Mise en correspondance rapide de patchs plan pour la localisation d'un robot en utilisant le GPU

Baptiste Charmette, Frédéric Chausse, Eric Royer

Mise en correspondance rapide de patchs plan pour la localisation d'un robot en utilisant le GPU

Baptiste CHARMETTE[1,3] Frédéric CHAUSSE[2,3] Éric ROYER[2,3]

[1] Clermont Université, Université Blaise Pascal, LASMEA, BP 10448, F-63000 CLERMONT-FERRAND
[2] Clermont Université, Université d'Auvergne, LASMEA, BP 10448, F-63000 CLERMONT-FERRAND
[3] CNRS, UMR 6602, LASMEA, F-63177 AUBIERE

24 avenue des landais
63177 Aubière
prénom.nom@lasmea.univ-bpclermont.fr

Résumé

L'appariement entre des amers visuels et une carte est généralement un processus assez lourd à mettre en œuvre au sein d'algorithme de localisation de robot mobile ou de SLAM. Le principal problème est de pouvoir reconnaitre des objets vus depuis des points de vue différents. Plusieurs méthodes basées sur des descripteurs de points d'intérêt tels que SIFT ont été portées sur GPU pour être utilisables dans un contexte temps-réel. Dans ce papier, nous présentons une autre façon d'apparier des amers en utilisant une modélisation 3D des points d'intérêt ainsi qu'un modèle de mouvement du robot. Cet algorithme d'appariement conçu pour la localisation de robot mobile est toutefois beaucoup trop lent pour être exécuté sur CPU. Grâce à son implémentation sur GPU, nous avons montré qu'il est possible d'atteindre des performances temps-réel tout en conservant un algorithme plus robuste que les méthodes basées sur les descripteurs.

Mots Clef

Localisation, patch-plan, robotique, GPU

Abstract

Matching image features between an image and a map of landmarks is usually a time consuming process in mobile robot localization or Simultaneous Localisation And Mapping algorithms. The main problem is being able to match features in spite of viewpoint changes. Methods based on interest point descriptors such as SIFT have been implemented on GPUs to reach real time performance. In this paper, we present another way to match features with the use of a local 3D model of the features and a motion model of the robot. This matching algorithm dedicated to robot localization would be much too slow if executed on a CPU. Thanks to a GPU implementation, we show that it is possible to achieve real-time performance while offering more robustness than descriptor based methods.

Keywords

Localization, planar feature, robotic, GPU

1 Introduction

Le déplacement autonome d'un robot mobile nécessite généralement de suivre une trajectoire prédéfinie, c'est-à-dire de corriger constamment les écarts de la position courante par rapport à cette trajectoire. Il est donc très important que le système de commande du robot connaisse cette position avec une précision suffisante pour satisfaire les contraintes de l'application. Les systèmes basés sur la vision ont l'avantage de n'utiliser qu'une simple caméra et ne se localisent que par rapport aux éléments visibles des alentours, sans nécessiter de changement d'infrastructure. De tels systèmes fonctionnent généralement en créant une carte de la zone où se situe le véhicule. Il est donc nécessaire d'apparier des amers repérés sur l'image courante avec ceux enregistrés dans la carte. La plupart du temps, cet appariement est réalisé grâce à des descripteurs tel que SIFT [11] ou SURF [2]. Néanmoins, cette partie nécessite de lourds calculs pouvant faire obstacle à une utilisation en temps-réel. Pour accélérer les calculs, de nombreux auteurs [15], [6], [14] ont implémenté les algorithmes sur carte graphique (ou GPU pour Graphical Processing Unit).

D'autres méthodes d'appariement basées sur le modèle 3D des amers ont été proposées et se sont montrées plus robustes dans des applications de type SLAM (Simultaneous Localisation And Mapping : localisation et reconstruction simultanée) ou de robotique [12], [3], [5]. Cependant, tout comme les descripteurs cités précédemment, ces méthodes nécessitent de lourds calculs et ont besoin d'être accélérées pour être utilisées en temps réel avec de nombreux points. dans cet article, nous allons montrer comment implémenter en GPU notre méthode déjà publiée en [5]. Celle ci sera rappelée dans la section 2

1.1 Algorithme de localisation

Nous supposons qu'une carte d'amers visuels a été précédemment construite en utilisant un algorithme de type SLAM. L'objectif est donc d'apparier les amers de la carte avec les points d'intérêt de l'image courante pour déterminer la position du robot. La principale difficulté consiste à apparier les points observés depuis des points de vue parfois très différents.

De nombreux travaux utilisent des descripteurs invariants au point de vue. Ainsi chaque point détecté est transformé en un vecteur de valeurs qui reste quasiment le même quel que soit le point de vue. En appliquant cette transformation à chaque point détecté, une comparaison des valeurs obtenues fournit un score de ressemblance avec les points enregistrés, et un simple seuillage permet de déterminer s'ils sont identiques ou non. Les descripteurs classiques tels le SIFT [11] ou le SURF [2] utilisent l'information d'une seule image pour générer le vecteur de données qui est robuste aux changements de point de vue. D'autre travaux [4], [10] utilisent plusieurs images du même point et considèrent l'appariement de différents points de vue comme un problème de classification. Finalement certains travaux [9], [16] considèrent des scènes dont le modèle 3D est connu et exploitent cette information pour créer des descripteurs invariants aux changements de point de vue.

