NNT : 2017SACLS058

Thèse de doctorat de l'Université Paris-Saclay préparée à l'Université Paris Sud

Ecole doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Robotique

par

M. ABOUZAHIR MOHAMED

Algorithmes SLAM: Vers une Implémentation Embarquée

Thèse présentée et soutenue à "Agadir, Maroc", le 25 février 2017.

Composition du Jury :

M.	Stéphane Espié	Directeur de Recherche IFSTTAR, Paris, France	(Président)
M.	Mohammed Essaaidi	Professeur ENSIAS, Rabat, Maroc	(Rapporteur)
M.	Said Belkouch	Professeur Habilité ENSA, Marrakech, Maroc	(Rapporteur)
M.	Mohamed EL Ansari	Professeur FSA, Agadir, Maroc	(Examinateur)
M.	Samir Bouaziz	Professeur Université Paris Sud - Paris Saclay	(Directeur de thèse)
M.	Abdelhafid Elouardi	Maître de conférence HDR Université Paris Sud - Paris Saclay	(Co-encadrant de thèse)
M.	Rachid Latif	Professeur ENSA, Agadir, Maroc	(Directeur de thèse)
M.	Abdelouahed Tajer	Professeur Habilité ENSA, Marrakech, Maroc	(Co-encadrant de thèse)

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Algorithmes SLAM: Vers une Implémentation Embarquée

Keywords : Localisation et cartographie, Systèmes Embarqués, Implémentation

Résumé : La navigation autonome est un axe de recherche principal dans le domaine de la robotique mobile. Dans ce contexte, le robot doit disposer des algorithmes qui lui permettent d'évoluer de manière autonome dans des environnements complexes et inconnus. Les algorithmes de SLAM (Simultaneous Localization et Mapping) permettent à un robot de cartographier son environnement tout en se localisant dans l'espace. Les algorithmes SLAM sont de plus en plus performants, mais aucune implémentation matérielle ou architecturale complète n'a eu lieu. Une telle implantation d'architecture doit prendre en considération la consommation d'énergie, l'embarquabilité et la puissance de calcul.

Ce travail scientifique vise à évaluer des systèmes SLAM embarqués. La méthodologie adoptera une approche A3 (Adéquation Algorithme Architecture) pour améliorer l'efficacité de l'implantation des algorithmes. Le système SLAM embarqué doit disposer d'une architecture électronique et logicielle permettant d'assurer la production d'information pertinentes à partir de données capteurs, tout en assurant la localisation de l'embarquant dans son environnement.

Les premiers travaux de cette thèse ont consisté à explorer les différentes approches algorithmiques permettant la résolution du problème de SLAM. Cette étude, nous a permet d'évaluer quatre algorithmes de différente nature: FastSLAM2.0, ORB SLAM, RatSLAM et le SLAM linéaire. Ces algorithmes ont été ensuite évalués sur plusieurs architectures pour l'embarqué afin d'étudier leur portabilité sur des systèmes de faible consommation énergétique et de ressources limitées. Après avoir analysé profondément les évaluations temporelles de chaque algorithme, le FastSLAM2.0 est finalement choisi, pour un compromis temps d'exécution-consistance de résultat de localisation, comme candidat pour une étude plus approfondie sur une architecture hétérogène embarquée.

La second partie de cette thèse est consacré à l'étude d'un système embarqué implémentant le FastSLAM2.0 monoculaire dédié aux environnements larges. Une réécriture algorithmique du FastSLAM2.0 a été necessaire afin de l'adapter au mieux aux contraintes imposées par les environnements de grande échelle. Dans une démarche A3, le FastSLAM2.0 a été implanté sur une architecture hétérogène CPU-GPU. Grâce à un partitionnement efficace, un facteur d'accélération global de l'ordre de 22 a été obtenu sur une architecture récente dédiée pour l'embarqué. Une deuxième instance matérielle basée sur une architecture programmable FPGA est proposée. Les gains obtenus sont conséquent, même par rapport aux GPU haut-de-gamme qui disposent actuellement d'un grand nombre de c?urs. Le système résultant peut cartographier des environnements larges tout en garantissant le compromis entre la consistance des résultats de localisation et le temps réel.

Ces travaux de thèse ont permis de mettre en avant l'intérêt des architectures hétérogènes parallèles (multicoeurs-GPU) pour le portage des algorithmes SLAM. Les architectures hétérogènes à base du FPGA peuvent particulièrement devenir des candidats potentiels pour porter des algorithmes complexes traitant des données massives.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : SLAM Algorithms: Towards an Embedded Implementation

Keywords : Localization and mapping, Embedded Systems, Implementation

Abstract : Autonomous navigation is a major research area in the field of mobile robotics. In this context, the robot must have algorithms that allow it to evolve autonomously in a complex and unknown environments. The SLAM algorithms (Simultaneous Localization and Mapping) allow a robot to map its environment while localizing itself in the explored space. SLAM algorithms are promising, but there is no complete hardware or architectural implementation has taken place. Such architectural implementation must take into account the energy consumption, the embeddability and the computing power.

This scientific work aims at evaluating embedded SLAM systems. The methodology will adopt an AAM (Algorithm Architecture Matching) approach to improve the efficiency of the implementation of the algorithms. The embedded SLAM system must have an electronic and software architecture enabling the production of relevant information from sensor data while ensuring the localization of the embedded platform in the explored environment.

The first work of this thesis consisted in exploring the different algorithmic approaches allowing to resolve the SLAM problem. This study allowed us to evaluate four different algorithms: FastSLAM2.0, ORB SLAM, RatSLAM and linear SLAM. These algorithms were then evaluated on several embedded architectures in order to study their portability on embedded systems with low energy consumption and limited resources. After a thorough analysis of the temporal evaluations of each algorithm, the FastSLAM2.0 is ultimately chosen, for a compromise between execution time and consistency of localization result, as a candidate for further study on an embedded heterogeneous architectures.

The second part of this thesis is devoted to the study of an embedded system implementing the monocular FastSLAM2.0 dedicated to large scale environments. An algorithmic changes of the FastSLAM2.0 were necessary in order to adapt it to the constraints imposed by the large-scale environments. Using an AAM approach, the FastSLAM2.0 was implemented on a heterogeneous architecture based CPU-GPU. Thanks to efficient partitioning, an overall acceleration factor of the order of 22 was obtained on a recent architecture dedicated to embedded systems. A second hardware instance based on a programmable FPGA architecture is proposed. The gains obtained are consistent, over a high-end GPUs that currently have a large number of cores. The resulting system can map large environments while ensuring the compromise between the consistency of the localization results and real time performances.

