMode = cudaFilterModePoint;
  _levelTexture2.filterMode = cudaFilterModePoint;

  CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture, revx[level+1]) );
  CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture2, revy[level+1]) );

  LKCudaInterpolKernel<<< grid , threads >>>(wrvx,wrvy, pitch[level]/sizeof(float),Width, Height);

  CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture));
  CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture2));
  CUDA_SAFE_CALL( cudaThreadSynchronize());
```

```
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(revx[level], 0, 0, wrvx, pitch[level],
Width*sizeof(float), Height, cudaMemcpyDeviceToDevice) );
  CUDA_SAFE_CALL(     cudaMemcpy2DToArray(revy[level], 0, 0, wrvy, pitch[level],
Width*sizeof(float), Height, cudaMemcpyDeviceToDevice) );
}


//Fonction d'appel du kernel de calcul
void
LKCudaLK ( unsigned int window,
                  unsigned iter,
                  float alpha,
                  unsigned level,
                  unsigned int Width,
                  unsigned int Height)
{
        static unsigned hw = window / 2;

        dim3 grid(iDivUp(Width, CB_TILE_W), iDivUp(Height, CB_TILE_H));
        dim3 threads(CB_TILE_W, CB_TILE_H);

        // Lien entre les tableaux et les textures
        _levelTexture.filterMode =  cudaFilterModeLinear;
        _levelTexture2.filterMode = cudaFilterModeLinear;
        _levelTexture3.filterMode = cudaFilterModeLinear;
        _levelTexture4.filterMode = cudaFilterModeLinear;
        _levelTexture5.filterMode = cudaFilterModeLinear;
        _levelTexture6.filterMode = cudaFilterModeLinear;
        _levelTexture7.filterMode = cudaFilterModeLinear;
        _levelTexture8.filterMode = cudaFilterModeLinear;

        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture, reDer_xI[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture2, reDer_yI[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture3, reDer_xJ[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture4, reDer_yJ[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture5, reDer_t1[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture6, reDer_t2[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture7, revx[level]));
        CUDA_SAFE_CALL( cudaBindTextureToArray(_levelTexture8, revy[level]));

        LKCudaLKernel<<< grid , threads >>>(window,iter,hw,alpha,wrvx,wrvy,
pitch[level]/sizeof(float),Width,Height);

        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture2));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture3));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture4));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture5));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture6));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture7));
        CUDA_SAFE_CALL( cudaUnbindTexture(_levelTexture8));
        CUDA_SAFE_CALL( cudaThreadSynchronize());

        //copie
        CUDA_SAFE_CALL(       cudaMemcpy2DToArray(revx[level], 0, 0, wrvx, pitch[level],
Width*sizeof(float), Height, cudaMemcpyDeviceToDevice));
        CUDA_SAFE_CALL(       cudaMemcpy2DToArray(revy[level], 0, 0, wrvy, pitch[level],
Width*sizeof(float), Height, cudaMemcpyDeviceToDevice));

}

//Fonction d'appel du kernel de remise à zéro des vitesses
void
LKCudaReset(unsigned nLevels,
                      unsigned int w,
                      unsigned int h)
{
        dim3 grid(iDivUp( w, CB_TILE_W), iDivUp( h, CB_TILE_H));
        dim3 threads(CB_TILE_W, CB_TILE_H);

        LKCudaResetKernel<<< grid , threads >>>(wrvx,wrvy,pitch[nLevels-1]/sizeof(float),w, h);

        CUDA_SAFE_CALL(       cudaMemcpy2DToArray(revx[nLevels-1], 0, 0, wrvx,pitch[nLevels-
1],w*sizeof(float),h,cudaMemcpyDeviceToDevice) );
        CUDA_SAFE_CALL(       cudaMemcpy2DToArray(revy[nLevels-1], 0, 0, wrvy,pitch[nLevels-
1],w*sizeof(float),h,cudaMemcpyDeviceToDevice) );
}
```

```
//Fonction principale
void
LKCudaRun( unsigned char* image1,
                  unsigned char* image2,
                  unsigned width,
                  unsigned height,
                  unsigned step,
                  float* vx1,
                  float* vy1,
                  unsigned numLevels,
                  unsigned iter,
                  unsigned win,
                  float alpha)
{
        unsigned w;
        unsigned h;

        LKCudaPyr( image1, image2, width, height, step, numLevels, &w,&h);

        LKCudaReset(numLevels, w, h);

        for ( int l=numLevels-1; l>=0; --l )
        {
                if (l<numLevels-1)
                {       //interpolation du champ précédent et multiplication par deux
                        LKCudaInterDown( l, w, h );
                }

                LKCudaDeriv( l, w, h);