Une autre approche consiste, au lieu de chercher à représenter un point de manière indépendante du point de vue, à définir un modèle 3D du point et à adapter sa représentation au point de vue. En particulier, [3], [5] et [12] considèrent que les points d'intérêt reposent localement sur des surfaces planes. Après avoir déterminé leur orientation, il est possible en utilisant une prédiction de la pose de la caméra, d'utiliser l'homographie induite par les plans pour leur donner la même apparence que l'observation courante.

Dans le cadre de la localisation d'un robot, la position peut être prédite à partir d'un modèle de mouvement du robot. Cette information est ignorée lorsque l'on utilise un descripteur invariant au point de vue alors qu'elle amène des informations qui peuvent rendre l'appariement plus robuste. L'utilisation du modèle 3D tire parti de cette information et les expériences pratiques montrent l'amélioration. En particulier, [5] n'utilise qu'une seule caméra et une séquence d'apprentissage pour apparier davantage de points.

1.2 Implémentation GPU

L'algorithme [5] est généralement limité à un petit nombre de points d'intérêt par un temps de calcul prohibitif. Pour pallier ce problème et parvenir à des calculs en temps réel, il est possible d'utiliser la puissance de calcul intégrée dans les processeurs massivement parallèles des cartes graphiques. Néanmoins, transformer une implémentation classique en un processus parallèle présente certaines difficultés. En général la conception même de l'algorithme doit être revue pour tirer partie de la structure parallèle et de l'organisation de la mémoire du GPU.

Certains travaux ont déjà permis de réaliser des appariements en utilisant le GPU. Par exemple les algorithmes du KLT et SIFT ont été portés sur GPU ([15]). Dans cet article, la librairie OpenGl permet l'utilisation de la carte graphique et les résultats montrent une nette amélioration des performances par rapport aux implémentations sur CPU. Un autre descripteur de point célèbre, le SURF a été implémenté en GPU par [6], en utilisant l'interface de programmation CUDA [13] [1]. Pour finir, un programme d'appariement de points fonctionnant sur GPU a été développé par [14].

Nos travaux consistent à implémenter l'algorithme décrit en [5] sur le GPU. Cet algorithme utilise des plans projetés dans une scène 3D pour construire une vue synthétique basée sur la prédiction de la position du robot. Cela est bien adapté au travail sur GPU, car non seulement l'algorithme est facilement parallélisable en traitant chaque patch séparément, mais en plus la reprojection des patchs nécessite de nombreuses interpolations linéaires qui, si elle sont relativement lente sur le CPU, peuvent être réalisées très efficacement par le GPU qui intègre une structure câblée spécifiquement pour cela.

Dans ce papier nous allons tout d'abord rappeler le fonctionnement de l'algorithme. Par la suite, nous décrirons l'implémentation sur le GPU et donnerons quelques résultats comparatifs pour montrer l'apport du GPU sur le temps de calcul.

2 Algorithme de localisation

2.1 Patchs-plan

Dans notre application, le véhicule est d'abord conduit manuellement dans la zone de test pour enregistrer une séquence d'images avec la caméra embarquée. Cette séquence permet de générer les patch-plans comme décrit dans [5]. Le principe consiste à suivre plusieurs points sur l'ensemble de la séquence et, en utilisant un algorithme de reconstruction à partir du mouvement (structure from motion), la position 3D des points et la pose des caméras sont calculées. L'hypothèse est faite que ces points reposent sur une surface localement plane. La normale de ce plan est alors calculée grâce à une optimisation utilisant la transformation entre les différentes vues. Après cela l'homographie induite par le plan peut être utilisée pour associer une texture au patch. La texture correspond en fait à l'image du patch tel qu'il serait vue depuis une pose de caméra Ψ_{patch}. Il ne s'agit pas forcément de la vue fronto-parallèle. Ceci est important pour éviter de devoir faire un important rééchantillonnage lorsque le patch a été observé de côté. Un facteur de qualité est ensuite évalué pour éliminer les patchs issus de faux appariements. Pour finir, une zone d'observabilité est définie comme suit : si une caméra n'est pas dans la zone d'observabilité du

1. CUDA pour Compute Unified Device Architecture est fournie par Nvidia et permet de programmer des applications sur GPU dans une syntaxe très proche du C

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP

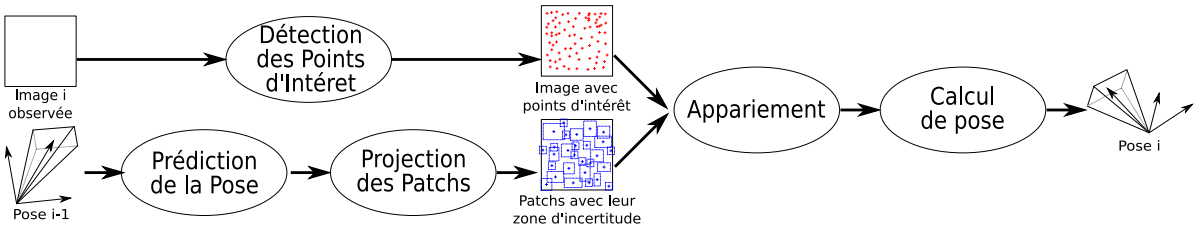

FIGURE 1 – Diagramme représentant l'algorithme de localisation

patch Π, alors il est inutile d'essayer de trouver Π sur l'image.

