Estimation Temps Réel du Flot Optique

Julien Marzat
Institut National Polytechnique de Lorraine
Encadrants : Didier Wolf (ENSEM), André Ducrot (INRIA)

Mots-clés : *Vision Monoculaire, Flot optique, Calcul parallèle, CUDA.*

Résumé

Le calcul du flot optique est une étape essentielle en vision car il fournit une bonne estimation du mouvement image, qui peut servir de base à des processus de segmentation. Un des principaux problèmes est la lenteur d'exécution des algorithmes existants. L'objectif de cette étude est donc le calcul du flot optique en temps réel (10 Hz minimum). Ce but est atteint en déterminant un algorithme de calcul parallélisable et en l'implémentant sur GPU (Graphics Processing Unit) à l'aide de l'environnement de développement CUDA (Compute Unified Device Architecture) dédié aux architectures parallèles NVIDIA.

1. Introduction

La perception de l'environnement est un élément essentiel pour l'automatisation des véhicules. Il est ainsi possible de réaliser un processus de détection d'obstacles en vision monoculaire (une seule caméra). Un tel processus a été développé dans le cadre de la thèse de Yann Dumortier («*Perception de l'environnement en vision monoculaire pour les véhicules autonomes*») dans lequel s'inscrit mon stage. 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, qui fournit une estimation du mouvement réel des objets dans la scène.

Il existe de nombreuses méthodes de calcul du flot optique, toutes ayant en commun de ne pas être temps réel. Or, dans l'optique d'embarquer le processus sur véhicule, il est nécessaire de disposer de plusieurs estimées du flot optique par seconde (10 minimum).

Dans cette partie est présenté le processus de segmentation ainsi que le cahier des charges qui en découle du point de vue du calcul du flot optique. La partie 2 présente un état de l'art non exhaustif des techniques d'estimation du mouvement image. La partie 3 décrit l'algorithme choisi vis-à-vis du cahier des charges, la partie 4 s'intéressant quant-à-elle à sa parallélisation et à la description de l'environnement de développement CUDA. Enfin, la partie 5 présentera les résultats obtenus en les comparant aux approches existantes.

1.1 Processus de détection d'obstacles

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. Le caractère novateur du processus réside notamment dans l'utilisation de l'outil « Tensor Voting ». Le Tensor Voting [10] 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. Il s'agit donc d'un outil puissant de segmentation. Suite à ce traitement effectué sur une ou plusieurs estimations du flot optique, la détermination du plan de la route s'effectue en estimant son homographie (asservissement visuel) par la technique RANSAC (RANdom SAmple Consensus). Les objets mobiles sont quant à eux discriminés à l'aide de contraintes de rigidité à partir de la segmentation multi-échelle du mouvement image [11].

1.2 Cahier des charges

Pour être incorporé au processus, l'algorithme d'estimation du flot optique doit fournir une valeur du mouvement pour chaque pixel, c'est-à-dire un flot dense. L'estimation doit par ailleurs être aussi précise que possible (subpixelique) et utiliser un minimum de filtrage (perte d'information). Le calcul se fait sur deux images successives de la séquence (flux causal). Enfin, dans un souci de mise en œuvre temps réel à l'aide de GPU (processeur graphique dédié au calcul parallèle), l'algorithme doit être parallélisable**.**

2. Évaluation du flot optique

La plupart des méthodes d'estimation du flot optique reposent sur une hypothèse fondamentale : l'intensité lumineuse se conserve entre deux images successives.

Cela s'écrit sous la forme générale :

$$I(x+\omega, t+1) - I(x,t) = 0 \quad (1.1)$$

où $\omega = [ux, uy]^T$ est le déplacement recherché. Il est également possible de formuler le problème sous forme différentielle, ce qui conduit à :

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

soit l'équation du flot optique :

$$\boxed{(\nabla I)^T \omega + I_t = 0} \quad (1.2)$$

avec $\nabla I = [I_x, I_y]^T$ le gradient spatial de l'intensité lumineuse, I_t la dérivée temporelle et ω la vitesse recherchée.

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. Il faut donc émettre une hypothèse supplémentaire. Les méthodes détaillées dans la suite de cette partie proposent différentes solutions au problème.

2.1 Améliorations de l'estimation

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.