                //Calcul pour tous les pixels avec raffinement iteratif
                LKCudaLK( win, iter, alpha, l, w, h);

                //mise a jour des tailles
                w*=2;
                h*=2;
        }

        //sortie vers CPU (seulement pour les tests)

CUDA_SAFE_CALL( cudaMemcpy2DFromArray(vx1,step*sizeof(float),revx[0],0,0,width*sizeof(float),height,
cudaMemcpyDeviceToHost));

CUDA_SAFE_CALL( cudaMemcpy2DFromArray(vy1,step*sizeof(float),revy[0],0,0,width*sizeof(float),height,
cudaMemcpyDeviceToHost));
}
```

VII – Code CUDA : kernels

```
//définition des textures
static texture<unsigned char, 2, cudaReadModeElementType> _levelTextureImg1;
static texture<unsigned char, 2, cudaReadModeElementType> _levelTextureImg2;
static texture<float, 2> _levelTexture;
static texture<float, 2> _levelTexture2;
static texture<float, 2> _levelTexture3;
static texture<float, 2> _levelTexture4;
static texture<float, 2> _levelTexture5;
static texture<float, 2> _levelTexture6;
static texture<float, 2> _levelTexture7;
static texture<float, 2> _levelTexture8;

#define KERNEL_SIZE 3
#define HALF_KERNEL 1
#define NORM_FACTOR 0.028 // 1.0/(6.0^2)

//
// Gaussian 3 x 3 kernel = [1, 4, 1]/6
//

//.................. PYRAMIDES      ..................//

__global__
void
LKCudaPyrDownKernel(unsigned char *imgI,
                                    unsigned char *imgJ,
                                    size_t pitch,
                                    unsigned int downWidth,
                                    unsigned int downHeight)
{
   // calculate normalized texture coordinates
  unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

  if(x < downWidth && y < downHeight) {
    float buf[KERNEL_SIZE];

    unsigned u0 = (2 * x) - HALF_KERNEL;
    unsigned v0 = (2 * y) - HALF_KERNEL;

    for(int i = 0; i < KERNEL_SIZE; i++) {
      buf[i] =
           (tex2D(_levelTextureImg1, u0 , v0 + i) + tex2D(_levelTextureImg1, u0 + 2, v0 + i)) +
       4 * tex2D(_levelTextureImg1, u0 + 1, v0 + i);
    }

    imgI[y * pitch + x] = (unsigned char)((buf[0] + buf[2] + 4*buf[1]) * NORM_FACTOR);

       for(int i = 0; i < KERNEL_SIZE; i++) {
      buf[i] =
           (tex2D(_levelTextureImg2, u0 , v0 + i) + tex2D(_levelTextureImg2, u0 + 2, v0 + i)) +
       4 * tex2D(_levelTextureImg2, u0 + 1, v0 + i);
    }

    imgJ[y * pitch + x] = (unsigned char)((buf[0] + buf[2] + 4*buf[1]) * NORM_FACTOR);
  }
}

//.................. FIN PYRAMIDES  ..................//

//.................. INTERPOLATION  ..................//
__global__ void
LKCudaInterpolKernel(float *wrvx,
                                    float *wrvy,
                                    size_t pitch,
                                    unsigned int Width,
                                    unsigned int Height)
{
   // calculate normalized texture coordinates
       unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
       unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

       if(x < Width && y < Height)
```

```
        {
                wrvx[y*pitch + x]= 0.5*(tex2D(_levelTexture,(int)((x+1)/2),(int)((y+1)/2)) +
tex2D(_levelTexture,(int)((x+1)/2)+1,(int)((y+1)/2)+1) + tex2D(_levelTexture,(int)((x+1)/2)+1,(int)
((y+1)/2)) + tex2D(_levelTexture,(int)((x+1)/2),(int)((y+1)/2)+1));
                wrvy[y*pitch + x]= 0.5*(tex2D(_levelTexture2,(int)((x+1)/2),(int)((y+1)/2)) +
tex2D(_levelTexture2,(int)((x+1)/2)+1,(int)((y+1)/2)+1) + tex2D(_levelTexture2,(int)((x+1)/2)+1,
(int)((y+1)/2)) + tex2D(_levelTexture2,(int)((x+1)/2),(int)((y+1)/2)+1));
        }
}