En résumé les valeurs disponibles pour la localisation sont :
– les coordonnées 3D du point P au centre du patch
– l'orientation de la normale au plan
– la texture du patch
– la pose Ψ_{patch} associée à la texture.
– la zone d'observabilité

2.2 Localisation par analyse des patchs

La figure 1 décrit l'algorithme de localisation pour une itération i.
Chaque pose i est d'abord prédite en utilisant la pose $i-1$ et un modèle d'évolution du robot. Cette prédiction permet alors de connaître la position approximative des patchs dans la vue courante ainsi que leur zone d'incertitude. De plus, la position prévue de la caméra permet, en utilisant la zone d'observabilité, d'éliminer directement les patchs qui ne sont pas visibles.
On applique l'homographie qui permet de projeter les patchs dans la vue virtuelle et de créer ainsi de nouveaux patchs Π. L'image des patchs tels que vus depuis la pose prédite permet alors de chercher les correspondances dans l'image.
Parallèlement, le même détecteur de points d'intérêts que celui de l'algorithme de reconstruction est appliqué sur l'image courante pour obtenir l'ensemble des positions P_{img} où pourrait se situer un patch. Le détecteur utilisé doit être le même que celui de l'algorithme de reconstruction, pour garantir de trouver les mêmes points que ceux qui ont généré les patchs.
Ensuite, chaque P_{img} qui se trouve dans la zone d'incertitude d'un patch est considéré comme une correspondance potentielle. On définit alors le vecteur Δ allant de P_{img} au centre du patch et calcule le score de correspondance S avec une ZNCC selon l'équation 1.

$$S = \frac{\sum_{x \in E} \left[\left(\Pi(x+\Delta) - \overline{\Pi} \right) \left(I_i(x) - \overline{I_i} \right) \right]}{\sqrt{\sum_{x \in E} \left(\Pi(x+\Delta) - \overline{\Pi} \right)^2 . \sum_{x \in E} \left(I_i(x) - \overline{I_i} \right)^2}} \quad (1)$$

avec E le voisinage du point d'intérêt, un carré de 16 par 16 pixels dans notre cas, I_i l'image courante.

Les appariements avec un score plus petit que le seuil de 0,5 sont éliminés. Il est inutile d'avoir un descripteur indépendant du point de vue pour ce test puisque les patchs ont été reprojetés et ont donc théoriquement la même apparence que l'image. Seules des différences de luminosité peuvent perturber la correspondance lors de changement météorologique ou de changement de conditions d'éclairement (matin et soir par exemple). C'est pourquoi une ZNCC est utilisée, elle permet d'être robuste à des changements affines de luminosité tout en restant assez rapide. Cette opération est appliquée pour toutes les correspondances afin de leur associer un score à chacune.
Pour terminer, on vérifie que chaque patch et chaque point d'intérêt ne sont pas présents dans deux correspondances différentes. Lorsque ce cas se présente, la correspondance avec le plus faible score est éliminée. On peut donc lier les coordonnées 3D des patchs avec les coordonnées dans l'image des P_{img} correspondants et en utilisant l'algorithme décrit en [1] on détermine la pose de caméra qui minimise l'erreur de reprojection.

2.3 Résultats obtenus

Pour tester les résultats de cet algorithme, une séquence a été prise dans un espace dégagé, ce qui permet d'utiliser un GPS différentiel de précision centimétrique comme référence. Le véhicule a un déplacement rectiligne d'une longueur d'environ 72 m sur le bord droit de la zone lors de l'apprentissage. La séquence obtenue lors de ce déplacement est composée de 365 images à partir desquelles 911 patchs utilisables pour la localisation ont été générés. Les images utilisées avaient une résolution de 512×384 pixels prises avec une caméra d'ouverture 120°.
Par la suite, différentes séquences ont été réalisées où le véhicule s'écarte de la trajectoire d'apprentissage en se positionnant à une distance comprise entre 1 et 8 m, comme illustré sur le schéma de la figure 2.

Nombre de points appariés. Pour vérifier les apports de cet algorithme, le nombre d'appariements corrects a été comparé avec un autre algorithme, nommé par la suite `algorithme SURF`, qui, au lieu d'utiliser les patchs, récupère directement les points vus depuis une image de référence. A chaque itération, la prédiction permet de déterminer quelle est l'image de référence la plus proche de la position, et les points de l'image de référence sont comparés à ceux de l'image courante. Pour réaliser la com-

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP

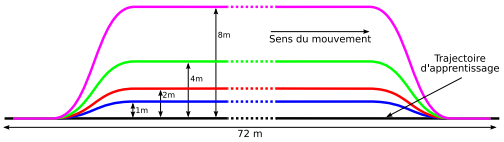

FIGURE 2 – Schéma représentant les trajectoires réalisées (en couleur) à coté de la trajectoire de référence (en noir)