This work allowed us to highlight the interest of parallel heterogeneous (multi-core-GPU) architectures for porting SLAM algorithms. Heterogeneous FPGA-based architectures can become potential candidates for complex algorithms dealing with massive data

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Remerciements

En tout premier lieu, je tiens à remercier Monsieur Stéphane Espié pour m'avoir fait l'honneur de présider mon jury de thèse et pour son rôle en tant qu'examinateur. Je tiens à remercier également Messieurs Mohammed Essaaidi, Said Belkouch et Mohamed El Ansari pour leur travail de rapporteurs.

Je tiens à remercier tout particulièrement mes directeurs de thèse Rachid Latif et Samir Bouaziz et mes co-directeurs de thèse Abdelhafid Elouardi et Abdelouahed Tajer qui m'ont encadré tout au long de ces trois années. Ils m'ont guidé, orienté, aidé et soutenu durant l'intégralité de ma thèse.

Je remercie le groupe MOSS du laboratoire Systèmes et Applications des Technologies de l'Information et de l'Énergie (SATIE) à l'école normale supérieure de Cachan (ENS) pour m'avoir accueilli afin de réaliser mes travaux de thèses au sein du groupe et pour le cadre de travail agréable qu'il m'a offert tout au long de ces trois années. Mes sincères remerciements vont à toute l'équipe systèmes embarqués à Digiteo et l'équipe signaux systèmes et informatique de l'ENSA d'Agadir pour l'accueil chaleureux dont j'ai bénéficié et à toutes les personnes avec qui j'ai pu partager mes connaissances et ma passion pour la robotique.

Je tiens à remercier aussi Mr Omar Hammami de m'avoir accueilli au sein de l'ENSTA ParisTech pour pouvoir évaluer les algorithmes développés sur une plateforme matérielle et utiliser les outils associés.

Je tiens à remercier également le service des relations internationales de l'université Paris Sud et le centre national pour la recherche scientifique et technique (CNRST) du Maroc de m'avoir octroyé des financements de soutien durant ces trois années de thèse. Je leur serai particulièrement reconnaissant pour cette généreuse contribution qui m'a donné un souffle pour continuer mes travaux de thèse.

Enfin, je tiens à remercier mes frères et sœurs qui m'ont toujours soutenu et encouragé tout au long de ces années d'études et mes parents sans qui je n'aurais pas réalisé le parcours universitaire que j'ai effectué. Je vous serai éternellement reconnaissant de m'avoir encouragé dans les moments difficiles.

Publications

Revues internationales avec comité de lecture

1. "Large Scale Monocular FastSLAM2.0 Acceleration on an Embedded Heterogeneous Architecture"
   M. Abouzahir, A. Elouardi, S. Bouaziz, R. Latif, A. Tajer. *EURASIP Journal on Advances in Signal Processing*, *SpringerOpen,* Juillet 2016, DOI : 10.1186/s13634-016-0386-3
2. "Implementation of FastSLAM2.0 on an Embedded System and HIL Validation Using Different Sensors Data"
   M. Abouzahir, A. Elouardi, S. Bouaziz, R. Latif, A. Tajer. *International Journal of Adaptive, Resilient and Autonomic Systems, IJARAS, 6(2), 2015.*

Congrès internationaux avec actes et comités de lecture

1. "High Level Synthesis For FPGA Design Based-SLAM Application"
   Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Omar Hammami and Ismail Ali. *13th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2016), November 29th to December 2nd, 2016.*
2. "Localization and Mapping Algorithms Implemented on a Low-Power Embedded Architectures : A case Study".
   Mohamed Abouzahir, Rachid Latif, Abdelouahed Tajer, Abdelhafid Elouardi and Samir Bouaziz. *5th IEEE International Conference on Multimedia Computing and Systems (ICMCS 2016). Marrakech-Maroc, 29 Septembre -1 Octobre 2016.*
3. "GPU Accelerated Robust-Laser Based Fast Simultaneous Localization and Mapping",
   Dai Duong Nguyen, Mohamed Abouzahir, Abdelhafid Elouardi, Bruno Larnaudie and Samir Bouaziz. *16th IEEE International Conference on Scalable Computing and Communications (ScalCom 2016). Toulouse, France, Juillet, 18-21, 2016.*
4. "FastSLAM2.0 Running on a Low-Cost Embedded Architecture"
   Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif and Abdelouahed Tajer. *13th IEEE International Conference on Robotics and Automation (ICARCV 2014). Marina Bay Sands, Singapour. Décembre 10-12, 2014.*
5. "An Improved Rao-Blackwellized Particle Filter Based-SLAM Running on an OMAP Embedded Architecture"
   Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif, Abdelouahed Tajer. *2nd IEEE World Conference on Complex System (WCCS 2014). Novembre 10-12, 2014, Agadir, Morocco*

6. "Platform Simulation Based Unmanned Aircraft Systems Design"
   Rabah Louali, Samir Bouaziz, Abdelhafid Elouardi and Mohamed Abouzahir. *2nd IEEE World Conference on Complex System (WCCS 2014). Novembre 10-12, 2014, Agadir, Morocco*
7. "FastSLAM2.0 : HIL Validation"
   Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Abdelouahed Tajer and Rachid Latif, Colloque International sur le Monitoring des Systèmes Industriels (CIMSI 2014). *Décembre, 25-26 2014, Marrakech, Maroc*

Table des matières

Table des figures

Liste des tableaux

Introduction générale

La conception de robots autonomes capables d'explorer des endroits inaccessibles à l'homme a été un sujet de recherche en plein essor depuis de nombreuses années. Aujourd'hui, l'intérêt pour le développement de ces robots est accru par l'émergence des systèmes robotisés déployés dans de nombreuses applications que ce soit pour explorer des mondes éloignés ou pour intervenir dans des situations dangereuses. Ces systèmes doivent disposer d'une capacité pour aller vers une autonomie totale dans des environnements forcément inconnus et qui ne peuvent pas être cartographié à l'avance par l'homme.