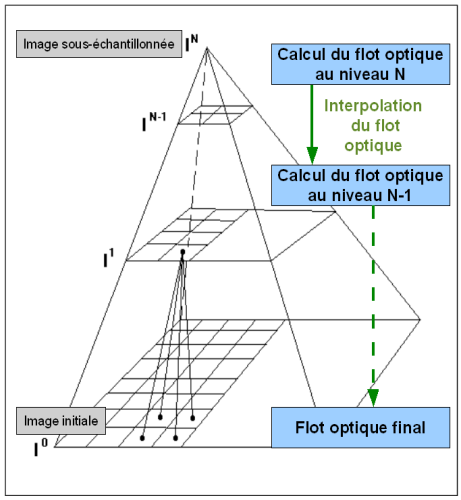


Figure 1. Implémentation pyramidale d'une méthode de calcul du flot optique

- l'implémentation pyramidale : 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 zéro correspondant à l'image initiale, le niveau Lm correspondant au niveau le plus grossier. 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 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 (voir Figure 1).

2.2 Méthodes variationnelles

Les méthodes basées sur l'approche variationnelle consistent à résoudre un problème d'optimisation (local ou global) en minimisant une fonctionnelle, généralement basée sur l'équation (1.2) à laquelle on ajoute une contrainte pour particulariser les solutions.

2.2.1 Méthodes globales

Ce type d'approche consiste à minimiser sur le domaine entier de l'image une fonctionnelle prenant en compte l'équation (1.2) 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 principale méthode globale de calcul du flot optique a été développée en 1980 par Horn & Schunck [3] et consiste à minimiser sur l'ensemble de l'image la fonctionnelle :

$$J_{HS} = \int\int ((\nabla I)^T \omega + I_t)^2 + \alpha((\nabla v_x)^2 + (\nabla v_y)^2)\,dx\,dy \quad (2.1)$$

Cette régularisation se justifie par le fait que les vitesses voisines sont presque identiques. Il existe des variantes de cette méthode utilisant d'autres opérateurs de régularisation [1]. Les résultats présentent toutefois tous les mêmes types de défauts [2] : certes le flot est dense, mais lisse et très bruité. Par ailleurs, le caractère global de l'approche ampute le résultat des petits mouvements de l'image. La conséquence de ceci est un lissage excessif du flot (grand mouvement d'ensemble observé).

2.2.2 Méthodes 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 développée par Lucas & Kanade [5] : la vitesse locale est supposée constante sur un voisinage spatial Ω, on minimise alors sur le domaine la fonctionnelle :

$$J_{LK}=\sum_{\Omega} W^2[\nabla I . \vec{\omega}+I_t]^2 \quad (2.2)$$

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). 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, notamment à l'aide d'une implémentation pyramidale [6].

2.3 Méthodes fréquentielles

Une autre classe de méthodes s'appuie sur les manipulations possibles dans le domaine fréquentiel.

La transformée de Fourier de l'équation (1.2) du flot optique nous donne : $\boxed{v_x f_x+v_y f_y+f_t=0}$ (2.3) où f_x, f_y sont les fréquences spatiales et f_t la fréquence temporelle.

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

$$g(x,y,t)=\frac{1}{(2\Pi)^3\sigma_x\sigma_y\sigma_t} e^{\frac{-x^2}{2\sigma_x^2}-\frac{y^2}{2\sigma_y^2}-\frac{t^2}{2\sigma_t^2}} . \cos(2\Pi(f_{x0}x+f_{y0}y+f_{t0}t)) \quad (2.4)$$

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 $\left(\frac{1}{\sigma_x}\frac{,1}{\sigma_y}\frac{,1}{\sigma_t}\right)$.

2.3.1 Méthodes par filtrage

Plusieurs méthodes proposent une combinaison de filtrages pour évaluer différents ordres de grandeur de vitesse, soit par une approche multi-résolution (pyramide de Heeger), soit en filtrant l'image par plusieurs filtres de Gabor orientés différemment de manière à être en présence d'un système surdéterminé pouvant être résolu immédiatement (Weber et Malik) [2]. De même, Fleet et Jepson [8] proposent l'utilisation de la phase obtenue après convolution de l'image avec une famille de filtres de Gabor. Il existe également des approches utilisant des familles d'ondelettes particulières [9] (équivalentes à des projections sur des filtres particuliers).

L'inconvénient de toutes ces méthodes est l'utilisation excessive du filtrage, qui entraîne une perte d'information préjudiciable notamment en ce qui concerne les petits mouvements. Par ailleurs, le nombre important de paramètres à régler (6 pour chaque filtre, davantage pour les familles d'ondelettes) est problématique dans l'optique d'une estimation non biaisée et la plus naturelle possible.