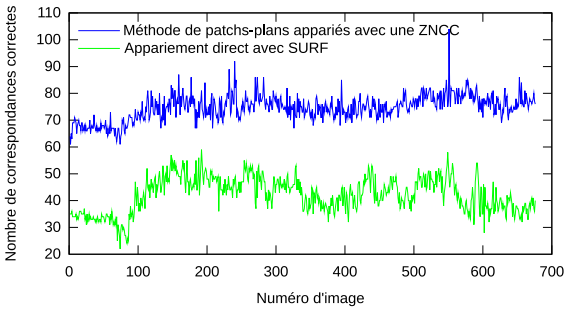

FIGURE 3 – Nombre d'appariements corrects au cours de la séquence de 1 077 images lorsque le véhicule était à environ 1 m de la trajectoire d'apprentissage

paraison, cette méthode utilise le descripteur SURF [2] qui est robuste aux changements de point de vue, et a été implémenté directement avec les librairies fournies par son auteur. D'autre part, le descripteur SURF ayant besoin du facteur d'échelle de la détection pour être calculé, le détecteur de point d'intérêt fourni avec la librairie a été utilisé. Ce même détecteur a été également utilisé avec les patchs afin de comparer les résultats dans les mêmes conditions. Ainsi pour chaque image les mêmes points sont détectés et les mêmes points de référence sont disponibles. En comparant le nombre d'appariements corrects, la pertinence de chaque descripteur est donc bien évaluée. Le résultat de cette comparaison est représenté figure 3 pour la trajectoire à 1 m de la référence. L'utilisation des patchs améliore significativement le nombre d'appariements corrects par rapport à l'autre méthode. Le fait d'avoir davantage de points diminue le risque, lors d'un écart à la trajectoire par exemple, de ne plus avoir assez d'appariements pour se localiser. Cela rend donc la localisation plus robuste.

Précision de localisation. Par la suite, afin de comparer la précision de la localisation, la position fournie par chaque algorithme (le notre et `algorithme SURF`) est comparée aux données GPS qui servent de référence. Malheureusement, la reconstruction à partir des images est réalisée dans un repère R_{Vision} différent du repère R_{GPS} dans lequel sont définies les coordonnées GPS. Il n'est donc pas possible de comparer directement les positions calculées dans R_{Vision} et les positions dans R_{GPS}. La précision est donc évaluée en mesurant l'écart latéral par rapport à la référence donnée par les différentes méthodes à

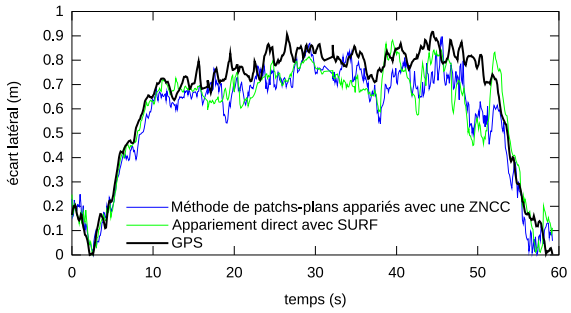

FIGURE 4 – Ecart latéral lors de la séquence à environ 1 m de l'apprentissage.

chaque instant. Pour la trajectoire située à environ 1 m de l'apprentissage on obtient les courbes figure 4.
On s'aperçoit que globalement la distance à la référence fournie par le GPS est à peu près la même que celle fournie par les deux algorithmes de vision. Pour le confirmer numériquement, on calcule l'erreur de localisation des algorithmes de vision en prenant le GPS comme référence. Notre algorithme utilisant les patchs a une erreur moyenne de 8,2 cm, avec un écart type de l'erreur de 5,8 cm. Pour l'`algorithme SURF` l'erreur moyenne est de 7,8 cm pour un écart type de l'erreur de 6,6 cm. La précision est donc clairement identique compte tenu du bruit sur la position. Notre algorithme n'augmente donc pas la précision de l'algorithme. Cependant comme vu précédemment, il apporte des appariements supplémentaires qui seront avantageux au niveau de la robustesse.
Lorsqu'on s'éloigne à 2 m de la trajectoire d'apprentissage, on obtient sensiblement les mêmes résultats que précédemment. Puis à 4 et 8 m, l'`algorithme SURF` ne parvient plus à se localiser. Avec notre méthode la localisation est toujours possible, mais perd sensiblement de la précision lorsque l'on dépasse les 2 m de distance. Au bilan on s'aperçoit que l'on a généralement la même précision que l'`algorithme SURF` mais lorsque l'on s'écarte de la trajectoire d'apprentissage, on parvient toujours à se localiser au-delà des limites des algorithmes basés sur des descripteurs.

Remarques. Lors des expériences précédentes, afin de comparer dans les mêmes conditions les algorithmes, seul le détecteur de points d'intérêt fourni avec la librairie de SURF (qui fournit le facteur d'échelle des points) a été utilisé. Cependant, il est possible avec notre méthode d'utiliser d'autres types de détecteur, par exemple le détecteur de Harris. Ce dernier détecteur, bien que ne fournissant pas de facteur d'échelle, est capable de trouver davantage de points sur l'image, et lorsque on l'utilise avec notre algorithme, les performances sont significativement améliorées, permettant de s'éloigner encore davantage de la trajectoire d'apprentissage sans perdre la localisation. On peut noter que l'absence de facteur d'échelle n'est pas un problème avec notre méthode, puisque la reprojection

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP

des patch tiendra compte du changement de position et donc du changement d'échelle du patch.