La navigation autonome d'un robot mobile consiste à répondre à deux questions fondamentales : Où suis-je et à quoi ressemble l'environnement d'entourage ? Le processus complet qui permet à un robot mobile d'identifier la nature de son environnement et de s'y localiser est connu sous l'acronyme SLAM (Simultaneous Localization And Mapping). La cartographie est la phase qui permet la construction d'une carte représentant la structure spatiale de l'environnement à partir de certaines informations récoltées par les capteurs du robot. La localisation revient, quand à elle, à déterminer la position du robot dans la carte qui correspond à sa position dans l'environnement réel.

Les algorithmes SLAM offrent des avantages indéniables au niveau de la modélisation de la scène, la localisation et la cartographie. Leur exigence en termes de calcul, de précision et d'embarquabilité constituent un facteur critique qui limite leur utilisation dans des plate-formes embarquées. D'un autre côté, les tendances vers des implémentations bas coût et le traitement de faible puissance nécessite un parallélisme massif et une implémentation hétérogène. L'implémentation des algorithmes de SLAM est souvent précédée par une étude formalisée (Approche A3 : Adéquation Algorithme Architecture) permettant d'une part d'effectuer des vérifications formelles le plus tôt possible afin d'assurer la faisabilité et la continuité de la conception et d'autre part, de poser des problèmes d'optimisation permettant de dimensionner au mieux les architectures envisagées. L'approche A3 permet d'effectuer une implantation adéquate de l'algorithme par l'étude simultanée de l'algorithme et l'architecture. L'adéquation est le processus de mise en correspondance de l'algorithme et de l'architecture. Ce processus consiste à étudier et maitriser le problème en prenant en compte certaines contraintes (embarquabilité, temps réel, nature de l'information à traiter et le besoin en puissance de calcul) pour concevoir un système unifiant l'algorithme et l'architecture tout en se basant sur une formalisation [4].

Plusieurs algorithmes SLAM ont été développés pour résoudre le problème de cartographie et localisation. Ces algorithmes sont catégorisés en fonction du type de capteurs et la nature de l'approche mathématique utilisée. Les premiers algorithmes SLAM étaient principalement basés sur l'utilisation des capteurs laser capable de fournir une information précise sur la profondeur des objets dans la scène. Ces capteurs restent généralement très couteux, ce qui limite leur utilisation pour le grand pu-

blic. Ensuite, la vision monoculaire a été adoptée pour remplacer les capteurs laser. Pour ce faire, de nombreuses recherches sont effectuées pour développer des algorithmes mathématique qui permettent la reconstruction 3D de la scène en utilisant une simple camera. Les techniques utilisées pour résoudre le problème du SLAM visuel peuvent être divisées en trois catégories. Les premières méthodes classiques sont basées sur les filtres probabilistes dans lesquelles le système SLAM décrit la position des amers et du robot par une représentation probabiliste dans l'environnement. Les deuxièmes méthodes sont des techniques qui utilisent le principe de structuration à partir du mouvement (SfM : Structure From Motion) de manière incrémental, ce qui permet de calculer la structure 3D de la position de la caméra et de la scène à partir d'un ensemble d'images. Finalement, les techniques basées sur les approches bio-inspirées sont apparues récemment et étaient étudiées suite à la découverte des cellules qui constituent le système de positionnement dans le cerveau. Cette approche consiste à utiliser des modèles de l'hippocampe chez les rongeurs pour concevoir des systèmes impliquant de la localisation et la cartographie simultanées. Cependant, les performances de ces algorithmes reste limitées pour des séquences d'images réduites et dans environnements internes de petite tailles. Leur utilisation en environnements larges avec plusieurs images requiert des modifications algorithmiques afin de les adapter aux contraintes de l'environnement et aussi en fonction des capacités spécifiques du la plateforme expérimentale utilisée.

Pendant ces dernières décennies, les algorithmes SLAM ont souvent été exécutés sur des machines performantes à cause de leur complexités. Ceci est réalisé dans l'objectif d'assurer l'exécution temps réel de ces algorithmes et de garantir la consistance des résultats de localisation et de cartographie. En pratique, les algorithmes SLAM deviennent plus complexes spécialement pour des tâches d'exploration d'endroits difficiles où la dimension du robot ne permet pas d'utiliser des machines performantes pour l'exécution temps réel. Par conséquence, l'utilisation des architectures embarquées s'avère donc indispensable. Dans ce cas, la disposition d'une architecture embarquée assurant une implémentation efficace est une nécessité pour assurer les contraintes temps réel. Aujourd'hui, il existe une variante de calculateurs embarqués possédant des spécificités propres qui leur permettent une grande efficacité. Or, ces spécificités peuvent être utilisées dans le contexte du SLAM pour optimiser le calcul et envisager un système performant du SLAM embarqué. Ces optimisations, peuvent être réalisées en utilisant des processeurs multicœurs et des architectures massivement parallèles tel que les processeurs graphiques où les architectures programmables FPGA.

L'émergence des architectures embarquées multicœurs à faible consommation, offre une opportunité d'explorer les possibilités du calcul embarqué des algorithmes SLAM. Cependant, le portage de ces algorithmes dépend fortement de la nature de l'algorithme et de l'architecture cible. Par conséquence, des contraintes algorithmiques et matérielles sont à tenir en compte pour définir un couple algorithme/architecture adéquat

Objectifs et contributions

L'objectif principal de cette thèse est l'étude de la portabilité des algorithmes SLAM sur des architectures hétérogènes embarquées. Cette étude nécessite en conséquence des phases de développement, de conception et d'évaluation de systèmes prototypes permettant l'exécution des applications SLAM à forte contraintes. L'étude se base sur le principe d'Adéquation Algorithme Architecture.
La première contribution consiste à étudier une variantes d'algorithmes SLAM de nature différentes.