2.3.2 Méthode de corrélation de phase

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 :

$$\hat{f}_2(f_x,f_y)=\hat{f}_1(f_x,f_y)e^{-i(f_x v_x+f_y v_y)}$$

$$\text{puis} \quad \frac{\hat{f}_2\hat{f}_1^*}{\left|\hat{f}_2\hat{f}_1^*\right|}=e^{-i(f_x v_x+f_y v_y)} \quad (2.5)$$

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

$$c_{t,t+1}(v_x,v_y)=F^{-1}\left(\frac{\hat{f}_2\hat{f}_1^*}{\left|\hat{f}_2\hat{f}_1^*\right|}\right) \quad (2.6)$$

où F^{-1} désigne l'opérateur transformée de Fourier inverse.

Une estimée du mouvement est alors :

$$[v_x, v_y]=argmax(\Re(c_{t,t+1})) \quad (2.7)$$

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 de la requête du cahier des charges en terme d'estimées subpixeliques.

2.4 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 : corrélation simple (2.8), somme de carrés des différences (2.9) et somme des différences absolues (2.10)

$$\int_{\Omega} f_1(x+v_x, y+v_y) f_2(x, y)\, dxdy \quad (2.8)$$

$$SSD = \sum \sum (I(i, j) - J(i+u, j+v))^2 \quad (2.9)$$

$$SAD = \sum \sum |I(i, j) - J(i+u, j+v)| \quad (2.10)$$

De nombreux algorithmes d'exploration ont été développés [13], le plus simple étant l'exploration de tous les blocs de la fenêtre de recherche ("Full Search"). L'inconvénient majeur des méthodes de block matching est qu'elles postulent un déplacement maximal sur une certaine fenêtre, on exclut ainsi les grands déplacements et le résultat est biaisé, en plus d'être mosaïqué. Une solution consiste donc à utiliser une pyramide, de manière à disposer de niveaux de raffinement différents.

Le résultat (dense) 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 (on calcule des niveaux de pyramide de taille supérieure à l'image originale). Les méthodes de Block Matching sont donc à considérer avec intérêt, les résultats fournis étant denses, potentiellement subpixeliques et parallélisables car chaque recherche est indépendante des recherches voisines, ce qui permet également d'identifier les faibles mouvements.

3. Algorithme d'estimation

3.1 Approches retenues

Le cahier des charges impose comme au résultat d'être dense, le plus précis possible (subpixelique), dépourvu de paramétrisation du flot et de filtrages excessifs. Du fait de ces contraintes, les méthodes fréquentielles ne sont pas retenues. En effet, la corrélation de phase donne des pics locaux du flot et donc un résultat non dense et les méthodes par filtrage (spatiotemporel, ondelettes...) effectuent un lissage trop important et sont soumises à un réglage de paramètres conséquent. Les approches répondant au cahier des charges sont donc les méthodes variationnelles (globales et locales) et les techniques de block matching.

3.2 Tests

Figure 2. Séquence Test

Les tests sont effectués avec le logiciel Matlab, sur une séquence réelle de déplacement urbain (Figure 2). L'algorithme qui sert de référence dans le domaine du flot optique est l'implémentation pyramidale de Lucas & Kanade disponible dans la librairie OpenCV d'Intel [6].

Une carte de couleurs est utilisée pour afficher les résultats obtenus (Figure 3). La couleur représente la direction du vecteur vitesse et l'intensité sa norme.

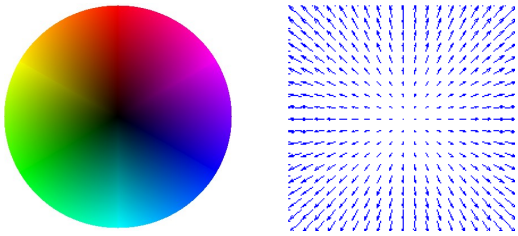
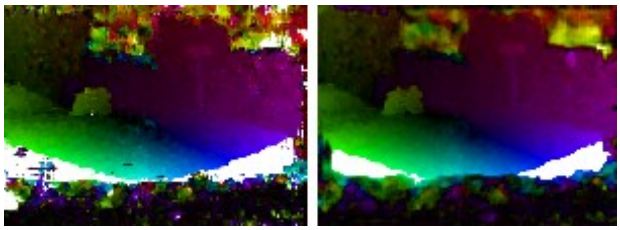
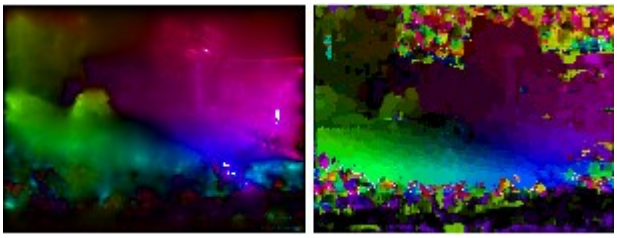