Par ailleurs, l'utilisation des patchs nécessite de prédire la pose de la caméra. Comme les patchs sont projetés dans la vue prédite, ils sont très sensibles à la précision de cette prédiction. Cela reste un réel problème lors de l'initialisation avec la première image de la séquence, puisque aucune information n'est disponible pour la prédiction. Une manière de réaliser cette initialisation pourrait être en utilisant un GPS bas-coût, ou d'utiliser un algorithme de localisation globale tel que [8].

Pour finir un dernier avantage des patchs est la faible occupation en mémoire. En effet, au lieu de sauvegarder l'ensemble des observations précédentes, pour comparer les différentes positions, seulement un patch est sauvegardé ainsi que les coordonnées de la normale.

Néanmoins tous ces résultats n'ont pu être réalisés que lors de tests sur table à cause de la lourdeur de calcul. L'algorithme de localisation utilisant les patchs est en effet, dans son implémentation sur CPU, trop lent, pour une utilisation temps-réel, nécessitant près de 2 secondes par image avec un processeur core 2 duo de 2,5 GHz. Afin de pallier ce problème une implémentation sur GPU a été mise en œuvre.

3 Implémentation sur GPU

Pour résumer, l'algorithme ainsi décrit utilise la force brute pour projeter de nombreux patchs sur toutes les images. Bien que les résultats semblent prometteurs, le fort coût de calcul rend l'utilisation en temps réel impossible. Cependant, de nombreuses opérations peuvent être parallélisées, en traitant plusieurs patchs simultanément. De plus, un profilage a montré que la grande majorité du temps de calcul est passée à réaliser l'interpolation linéaire nécessaire pour reprojeter les patchs. Pour ces raisons nous avons décidé d'utiliser l'implémentation sur carte graphique. Cette structure présente l'avantage non seulement d'être massivement parallèle mais possède également une structure matérielle intégrée pouvant réaliser l'interpolation linéaire de manière extrêmement rapide. L'implémentation a été réalisée en utilisant l'interface de programmation CUDA fournie par Nvidia dont la syntaxe proche du langage C est plus intuitive. La structure de l'algorithme parallélisé est détaillée sur la figure 5. Il s'agit pratiquement du même fonctionnement que l'implémentation CPU (figure 1), avec deux traitements (la détection des point d'intérêt et la reprojection des patch) au départ, se regroupant au moment du calcul du score. On peut voir en sus le passage, selon les besoin du CPU vers le GPU. Chaque case de la figure correspond à un kernel dont le fonctionnement sera décrit par la suite. Avant de détailler notre implémentation, nous allons réaliser un bref rappel des avantages et principes à adopter lors de la programmation sur GPU.

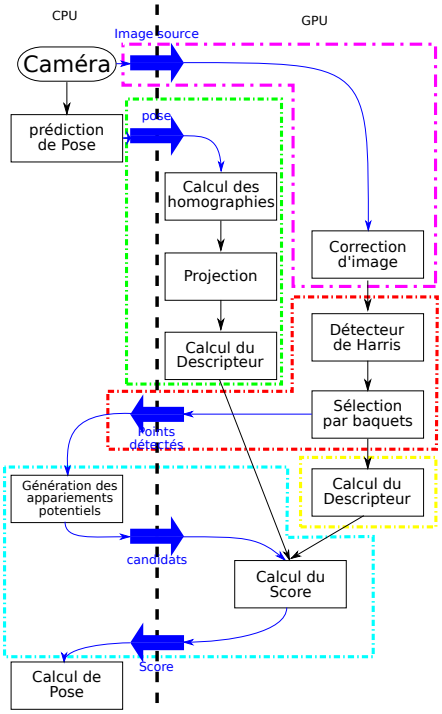

FIGURE 5 – Schéma global de l'implémentation sur GPU. Les grosse flèches bleues symbolisent les transferts de donnée entre CPU et GPU, et chaque bloc en pointillé correspond aux étapes décrites sur la figure 1 dont le temps d'exécution est mesuré par la suite. Les rectangle du coté GPU correspondent chacun à une kernel.

3.1 Programmation GPU

Notion de Kernel. Tout d'abord, chaque partie d'un programme exécuté par un GPU se présente sous la forme d'un kernel qui s'exécute simultanément en plusieurs threads (32 sur notre matériel). Ces threads sont groupés dans des `blocs`. Chaque multiprocesseur se charge d'exécuter un bloc de thread en répartissant les threads sur ses différents cœurs. Comme il peut y avoir plusieurs multiprocesseurs sur les cartes graphiques, les blocs sont également multipliés et regroupés dans une `grille`. Chaque thread exécute toujours la même partie du code (le `kernel`). Les threads se différencient uniquement par leur coordonnées dans leur bloc et les coordonnées du bloc dans la grille. Ces coordonnées peuvent être organisées en une, deux ou trois dimensions. Le diagramme de la figure 6 donne un exemple de cette organisation en bloc et grille des threads. Pour une application de traitement d'images, le plus simple est d'associer chaque pixel à un thread en conservant l'organisation bidimensionelle de l'image.