Le choix de ces algorithmes est basé sur des critères liés à la nature des traitements, la consistance des résultats de localisation et de cartographie et le type des capteurs utilisés. Les algorithmes choisis sont : le FastSLAM2.0, l'ORB SLAM, le RatSLAM et le Linear SLAM. Ces algorithmes sont ensuite évalués sur des architectures embarquées afin d'étudier la possibilité de leur portabilité. Une comparaison est faite en se basant sur certaines métriques tels que les ressources mémoires utilisées et le temps d'exécution. Le FastSLAM2.0 est finalement choisi comme candidat potentiel pour des applications embarquées temps réel [5]
La seconde contribution consiste à valider l'aspect fonctionnel de l'algorithme FastSLAM2.0 avec des capteurs laser et des images dans des environnements simulés et réels de courte trajectoire [6, 7]. Un système de validation incluant le système embarqué dans une boucle de traitement logicielle est conçu pour valider le FastSLAM2.0 [8]. Ensuite, des réécritures algorithmiques sont étudiées pour adapter le FastSLAM2.0 avec une vision monoculaire pour des environnements larges à grande échelle [9].
Enfin, les modifications algorithmiques précédentes mettront en avant une nécessité de ressources de calcul importantes pour garantir l'aspect temps réel. En effet, ces modifications augmentent la consistance des résultats mais au détriment du fonctionnement en temps réel. Afin de trouver un compromis, nous proposons donc une implémentation parallèle complète du FastSLAM2.0 monoculaire conçue autour des architectures massivement parallèle à savoir le GPU [9] et le FPGA [10]. Cette contribution présente un système matériel/logiciel adapté au FastSLAM2.0 monoculaire à grande échelle.

Organisation du mémoire

Le premier chapitre présente une bibliographie des différents algorithmes SLAM. Le problème de localisation et de cartographie est rappelé. Différentes applications du SLAM dans le domaine de la robotique mobile sont présentées. Ensuite, les algorithmes SLAM sont étudiés en fonction de l'approche mathématique utilisée, le types des capteurs intégrés ainsi que leurs performances en termes de résultat de localisation et de reconstruction. Finalement, un état de l'art sur les architectures des systèmes embarqués dédiés pour résoudre la problématique SLAM est présenté.

Le second chapitre est consacré à la définition de la méthodologie A3 et ses formalismes mathématiques ainsi qu'au différentes étapes de l'adéquation. Une telle méthodologie est indispensable pour définir un système SLAM performant. Elle consiste à établir un graphe définissant l'ordre des opérations et les dépendances de données d'un algorithme et un graphe définissant l'architecture cible. Ces deux graphe sont transformés pour générer un graphe d'implémentation adéquate. Enfin, le graphe d'implémentation est transformé en exécutif par des outils de développement et de conception qui seront aussi présenté dans ce chapitre. Finalement, des méthodes d'évaluation sont définies pour valider le comportement du système de SLAM conçu.

Le troisième chapitre sera dédié à la définition algorithmique des quatre algorithmes SLAM choisis. Pour ce faire, on étudie de manière détaillée le formalisme mathématique caractérisant chaque algorithme. Ensuite, nous donnons une étude plus approfondie de chaque algorithme SLAM. Pour ce faire, chaque algorithme est étudié en termes d'opération et d'ordre d'instructions ce qui permet son découpage en blocs fonctionnels exécutant chacun une tâche bien spécifique. Chaque bloc fonctionnel est ensuite évalué sur plusieurs architecture embarquées afin déterminer son temps de calcul. Après avoir choisi et étudié l'algorithme, son aspect fonctionnel tant bien que son aspect système est validé en utilisant les méthodes définies dans le deuxième chapitre.

Le dernier chapitre sera consacré à l'adéquation algorithme architecture du FastSLAM2.0 monoculaire sur des architectures massivement parallèle. En effet, on étudie la promesse tenue par les architectures parallèles pour l'accélération du FastSLAM2.0 monoculaire modifié et adapté aux environnements larges. Enfin, une comparaison entre ces architectures est présentée afin de définir l'architecture la plus adaptés à ce problème.

Finalement, nous conclurons sur les contributions proposées au cours de ce travail de thèse, ainsi que sur les différentes voies de recherches en perspectives.

Chapitre 1

Algorithmes SLAM et architectures dédiées : état de l'art

Sommaire

1.1 Localisation et cartographie simultanées

Le problème de localisation et cartographie simultanées (SLAM) traite deux questions importantes dans la robotique mobile. La première question est : Où suis-je ? La réponse à cette question définit la localisation du robot. La deuxième question concerne les caractéristiques de l'environnement du robot : À quoi ressemble l'environnement où je me trouve ?

Avec un système SLAM, un véhicule robotisé placé dans un environnement inconnu doit construire la carte de l'environnement tout en essayant de se localiser par rapport à cette carte. Le robot dispose de plusieurs capteurs qui l'aident à récupérer les informations dont il a besoin. La réalisation de cette tâche peut paraître impossible dans la mesure où le robot a besoin d'une carte pour se localiser, mais en même temps il doit connaître sa position pour pouvoir construire la carte. Afin de faciliter le traitement de ce problème concurrent , les chercheurs ont formulé les deux problèmes précédents en un seul problème : Où suis-je susceptible d'être dans la carte la plus probable de l'environnement que j'ai observé jusqu'à maintenant ?

La localisation et la cartographie simultanées sont indispensables dans le domaine de la robotique, en particulier pour la robotique mobile. Le SLAM est une tâche préliminaire à de nombreux autres domaines tels que la navigation autonome ou la planification de trajectoire. En effet, obtenir une modélisation réaliste d'un environnement permet au robot d'augmenter son autonomie en prenant des décisions en fonction du contexte général.

Dans les cas simples, l'état du robot est décrit par sa position et son orientation. D'autre paramètre peuvent être utilisés pour décrire l'état du robot tel que la vitesse du robot et les paramètres d'étalonnage de la camera utilisée comme capteur. D'autre part, la carte est une représentation des points distinctifs de l'environnement ou tout simplement les points d'intérêts (par exemple, position des points de repère ou des obstacles) décrivant l'environnement dans lequel le robot opère.

Le besoin de construire une carte de l'environnement comporte deux volets. Premièrement, la carte est souvent nécessaire pour supporter d'autres tâches. Par exemple, une carte peut orienter la planification de chemin ou fournir une visualisation intuitive pour un opérateur humain. Deuxièmement, la carte permet de limiter l'erreur commise dans l'estimation de l'état du robot. En l'absence d'une carte, l'état du robot serait rapidement dérivé au fil du temps. En revanche, étant donné une carte, le robot peut mettre en cause son erreur de localisation en revisitant les zones connues (fermeture de boucle). Par conséquent, le SLAM se trouve dans des applications et des scénarios dans lesquels une carte préalable n'est pas disponible et doit être construite.