Figure 3.Carte de représentation et équivalent vecteur vitesse

En Figure 4 sont présentés les meilleurs résultats obtenus avec les trois méthodes parmi lesquelles le choix final se fait : méthode de Horn & Schunck, méthode de Lucas & Kanade et Block Matching Full Search, ces trois implémentations étant pyramidales, de manière à identifier les grands mouvements d'une part et les déplacements subpixeliques d'autre part.



Référence OpenCV — Type Lucas & Kanade



Type Horn & Schunck — Type Block Matching

Figure 4. Tests

Le résultat fourni par l'implémentation pyramidale d'Horn & Schunck est très sensible au bruit (sur la route notamment). Par ailleurs, le caractère global de la méthode lui confère un caractère très lisse, il en ressort une imprécision du résultat avec notamment une absence d'identification des mouvements locaux et une confusion entre le mouvement de la voiture à détecter et le mouvement radial de la route, ce qui est très préjudiciable dans le cadre de la segmentation. Le résultat du Block Matching (Full Search) est cohérente vis-à-vis de la référence, toutefois la méthode est assez

sensible au bruit (nécessité d'une taille de bloc assez conséquente, ce qui entraîne un flot lisse). 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...). On comprend ainsi le succès de cette méthode pour effectuer des estimations grossières rapides du flot optique mais elle se révèle assez inadaptée dans le cadre de l'étude conduite ici.

Le meilleur résultat obtenu est donc donné par l'implémentation pyramidale raffinée de Lucas & Kanade (5 niveaux de pyramide, taille de patch 10x10 et 10 itérations), tout à fait conforme voire supérieure à la référence.

3.3 Description de l'algorithme retenu

Compte tenu du cahier des charges et des résultats obtenus, le choix final de l'algorithme de calcul du flot optique est une implémentation pyramidale raffinée de la méthode de Lucas & Kanade, avec l'introduction des moindres carrés régularisés. L'algorithme est décrit plus précisément par le schéma en Figure 5.

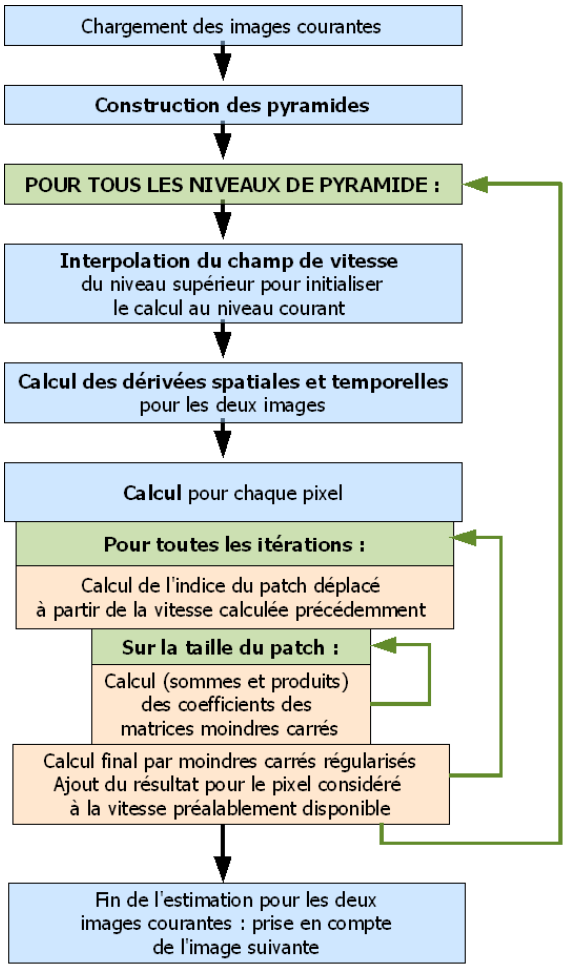


Figure 5. Algorithme d'estimation

L'estimateur des moindres carrés est assez sensible au bruit et aux erreurs de mesure. Il est possible de remédier à cela en utilisant la régularisation quadratique l_2l_2. Cet estimateur est toujours linéaire et s'exprime, tout calcul fait, de la manière suivante [14] :

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T A + \alpha I)^{-1} A^T b \quad (3.1)$$

avec α réglable, représentant la régularité de la solution (en pratique, $10^{-6} < \alpha < 10^{-3}$).