//.................... FIN INTERPOLATION .............//

//.................... DERIVEES       ...................//

__global__ void
LKCudaDerivKernel(float* Derx1,
                                float* Dery1,
                                float* Derx2,
                                float* Dery2,
                                float* Dert1,
                                float* Dert2,
                                size_t Pitch,
                                unsigned int Width,
                                unsigned int Height)
{
   // calculate normalized texture coordinates
  unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

 if(x < Width && y < Height) {
  Derx1[y*Pitch + x]  =       (tex2D(_levelTextureImg1,x,y) + tex2D(_levelTextureImg1,x,y+1) -
tex2D(_levelTextureImg1,x+1,y) - tex2D(_levelTextureImg1,x+1,y+1))*0.25 ;
  Dery1[y*Pitch + x]  =       (tex2D(_levelTextureImg1,x,y) - tex2D(_levelTextureImg1,x,y+1) +
tex2D(_levelTextureImg1,x+1,y) - tex2D(_levelTextureImg1,x+1,y+1))*0.25 ;
  Derx2[y*Pitch + x]  =       (tex2D(_levelTextureImg2,x,y) + tex2D(_levelTextureImg2,x,y+1) -
tex2D(_levelTextureImg2,x+1,y) - tex2D(_levelTextureImg2,x+1,y+1))*0.25 ;
  Dery2[y*Pitch + x]  =       (tex2D(_levelTextureImg2,x,y) - tex2D(_levelTextureImg2,x,y+1) +
tex2D(_levelTextureImg2,x+1,y) - tex2D(_levelTextureImg2,x+1,y+1))*0.25 ;
  Dert1[y*Pitch + x]  =       (tex2D(_levelTextureImg1,x,y) + tex2D(_levelTextureImg1,x,y+1) +
tex2D(_levelTextureImg1,x+1,y) + tex2D(_levelTextureImg1,x+1,y+1))*0.25;
  Dert2[y*Pitch + x]  =       (- tex2D(_levelTextureImg2,x,y) - tex2D(_levelTextureImg2,x,y+1) -
tex2D(_levelTextureImg2,x+1,y) - tex2D(_levelTextureImg2,x+1,y+1))*0.25 ;
 }
}



//.................... FIN DERIVEES   ................//

//.................... INITIALISATION ....................//

__global__
void
LKCudaResetKernel(float* wrvx,
                              float* wrvy,
                              size_t Pitch,
                              unsigned int Width,
                              unsigned int Height)
{
   // calculate normalized texture coordinates
  unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
  if(x < Width && y < Height)
  {
        wrvx[y*Pitch +x] = 0;
        wrvy[y*Pitch +x] = 0;
  }
}

//.................... FIN INITIALISATION    ................//

//..........   CALCUL POUR CHAQUE PIXEL      ..........//
__global__
void
LKCudaLKernel(unsigned int window,
                        unsigned iter,
                        unsigned int hw,
```

```
                        float alpha,
                        float* wr_vx,
                        float* wr_vy,
                        size_t Pitch,
                        unsigned int Width,
                        unsigned int Height)
{
  // calculate normalized texture coordinates
  unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

  //if(x < Width && y < Height)

  //{
    float vxPrec = tex2D(_levelTexture7,x,y);
       float vyPrec = tex2D(_levelTexture8,x,y);

       for (int r=0 ; r<iter ; ++r)
       {
    float lr = y%Pitch - hw + vyPrec;
    float lc = x - hw + vxPrec;

       if ( (lr<0)||(lr>=Height)||(lc<0)||(lc>=Width))
       {
       //si les indices déplacés sortent de l'image,
       //on garde la valeur précédente
       }
       else
       {
    float Ix;
       float Iy;
       float It;
       float G11=.0f;
       float G22=.0f;
       float G12=.0f;
       float bx=.0f;
       float by=.0f;

       //calcul des valeurs des matrices A et b
              for(int i=0;i<window;++i)
                     for(int j=0;j<window;++j)
              {

              //interpolation automatique par la texture
              Ix = tex2D(_levelTexture,x-hw+j,y-hw+i)+tex2D(_levelTexture3,lc+j,lr+i);;
              Iy = tex2D(_levelTexture2,x-hw+j,y-hw+i)+tex2D(_levelTexture4,lc+j,lr+i);
              It = tex2D(_levelTexture5,x-hw+j,y-hw+i)+tex2D(_levelTexture6,lc+j,lr+i);

              G11 += Ix*Ix;
              G22 += Iy*Iy;
              G12 += Ix*Iy;

              bx -= (It*Ix);
              by -= (It*Iy);
              }

              G11 += alpha;
              G22 += alpha;

       //determinant et inverse du determinant
       //on reutilise Ix qui ne sert plus pour stocker le determinant.
       Ix = (float)(1./(G11 * G22 - G12 * G12));

       //remplissage des vitesses
       vxPrec += (Ix * (G22*bx - G12*by));
       vyPrec += (Ix * (G11*by - G12*bx));
       }
       }

       //remplissage des vitesses
       wr_vx[y*Pitch + x] = vxPrec;
       wr_vy[y*Pitch + x] = vyPrec;
 // }
}

//.................. FIN CALCUL POUR CHAQUE PIXEL ..............//
```