Dans certaines applications de robotique, la carte de l'environnement est connue à priori. Par exemple, un robot fonctionnant sur une étage d'usine peut être équipé d'une carte construite manuellement par des amers artificiel invisibles dans l'environnement. Un autre exemple est le cas où le robot a accès aux données GPS (les satellites GPS peut être considéré comme des amers mobiles dont l'emplacement est connu).

La popularité du SLAM est liée à l'émergence d'applications de la robotique mobile en environnement interne. Ce dernier, exclut l'utilisation du GPS qui borne l'erreur de localisation. En outre, le SLAM offre une alternative attrayante pour la construction des cartes, montrant que le fonctionnement du robot est possible en l'absence d'une infrastructure de localisation ad-hoc.

Un examen historique approfondi des 20 premières années du problème SLAM est donné par Durrant-Whyte et Bailey dans deux enquêtes [11], [12]. Ceux-ci portent principalement sur ce que nous appelons l'âge classique des SLAM (1986-2004). Cet âge a vu l'introduction des principales formulations probabilistes pour le SLAM, y compris les approches fondées sur le filtre de Kalman étendu, le filtre particulaire de Rao-Blackwellised et l'estimation du maximum de la vraisemblance. En

outre, il a délimité les défis fondamentaux liés à l'efficacité et la mise en correspondance robuste. Deux autres références décrivant les trois formulations principales du SLAM de l'âge classique sont le chapitre de Thrun et Leonard [13], le chapitre [14] et le livre de Thrun, Burgard et Fox [15]. La période suivante est appelée l'âge de l'analyse algorithmique (2004-2015) qui est partiellement couverte par Dissanayake et coll. [16]. La période d'analyse algorithmique a vu l'étude des propriétés fondamentales du SLAM, y compris l'observabilité, la convergence et la consistance. Durant cette période, les contraintes critiques du SLAM étaient également comprises et les principales bibliothèques du SLAM open source ont été développées. Le tableau 1.1 récapitule les références récentes sur l'historique du SLAM pendant les 10 dernières années.

Année	Rubrique	Référence
2006	Approche probabiliste et mise en correspondance	Durrant-Whyte and Bailey [11], [12]
2008	Approche de filtrage	Aulinas et al [17]
2008	Le SLAM Visuel	Neira et al [18]
2011	Le SLAM back-end	Grisetti et al [19]
2011	Observabilité, Consistance et convergence	Dissanayake et al [16]
2012	Odométrie Visuelle	Scaramuzza and Fraundofer [20], [21]
2016	Le SLAM multi-robot	Saeedi et al [22]
2016	Reconnaissance des places	Lowry et al [23]

Table 1.1: *Références sur l'historique du SLAM*

1.2 Nécessité du SLAM pour la navigation autonome

Le SLAM vise à construire une représentation consistante de l'environnement s'appuyant sur les mesures du mouvement et la fermeture de boucle. Le mot clé « fermeture de boucle » a une grande importance. Si nous sacrifions la fermeture de boucle, le SLAM se réduit à l'odométrie. Dans les premières applications, l'odométrie a été obtenue par des codeurs de roues. L'estimation de position obtenues par les données odométriques se décale rapidement, ce qui augmente l'erreur de localisation après quelques mètres [[24], Ch. 6]. Ceci était l'une des raisons principales du développement du SLAM : l'observation des amers est utile pour réduire les erreurs odométriques et corriger éventuellement la trajectoire [25]. Cependant, des algorithmes d'odométrie qui sont apparus récemment et qui reposent sur l'information visuelle et inertielle, ont une erreur de glissement très faible (< 0.5 % de la longueur de la trajectoire [26]). Par conséquent, la question qui s'imposent : avons-nous vraiment besoin de SLAM ?

Tout d'abord, les recherches qui ont été menées sur le SLAM pendant ces dix dernières années, étaient l'élément principal produisant les algorithmes d'odométrie visuelle et inertielle qui représentent actuellement l'état de l'art [[26], [27]]. Dans ce sens, la navigation visuelle et inertielle (NVI) est un SLAM en elle même. Autrement dit, les algorithmes NVI peuvent être considérés comme un système de SLAM réduit dans lequel le module de la fermeture boucle (ou la reconnaissance de la place) est désactivé. Plus généralement, le SLAM a conduit à étudier la fusion de données de capteurs sous des contraintes plus difficiles (sans GPS, capteurs bas-coût de faible qualité).

La deuxième réponse concerne les fermetures de boucle. Un robot effectuant l'odométrie et négligeant la fermeture de boucle interprète le monde comme un couloir infini (Fig. 1.1-a) dans lequel le robot continue à explorer de nouveaux domaines indéfiniment. Un événement de fermeture de boucle

informe le robot que ce couloir s'intersecte en lui-même (Fig. 1.1-b). L'avantage de la fermeture de boucle devient maintenant évident : en trouvant une fermeture de boucle, le robot comprend la topologie réelle de l'environnement et est en mesure de trouver des raccourcis entre différents sites (par exemple, le point B et C dans le plan). Par conséquent, la topologie de l'environnement est l'un des avantages du SLAM. Dans ce sens, le SLAM pourrait être un principe redondant de la reconnaissance de place (un module de reconnaissance de place suffirait pour construire une carte topologique). En revanche, des scènes similaires, correspondant à des emplacements distincts dans l'environnement, tromperait le module de reconnaissance des lieux, alors que le SLAM offre une défense naturelle contre le mauvais appariement des amers et les données aberrantes. En ce sens, la carte fournit par le SLAM constitue un moyen de prévoir et de valider les mesures futures.