3.4 Validation et Réglages des paramètres

3.4.1 Séquence test et mesure de l'erreur

De manière à mesurer l'erreur d'estimation du flot optique calculé, il faut disposer d'une référence dont on connaît exactement le mouvement. C'est pourquoi on utilise une séquence de référence synthétique dont le flot est parfaitement connu (Figure 6). Le calcul de l'erreur s'effectue alors de la manière suivante :

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

et $Norm = \sqrt{((u_r - u_c)^2 + (v_r - v_c)^2)}$ (3.2)

où (u_c, v_c) est le flot calculé et (u_r, v_r) le flot réel.

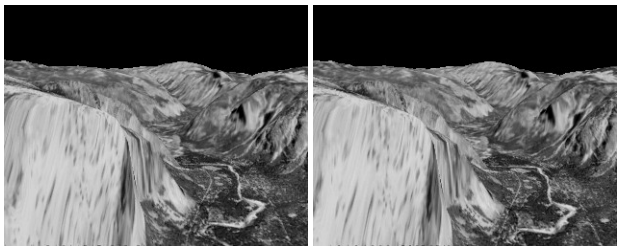


Figure 6. Séquence Yosemite

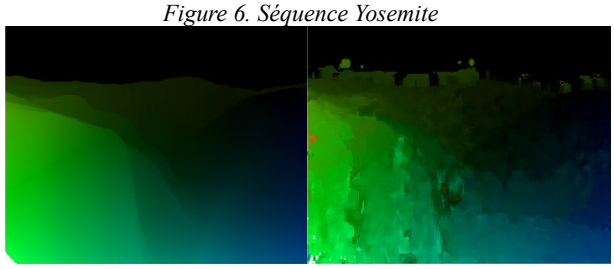


Figure 7. Flot réel et Flot calculé sur la séquence test

La figure 7 présente un exemple de flot calculé sur la séquence test avec l'algorithme construit précédemment, comparé avec le flot réel connu.

3.4.2 Réglages des paramètres

Le réglage des paramètres de la méthode (nombre de niveaux de pyramide, taille du patch sur lequel la vitesse est considérée comme constante, nombre d'itérations)

peut donc s'effectuer en choisissant l'ensemble qui minimise l'erreur. Pour l'algorithme choisi, les résultats nous indiquent d'utiliser 3 à 4 niveaux de pyramide, une taille de patch comprise entre 9x9 et 11x11 et un nombre d'itérations compris entre 2 et 4 (Figure 8).

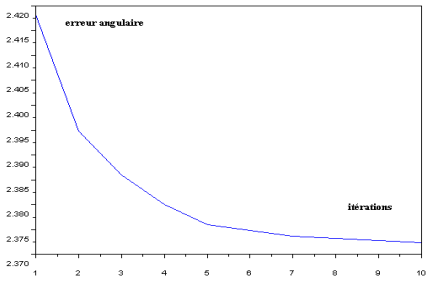


Figure 8. Erreur angulaire en fonction du nombre d'itérations

4. Implémentation sur GPU

Le temps de calcul de l'algorithme traduit en langage C, pour 4 niveaux de pyramide, une taille de patch de 10x10 et 3 itérations est de 7 secondes (30x plus rapide que le code Matlab). Il faut toutefois encore améliorer ce temps d'exécution, en ayant recours au calcul parallèle pour atteindre l'objectif initial de 10 estimées par seconde.

4.1 GPU en calcul scientifique

Historiquement, les programmes étaient écrits pour un traitement séquentiel et pour être exécutés sur une seule machine avec une seule unité de calcul (architecture SISD : Single Instruction Single Data). Le développement des approches parallèles est relativement récent, avec notamment de nombreuses tentatives d'implémentation de type GPGPU (General-Purpose computing on GPU), à savoir l'utilisation des structures existantes de processeurs graphiques (architecture SIMD: Single Instruction, Multiple Data) pour effectuer du calcul intensif d'algorithmes hautement parallèles. La démocratisation de ce type d'approche a amené NVIDIA à produire des chipsets graphiques entièrement dédiés au calcul parallèle (Figure 9) couplés à une interface et un langage de programmation, dérivés directement du C, sorti en 2007 : CUDA [12].