Utilisation de la mémoire. La mémoire embarquée dans le GPU est divisée en plusieurs parties en fonction de son usage. La mémoire `globale` est accessible par tous les threads. Cependant, sa lenteur d'accès et la nécessité d'alignement des données fait qu'on ne doit l'utiliser que

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP

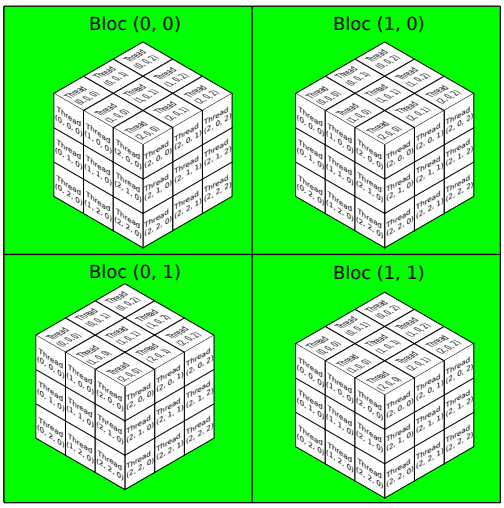

FIGURE 6 – Exemple d'organisation en grille et bloc. la grille en vert est composée de 2×2 blocs divisés en 3×3×3 threads

lorsque c'est indispensable.

L'accès à la mémoire `partagée` est beaucoup plus rapide mais le partage ne peut être fait qu'entre threads d'un même bloc. De plus, la taille de cette mémoire est limitée et sa durée de vie est celle d'un kernel. Les données sont donc effacées lorsque l'exécution du kernel se termine. Cette mémoire est donc principalement utilisée pour conserver des valeurs intermédiaires dans un calcul et les partager entre tous les threads d'un bloc.

La mémoire de `texture` est une partie de la mémoire globale qui peut être utilisée en lecture par chaque thread sans se préoccuper de l'alignement des données. De plus l'accès en lecture est très rapide et peut intégrer une interpolation linéaire.

La mémoire `constante` est également accessible très rapidement par chaque thread mais seul le CPU écrire des données dans cette mémoire.

Les flux de traitement (Stream). Un processus GPU peut être mis en œuvre en plusieurs flux de traitement en parallèle. Les flux ne sont pas réellement exécutés en même temps (un seul kernel peut être exécuté par le GPU à un instant donné) mais les transferts de données et l'exécution du kernel peuvent se dérouler simultanément. Par exemple, en utilisant des flux différents pour la gestion de l'image et des patchs, l'image peut être transférée de la caméra vers le GPU pendant que les patchs sont projetés.

3.2 Détection des points d'intérêt

Un flux est associé au processus de détection des points d'intérêt. Il s'exécute dès que l'image arrive. S'enchaînent alors une correction de distorsion radiale, la détection des points de Harris et pour terminer le calcul de leur descripteur.

Correction d'image. La distorsion est évaluée lors d'une phase initiale et une table de rééchantillonnage est créée pour associer chaque pixel de l'image corrigée aux coordonnées (non entières) de l'image distordue. Cette table est stockée dans la mémoire globale du GPU. Le kernel `Image correction` utilise un thread par pixel et transforme l'image en utilisant la table de rééchantillonnage. L'image initiale est stockée en mémoire texture afin d'utiliser l'interpolation linéaire associée. L'image résultat est stockée dans la mémoire globale pour pouvoir être utilisée par le kernel suivant.

Détecteur de Harris. Le kernel `Détecteur de Harris` utilise l'image pour détecter les points d'intérêt. Pour cela l'image est divisée en plusieurs parties rectangulaires. Chaque partie correspond à un bloc qui est subdivisé pour avoir un thread par pixel. Les différentes parties de l'image sont transférées dans la mémoire partagée pour permettre au thread d'utiliser le voisinage des points lors du calcul du critère de Harris.

Après avoir calculé le critère de Harris, seuls les maxima locaux sont conservés dans la mémoire globale sous la forme d'une image contenant une valeur nulle si le point n'est pas un maximum ou la valeur du critère si le point est effectivement un maximum.

Sélection des points détectés. Par la suite le kernel `Sélection par baquets` découpe l'image obtenue précédemment en plusieurs baquets (8 divisions horizontales et 4 divisions verticales dans nos tests) et sélectionne uniquement les meilleurs points dans chacune. Pour cela un thread est utilisé pour chaque subdivision dont il parcourt tous les pixels pour mettre les points détectés dans une table de valeurs. Le résultat obtenu à la fin est une table contenant seulement les meilleurs points détectés dans chaque baquet. Ces valeurs sont envoyées au CPU qui sélectionnera les appariements potentiels (partie 3.4).

3.3 Projection des patchs

La totalité de l'information concernant les patchs est transférée initialement dans le GPU. La mémoire texture reçoit d'une part la matrice de rotation, le vecteur de translation associé à la pose d'un patch ainsi que l'orientation de sa normale et les coordonnées 3D du point central et d'autre part la texture plane du patch.

Les données sont stockées sous la forme d'une mosaïque contenant 128 patchs par ligne. Cela permet d'utiliser un flux par ligne, et de paralléliser ainsi les transferts de données avec les calculs de projection.