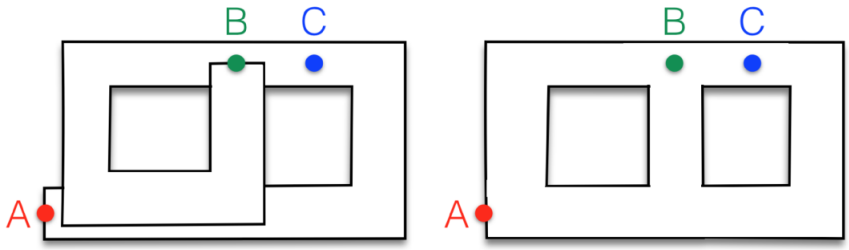


Figure 1.1: *Cartographie d'un couloir par le SLAM*

En conséquence, les algorithmes SLAM sont nécessaires car de nombreuses applications nécessitent, implicitement ou explicitement, une carte consistante reconstruite par rapport à l'échelle du monde. Par exemple, dans de nombreuses applications civiles et militaires, le but du robot est d'explorer un environnement et de rendre une carte à l'opérateur humain. Un autre exemple est le cas dans lequel le robot doit effectuer une inspection structurale (d'un bâtiment, pont, etc.). Dans ce cas, une reconstruction 3D consistante est une exigence pour la réussite de la tâche de l'opération Figure 1.2.

Figure 1.2: *Quelques applications du SLAM dans la robotique*

1.3 Le problème SLAM est-il résolu ?

Cette question est relativement souvent posée au sein de la communauté de robotique. La difficulté de répondre à cette question réside dans les cas d'applications. Le SLAM est un sujet aussi vaste que la question est bien posée pour des cas bien déterminés. En particulier, la maturité d'un algorithme de SLAM peut être évaluée en tenant en compte les aspects suivants :

— Robot : le type du mouvement (dynamique, la vitesse maximale), les capteurs disponibles (résolution, taux de rééchantillonnage), les ressources de calculs disponibles.
— Environnement : surface plane ou milieu en trois dimensions, présence de monuments naturels ou amers artificiels, quantité d'éléments dynamiques.
— Performance requise : la précision souhaitée dans l'estimation de l'état du robot, la précision et le type de représentation de l'environnement, le taux de réussite (pourcentage de tests dans lesquels les limites de précision sont remplies), l'estimation de latence, temps d'exécution maximal, la taille maximale de la zone cartographiée.

Pour le moment, les algorithmes de SLAM qui sont considérés résolus sont décrit dans les cas suivants :

— la cartographie d'une zone intérieur 2D avec un robot équipé de codeurs de roue et un scanner laser, avec une précision suffisante (<10cm) et robustesse suffisante (taux d'échec bas), peut être considéré résolue (un exemple de système industriel effectuant ce type de SLAM est la Solution de Navigation implantée sur Kuka [28]).
— Un SLAM basé sur la vision dans le cas d'un robot se déplaçant lentement (Mars rovers [29], robots domestiques [30])
— Un SLAM basé sur l'odométrie visuelle et inertielle (NVI) [31] peut être considéré comme domaine de recherche mature.

En revanche, d'autres cas d'applications du SLAM méritent encore un grand effort de recherche fondamentale. Les algorithmes de SLAM actuels peuvent facilement échoués lorsque le mouvement du robot ou l'environnement sont trop difficiles. De même, les algorithmes SLAM sont souvent incapables d'affronter les exigences strictes en termes de performances comme l'aspect temps réel.

1.4 Formalisation d'un système de SLAM moderne

L'architecture d'un système de SLAM moderne comprend trois composantes principales : la phase d'acquisition des données capteurs, la phase de prétraitement et la phase de traitement (cœur) du SLAM. Les données peuvent être acquises par deux types de capteurs : proprioceptif et extéroceptif. La partie prétraitement se résume dans des modèles mathématiques pour interpréter les données capteurs tandis que le cœur SLAM effectue une inférence des données abstraites produites par la deuxième phase. Cette architecture est résumée sur la figure 1.3.

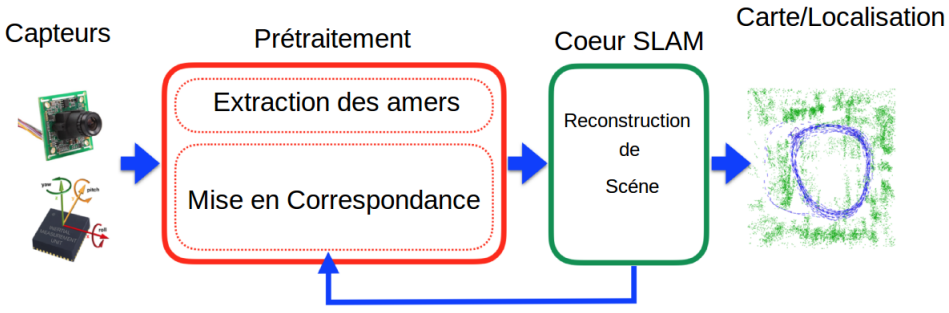


Figure 1.3: *Système de SLAM moderne*

1.4.1 Phase d'acquisition des données capteurs

1.4.1.1 Capteur proprioceptif : odomètres

Un encodeur est un dispositif électromécanique qui génère un signal électrique en fonction de la position de l'élément en déplacement. En robotique mobile, les encodeurs rotatifs sont utilisés pour mesurer le déplacement (sens et vitesse de rotation) de chacune des roues du robot. Un encodeur permet d'obtenir une information en quasi-temps réel sur la position. Sur un robot mobile à deux roues (conduite différentielle), la mesure effectuée permet de déduire le déplacement du robot, on parle d'odométrie. On distingue deux grands types d'encodeurs : incrémentaux et absolus. Un encodeur incrémental génère un signal permettant de déterminer le sens et la vitesse de rotation tandis qu'un encodeur absolu génère une information absolue indiquant la position du capteur. On considérera l'encodeur incrémentale en conjonction avec d'autres capteurs pour l'implémentation de notre algorithme SLAM. Le fonctionnement du capteur incrémental est décrit dans l'annexe 7.1, section 7.1.1.

1.4.1.2 Capteurs extéroceptif

Capteur de profondeur : Laser Les télémètres laser sont parmi les capteurs les plus populaires dans le domaine de la robotique mobile qui mesurent la profondeur des objets dans la scène. Il mesurent la distance qui sépare le centre cinématique du robot mobile des objets (amers ou point d'intérêts) dans l'environnement exploré. Ils sont faciles à modéliser mathématiquement mais leur utilisation sur une plateforme expérimentale pour l'exploration reste coûteuse et nécessite la mise en place des algorithmes de filtrage et d'extraction de données utiles à partir d'un flux laser brute. Le principe de fonctionnement de ce type de capteur est décrit dans l'annexe 7.1, section 7.1.2.