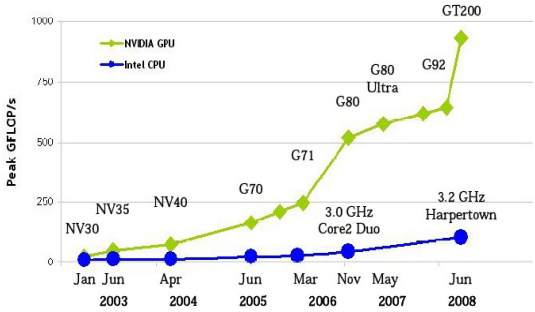


Figure 9. Evolution de la capacité de calcul des GPU NVIDIA

4.2 CUDA

Dans le cadre du stage, une carte Tesla C870 (chipset de type G80) a été utilisée. Ce GPU est constitué d'un assemblage de 128 multiprocesseurs. CUDA demande au développeur d'organiser la masse de travail à effectuer sur le GPU en un assemblage de blocs de threads. De manière à ce que l'approche soit générique (indépendante de la carte utilisée), la parallélisation est organisée selon trois niveaux (Figure 10) : chaque thread contient la même séquence d'instructions à exécuter sur des données différentes, chaque bloc de threads est exécuté sur un multiprocesseur, les blocs étant eux-mêmes contenus dans un grille de blocs de threads. Lorsque le nombre de blocs est supérieur au nombre de multiprocesseurs, les blocs sont placés en file d'attente [15].

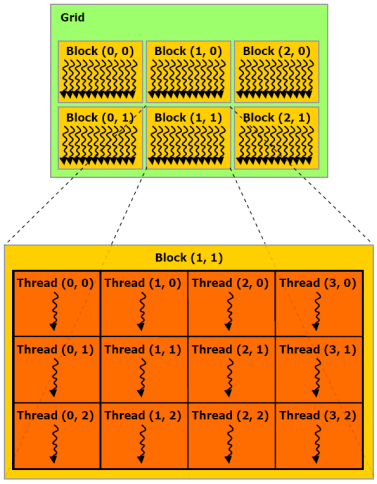


Figure 10. Trois niveaux d'abstraction

En ce qui concerne la mémoire, on trouve également trois niveaux de stockage. Chaque thread dispose de 32 registres de 4 octets. 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) de 1,5Go appelée « global memory ». Il faut également noter que les transferts de mémoire du CPU vers le GPU sont très lents tandis que les transferts en interne sur la carte sont immédiats. Il est donc beaucoup plus intéressant de conserver les données calculées au fur et à mesure du processus sur la carte.

Les cartes (*device*) sont vues comme des co-processeurs par le CPU (*host*). Les programmes sont écrits en C, exécutés sur le CPU et font appel aux GPU par le biais de CUDA. 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.

4.3 Parallélisation de l'algorithme

L'idée de la parallélisation de l'algorithme décrit en 3.3 part du 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. Il y a plus précisément quatre étapes parallélisables : la construction des pyramides, le calcul des dérivées, l'interpolation (doublement de la taille) des champs de vitesse et le calcul pour chaque pixel. La construction des pyramides est à la fois séquentielle 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 totalité de la pyramide. Le calcul des dérivées se fait à chaque changement de niveau de pyramide (dans la boucle globale) une seule fois pour chaque image. La dérivée en chaque pixel est indépendante des dérivées voisines, cela ne pose donc aucun problème de parallélisation. De la même manière, l'interpolation des vitesses du niveau supérieur ne dépend que du champ de vitesse et non des voisins calculés. Enfin, le calcul en lui-même s'effectue à l'aide des objets calculés précédemment, chaque estimation étant calculée séparément des estimations voisines. Ces quatre éléments donnent lieu à autant de kernels, chaque thread s'occupant du calcul en un pixel de l'image à partir des données stockées sur la carte.

Les parties qui restent immuablement séquentielles sont l'initialisation (allocation mémoire) des objets utilisés pour le calcul, la boucle itérative sur le nombre de niveaux de pyramide et le raffinement itératif au sein d'un même niveau (ce dernier point sera toutefois inclus dans le kernel de calcul).