Calcul des homographies. Pour réaliser la projection des patchs, la pose est tout d'abord prédite par le CPU en utilisant un modèle de mouvement du robot. La prédiction est ensuite envoyée à la mémoire constante du GPU et le kernel `Calcul des homographies` calcule la matrice 3×3 d'homographie qui transforme chaque patch de la vue de référence en sa vue depuis la pose de la caméra. Neuf threads sont utilisés pour chaque patch, chacun calculant une valeur de la matrice d'homographie.

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP

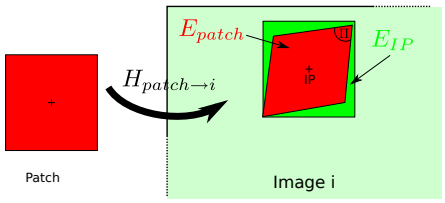

FIGURE 7 – Projection d'un patch sur une image i. On peut constater que le voisinage E_{patch} du patch ne correspond pas parfaitement avec le carré E_{IP} autour du point d'intérêt.

Projection. Le kernel `Projection` utilise la matrice d'homographie pour réaliser la reprojection. Chaque patch correspond à un bloc, avec un thread par pixel. L'image de chaque patch ainsi obtenue est stockée dans la mémoire globale.

3.4 Appariement

Génération des appariements potentiels. Le CPU sélectionne tout d'abord une liste de paires potentielles en recherchant les points d'intérêt présents dans la région d'intérêt autour des patchs. Cette liste est ensuite envoyée au GPU pour calculer les scores d'appariement.

Calcul du score. Lorsque les patchs sont reprojetés, la forme obtenue n'est généralement pas un carré parfait, laissant sur les bords des endroits où aucune valeur n'est disponible, comme le montre la figure 7. Le voisinage du patch E_{patch} n'est donc pas exactement le même que le carré E_{IP} défini autour des points d'intérêt.

Le calcul du score d'appariement S, utilisant la ZNCC définie par l'équation 1 est donc modifié pour devenir l'équation 2 :

$$S = \frac{\frac{\sum_{x \in E_{patch} \cap E_{IP}} \left[\left(\Pi(x) - \overline{\Pi}\right) \left(I_i(x) - \overline{I_i}\right) \right]}{N}}{\sqrt{\frac{\sum_{x \in E_{patch}} \left(\Pi(x) - \overline{\Pi}\right)^2}{N_{patch}}} \cdot \sqrt{\frac{\sum_{x \in E_{IP}} \left(I_i(x) - \overline{I_i}\right)^2}{N_{IP}}}} \quad (2)$$

avec N le nombre d'éléments de $E_{patch} \cap E_{IP}$ et N_{patch} (respectivement N_{IP}) le nombre d'éléments de E_{patch} (respectivement E_{IP}).

Comme l'ensemble E_{patch} est pratiquement identique à E_{IP} les résultats sont très similaires à la corrélation réalisée uniquement sur l'ensemble $E_{patch} \cap E_{IP}$. L'avantage de ces changements est de pouvoir définir séparément le descripteur de chaque élément. Ainsi, le calcul du score se réduit à une simple multiplication et lorsqu'un même point est comparé à plusieurs patchs, son descripteur n'est calculé qu'une seule fois.

Le kernel `Calcul du Descripteur` se charge, à partir soit de l'image du patch soit de l'image source et d'un point d'intérêt, de générer le descripteur D en utilisant pour chaque pixel x de l'image l'équation :

$$D(x) = \frac{I(x) - \overline{I}}{\sqrt{\sum_{x \in E} \left(I(x) - \overline{I}\right)^2 / N}} \quad (3)$$

Le dénominateur est obtenu grâce à un algorithme de réduction décrit dans [7] avec chaque thread correspondant à un pixel et un bloc correspondant à un descripteur.

Le kernel `Calcul du Score` calcule la moyenne du produit des descripteurs conformément à l'équation 2, en utilisant le même algorithme de réduction pour la moyenne.

3.5 Résultats

Les implémentations sur CPU et GPU ont été comparées avec le même jeu de données. L'algorithme utilisé sur les deux architectures est identique, en incluant les changements décrits section 3.4 pour tenir compte des différences de voisinage. La séquence est composée de 509 images avec une résolution de 512×384 pixels sur lesquelles ont été générées 4033 patchs de bonne qualité. L'ordinateur utilisé est le même pour les deux implémentations. Il s'agit d'un portable avec processeur core 2 duo de 2.5 GHz et 2 Go de RAM. Sa carte graphique est une nvidia quadro FX770M avec 512 Mo de mémoire. Il s'agit d'une carte d'entrée de gamme pour un portable.

Les scores d'appariement obtenus sont identiques avec les deux méthodes. Le temps mesurés dans chaque partie du programme correspond aux blocs pointillés de la figure 5 en incluant les temps de transfert :

- Correction d'image comprenant le kernel `Correction d'image` et le transfert de l'image vers le GPU (bloc rose)
- Détection comprenant les kernels `Détecteur de Harris` et `Sélection par baquets` ainsi que le transfert des points détectés vers le CPU (bloc rouge)
- Projection des patchs comprenant le transfert de la pose prédite et les kernels `Calcul de l'homographie`, `Projection` et `Calcul du descripteur` lorsqu'il est utilisé avec les patchs (bloc vert)
- Descripteur de PI comprenant le kernel `Compute Descriptor` lorsqu'il est utilisé avec les points d'intérêt (bloc jaune)
- Appariement comprenant la génération des appariements potentiels, le kernel `Score` et le transfert des candidats et des scores entre GPU et CPU (bloc cyan)
- Le temps total du processus complet

Les résultats sont résumés dans le tableau 1.