Capteur projectif : Camera monoculaire Une camera est un capteur de vision projetant un objet à trois dimensions dans la scène sur un plan à deux dimensions. Dans le domaine de la vision par ordinateur, la camera nécessite un système de calibrage avant son utilisation pour une tâche visuelle précise. Une fois la caméra est calibrée, ses paramètres doivent rester fixes en cours d'utilisation. Chaque fois que l'on désire modifier la mise au point, la distance focale voire l'ouverture de l'objectif, la caméra devra être recalibrée. Ces vingt dernières années, de très nombreux travaux ont permis d'obtenir des méthodes plus ou moins complexes et donc plus ou moins précises permettant de calibrer la caméra. Il existe plusieurs méthodes pour la modélisation d'une caméra monoculaire, nous utiliserons le modèle Pin-Hol détaillé dans les sections suivantes.

1.4.2 Phase de prétraitement :

Dans les applications robotiques, il pourrait être difficile de décrire les mesures capteurs directement comme une fonction analytique demandée par le traitement principal (cœur) du SLAM. Dans le cas où les données brutes du capteur sont des images, il pourrait être difficile d'exprimer l'intensité de chaque pixel en fonction de l'état du cœur SLAM. La même difficulté s'impose avec des capteurs plus simples (un laser avec un faisceau unique par exemple). Dans les deux cas, le problème est lié au fait que nous ne sommes pas en mesure de concevoir une représentation suffisamment générale de l'environnement. Même en présence d'une telle représentation, il serait difficile d'écrire une fonction analytique qui relie les mesures (observations) aux paramètres de cette représentation. Pour cette raison, avant l'exécution du cœur SLAM, il est fréquent d'avoir un module à priori, la phase de prétraitement, permettant d'extraire des points d'intérêts à partir des données capteurs. Dans un système SLAM basé sur la vision, la phase de prétraitement extrait l'emplacement du pixel de quelques points identifiables dans

l'environnement. Ces observations de pixels sont maintenant faciles à modéliser par le cœur SLAM. La phase de prétraitement est également en charge de la mise en correspondance d'une mesure avec un amer spécifique dans l'environnement (point 3D) ; il s'agit d'une opération d'appariement. Dans ce cas, l'appariement cherche à identifier une observation z_k dans un ensemble de variables inconnues χ_k tel que $z_k = h_k(\chi_k) + \epsilon_k$. Enfin, la phase de prétraitement peut également fournir une estimation initiale des variables. Par exemple, dans un SLAM monoculaire la partie prétraitement peut s'occuper de l'initialisation des amers par des méthodes de triangulation. Le type de calcul réalisé dans la partie prétraitement dépend fortement du type du capteur utilisé puisque la notion des amers change d'un capteur à l'autre.
Dans cette thèse nous étudierons plusieurs algorithmes SLAM utilisant différents types de capteurs. Dans la section suivante nous donnons le formalisme mathématique de la phase de prétraitement en fonction du type de capteur.

1.4.2.1 Modèle d'évolution

La cinématique du robot a été très étudiée au cours des dernières décennies. La robotique probabiliste généralise les équations cinématiques du fait que l'acquisition des données est bruitée. La cinématique est le calcul décrivant l'effet des mesures de contrôle sur la configuration d'un robot. La configuration d'un robot mobile à roues est souvent décrite par six variables, ses coordonnées cartésiennes tridimensionnelles (x, y, θ) et ses trois angles d'Euler (roll, pitch, yaw) par rapport à un repère externe. Tout au long de notre étude, nous considérons un robot à roues se déplaçant dans un repère à deux dimensions, dont les variables cinématiques sont résumées par trois variables (x, y, θ). Ceci est illustré dans la figure 1.4. La position du robot est décrite par deux cordonnées cartésiennes (x, y) relatives à un repère externe, avec son orientation angulaire θ. Dorénavant, on désigne par s_t le vecteur d'état du robot : $s_t = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$.

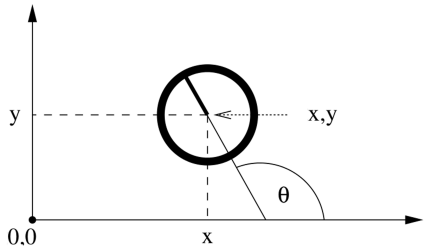


Figure 1.4: *Position du robot mobile dans le repère globale*

Le modèle cinématique probabiliste, ou bien le modèle du mouvement est un modèle de transition d'état en robotique mobile. Ce modèle s'agit de la densité conditionnelle 1.1. s_t et s_{t-1} sont les vecteurs de position du robot, u_t est la loi de contrôle du mouvement. Ce modèle décrit la densité de probabilité postérieur d'une transition d'état en partant d'une position initiale s_{t-1} et en exécutant le contrôle u_t.

$$p(s_t|u_t, s_{t-1}) \tag{1.1}$$

La figure 1.5 illustre deux exemples du modèle de mouvement d'un robot mobile à roues dans un

environnement bidimensionnels. Dans les deux cas la position initiale du robot est s_{t-1}. La densité de probabilité $p(s_t|u_t, s_{t-1})$ est représentée en deux dimensions par la zone grise : plus la zone est sombre plus la position est probable. Dans la figure 1.5.a, le robot fait un déplacement horizontale tout au long de l'axe x, au cours duquel il peut s'accumuler des erreurs de translation et de rotation comme indiqué dans la figure. La figure 1.5.b montre la distribution probabiliste résultante d'un déplacement un peu plus complexe qui engendre une grande incertitude sur la position. Dans la littérature on trouve plusieurs modèles probabilistes de mouvement pour des robots mobiles fonctionnant dans un environnement bidimensionnels. On trouve principalement le modèle de vitesse et le modèle odomètrique. Ces deux modèles sont plus au moins complémentaires dans le type de contrôle utilisé. Le premier modèle suppose que la donnée du contrôle u_t spécifie la vitesse des moteurs du robots. Le deuxième modèle suppose que le contrôle u_t spécifie les données odomètriques des roues droite et gauche du robot. Le modèle du mouvement probabiliste résultant en intégrant ces informations du contrôle est plus au moins différent. La plus part des industriels utilisent des capteurs odomètriques au lieu des capteurs de vitesse dans leur plateformes. En pratique, le modèle odomètrique est beaucoup plus précis que le modèle de vitesse, pour le simple prétexte que la plus part des robots commerciaux ne peuvent pas traiter l'information de vitesse avec un tel niveau de précision qui peut être achevé par la mesure du nombre de rotation des roues du robot mobile. Bien évidemment, les deux modèles souffrent des erreurs de glissement et de dérivation. Le modèle de vitesse souffre de plus des écarts entre le contrôle et son modèle mathématique. En conséquence, le modèle odomètrique est relativement souvent utilisé avec des algorithmes d'estimation type SLAM. C'est ce modèle qu'on va donc considérer dans l'implantation des algorithmes SLAM.