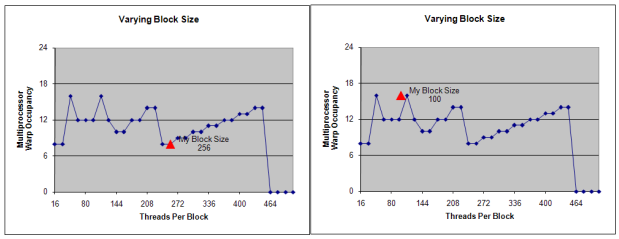
4.4 Optimisations

4.4.1 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. 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 donc finalement respecter la contrainte suivante :

$$Nb_{threads} * smem < 16\,KB \quad (4.1)$$

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. NVIDIA met à disposition un calculateur d'occupation en fonction des paramètres précédents. En figure 11 sont présentées les courbes d'occupation pour le kernel Pyramide, avant et après modification. En effet, le calculateur préconisait de réduire le nombre de threads par blocs de 256 à 100 pour maximiser l'occupation, cela s'est traduit par un gain de temps à l'éxecution de 33% pour ce kernel.



Avant modification Après modification

Figure 11. Occupation du kernel Pyramide

Il est également possible d'utiliser le profiler [16] fourni par NVIDIA pour vérifier l'enchaînement des appels de kernel lors de l'exécution du programme, de manière à repérer les anomalies.

4.4.2 Bande passante et 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)[15]. 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. Tous calculs faits, on trouve une valeur de 60 Go/s, tout à fait cohérente avec la valeur attendue.

Enfin, il est intéressant de regarder la valeur du nombre de calculs 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 lectures/écritures en mémoire est de 50 Gflop/s. Le Kernel principal effectue 2250 opérations par thread en 15.5 ms, la puissance de calcul est donc de 41,53 GFlop/s. Cette valeur correspond tout à fait à ce qui est attendu, cela conclut la validation de l'implémentation.

4.5 Résultats

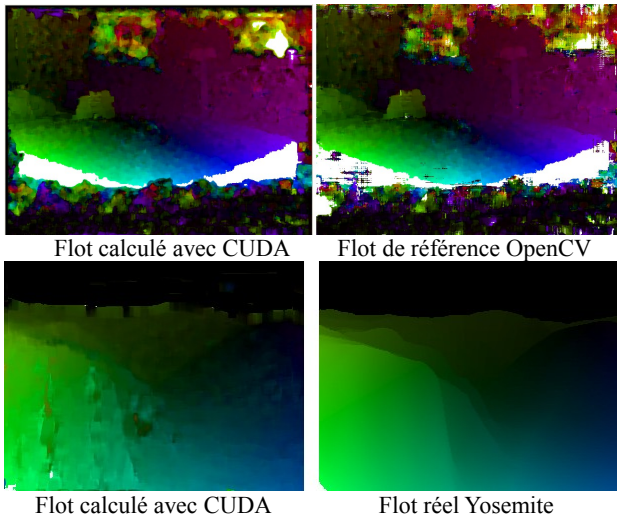


Figure 12. Résultat CUDA sur la séquence urbaine (fig 2) et sur la séquence-test Yosemite (fig 6)

Les résultat sont tout à fait satisfaisants et conformes au cahier des charges (1.2). 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 14,9 Hz. Le temps d'initialisation (allocations mémoire effectuées une seule fois lors du démarrage) est quant à lui de 45 millisecondes.

5. Conclusion

Cette étude complète, depuis l'étude bibliographique jusqu'à l'implémentation temps réel à l'aide d'un chipset graphique, a donc abouti au résultat souhaité: 15 estimations précises du flot optique répondant au cahier des charges sont fournies par seconde. L'implémentation pourra donc être aisément incorporée à tout processus de détection d'obstacles en vision monoculaire.

Il serait intéressant de tester la méthode décrite dans cet article sur plusieurs GPUs d'une part et sur une carte GT200 d'autre part (deux fois plus puissante que celle utilisée au cours du stage) de manière à obtenir un processus encore plus rapide (20 estimées par seconde) et donc encore plus aisé à intégrer dans un contexte plus large.

Références

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

[2] 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

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

[4] 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

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

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

[7] 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

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

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

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

[11] 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

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

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

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

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

[16] http://www.nvidia.com/cuda