On peut constater que les principaux gains en temps de calcul sont réalisés lors du calcul des descripteurs. Ce type d'algorithme est en effet massivement parallélisable et tire également parti de l'interpolation matérielle utilisée par le biais de la mémoire texture.

A l'opposé le processus de détection est plus long sur le GPU que sur le CPU. Cela est dû au fait que la détection sur

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP

Kernel	Correction d'image	Détection	Projection des patchs	Descripteur de PI	Appariement	Total
temps CPU (ms)	20.5	7.7	2572.8	228.1	58.0	2887.4
temps GPU (ms)	4.4	30.6	96.4	1.5	24.4	165.1

TABLE 1 – Temps d'exécution comparé entre les implémentation sur CPU et GPU. Le temps exprimé en millisecondes est le temps de traitement moyen par image.

le CPU est déjà optimisée en utilisant les instruction SSE2 pour paralléliser le traitement, ce qui atténue les améliorations possibles dues à la parallélisation massive sur le GPU. De plus, les points détectés étant situés aléatoirement sur l'image, les accès mémoire lors de la sélection des points ne sont pas alignés entre les threads, ce qui décroit grandement les performances du GPU. Pour finir le temps de transfert de la détection vers le CPU est également inclus dans la mesure. Cette partie n'existe pas lorsque l'on est sur le CPU, puisque les données sont déjà disponibles. Cela creuse donc encore l'écart de temps de calcul entre les deux implémentations. Une possibilité d'amélioration pourrait être de traiter la sélection des points sur le CPU, mais dans ce cas, il serait nécessaire de transférer tout le critère de Harris (et non seulement les points détectés) et le temps gagné par le CPU serait perdu dans le transfert.

Finalement, le temps total de traitement a été divisé par un facteur 17, ce qui permet d'utiliser un tel algorithme de localisation en temps-réel pour la navigation d'un robot mobile autonome.

4 Conclusion

Nous avons constaté que l'appariement d'amers utilisé pour la localisation d'un robot peut être significativement amélioré en utilisant la programmation sur GPU et ce, même avec une carte graphique d'entrée de gamme. Le gain en performance est particulièrement important lorsque l'algorithme nécessite de réaliser des interpolations ou des projections. Par ailleurs d'autres améliorations pourraient encore être réalisées, par exemple en évitant les transferts vers le CPU en traitant chaque étage sur le GPU.

Cet algorithme est particulièrement bien adapté à une utilisation en GPU, du fait que l'interpolation est la tâche qui lui prend le plus de temps. De plus le fait d'utiliser l'information disponible sur le mouvement du robot lui permet d'avoir une localisation plus robuste.

Références

[1] H. Araujo, RL Carceroni, and CM Brown. A fully projective formulation to improve the accuracy of Lowe's pose-estimation algorithm. *Computer vision and image understanding(Print)*, 70(2) :227–238, 1998.

[2] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110(3) :346–359, 2008.

[3] C. Berger and S. Lacroix. Using planar facets for stereovision SLAM. In *IROS*, pages 1606–1611, 2008.

[4] M. Calonder, V. Lepetit, and P. Fua. Keypoint signatures for fast learning and recognition. In *ECCV*, pages 58–71, 2008.

[5] B. Charmette, É. Royer, and F. Chausse. Mise en correspondance de patchs plan pour la localisation d'un robot. In *RFIA 2010*, January 2010.

[6] N. Cornelis and L. Van Gool. Fast scale invariant feature detection and matching on programmable graphics hardware. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008. CVPR Workshops 2008*, pages 1–8, 2008.

[7] M. Harris. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2007.

[8] A. Irschara, C Zach, J-M Frahm, and H. Bischof. From structure-from-motion point clouds to fast location recognition. In *CVPR*, 2009.

[9] K. Koser and R. Koch. Perspectively invariant normal features. In *ICCV 2007*, pages 1–8, 2007.

[10] V. Lepetit, P. Lagger, and P. Fua. Randomized trees for real-time keypoint recognition. In *CVPR*, pages 775–781, 2005.

[11] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157, 1999.

[12] N. Molton, A. Davison, and I. Reid. Locally planar patch features for real-time structure from motion. In *BMVC*, 2004.

[13] NVIDIA. http://www.nvidia.fr/cuda.

[14] M. Schweitzer and H-J. Wuensche. Efficient Keypoint Matching for Robot Vision using GPUs. In *Fifth IEEE Workshop on Embedded Computer Vision (ECV'09)*, 2009.

[15] S.N. Sinha, J.M. Frahm, M. Pollefeys, and Y. Genc. GPU-based video feature tracking and matching. In *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, volume 278, 2006.

[16] C. Wu, B. Clipp, X. Li, J.M. Frahm, and M. Pollefeys. 3D model matching with Viewpoint-Invariant Patches (VIP). In *CVPR 2008*, pages 1–8, 2008.

Ce travail a été financé par l'ANR Agence Nationale de la Recherche dans le cadre du projet cityVIP