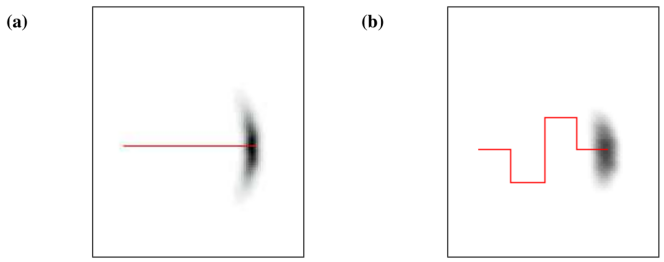


Figure 1.5: *Le modèle du mouvement*

La position initiale du robot est donnée par (x, y, θ), d_l et d_r définissent la distance parcourue respectivement par la roue gauche et droite du robot. Le modèle d'évolution calcule la position future du robot $\left(x^{'}, y^{'}, \theta^{'}\right)$, il est donné par les équations suivantes :

$$\boldsymbol{f}(x', y', \theta') = \begin{pmatrix} x' = x + d_c cos\left(\theta + \frac{\delta_\theta}{2}\right) \\ y' = y + d_c sin\left(\theta + \frac{\delta_\theta}{2}\right) \\ \theta' = \theta + \delta_\theta \end{pmatrix} \tag{1.2}$$

avec

$$\begin{aligned} d_c &= \tfrac{d_r + d_l}{2} \\ \delta_\theta &= \tfrac{d_r - d_l}{d_b} \end{aligned} \tag{1.3}$$

1.4.2.2 Modèle d'observation

Les robots aujourd'hui utilisent une variété de capteurs de perception, comme les capteurs de profondeur ou bien une camera. Les caractéristiques de chaque modèle dépendent fortement du type de capteur utilisé : les capteurs d'image sont bien modélisés par une géométrie projective, les capteurs sonores sont modélisés par la description des caractéristiques de l'onde sonore et la réfraction sur le milieu ambiant. La méthode probabiliste modélise explicitement le bruit de mesures. Ces modèles représentent l'incertitude inhérente sur les capteurs qui équipent le robot. Formellement, le modèle d'observation est défini comme étant une distribution probabiliste conditionnelle $p(z_t|s_t, m_t)$, ou s_t est la position du robot, z_t l'observation à l'instant t et m_t la carte de l'environnement. Bien que nous nous adressons principalement dans cette section aux deux types de capteurs, à savoir les télémètres laser et les cameras monoculaires, le principe sous-jacent et les équations mathématiques ne sont pas limités à ces types de capteurs. Le principe de base peut être appliqué à n'importe quel type de capteur, exemple : le code à barres pour la détection des points d'intérêts.

Pour illustrer le problème de la perception de l'environnement, la figure 1.6 montre un robot mobile équipé de plusieurs capteurs sonores et entrain d'explorer un environnement interne. La distance mesurée par chaque capteur sonore est représentée par une ligne noire et la carte reconstruite est montrée par des couloirs en noirs. La perception de l'environnement consiste à mesurer la distance qui sépare le robot des objets les plus proches qui l'entourent. Les données aberrantes qui constituent le bruit de mesure sont souvent interprétées par l'incapacité du capteur à mesurer la distance des objets proches. Ce bruit est prévisible et va être calculé pour en tenir compte dans l'implémentation de l'algorithme SLAM. En règle générale, plus le modèle d'observation est précis plus les résultats sont cohérents. Bien évidemment, il est presque impossible d'avoir un modèle exacte des capteurs extéroceptifs et souvent le développement d'un tel modèle nécessite la connaissance de certains paramètres qui sont imprévisibles. Donc les algorithmes résolvent le problème d'inexactitudes du modèle des capteurs extéroceptifs dans l'approche stochastiques : en modélisant ce modèle par une densité probabiliste conditionnelle $p(z_t|s_t, m_t)$ au lieu d'une fonction simple déterministe. Les capteurs extéroceptifs qu'on va étudier (laser, camera) génèrent plus d'une seule information. Une camera génère un tableau de valeurs (intensité, saturation, couleurs). Le télémètre laser génère une gamme de distances mesurées. On va désigner par z_t^k une seule observation acquise et par z_t un ensemble d'observations.

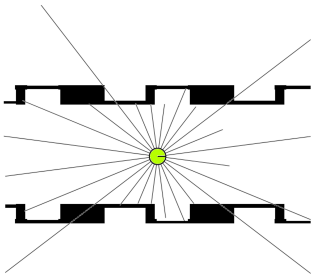


Figure 1.6: *Exemple illustratif de la perception dans un environnement*

Modèle d'observation à base d'un télémètre Laser Il existe des modèles d'observations qui sont basés sur les mesures brutes du capteur extéroceptif. Comme approche alternative, il existe des modèles qui s'appuient principalement sur un seul point d'intérêt extrait des données brutes mesurées. Dans notre implantation de l'algorithme SLAM on considère la deuxième approche. Un tel modèle est avantageux parce qu'il réduit le coup et la complexité de calcul par rapport au premier modèle

qui tient en compte toutes les données brutes issues du capteur. L'utilisation d'un tel modèle est adapté dans le cas où le fonctionnement en temps-réel est un facteur important. Ce modèle nécessite la mise en place d'un algorithme d'extraction de primitives à partir des données brutes. L'extraction des primitives (points d'intérêts) est étudiée dans la section suivante. Dans cette partie nous nous intéressons à la forme analytique du modèle d'observation.

Dans la plus part des applications robotiques, un point d'intérêt correspond à un objet distinctif dans le monde physique. Dans un environnement interne, un point d'intérêt peut être une porte ou une fenêtre. Dans un environnement externe, il peut être un tronc d'arbre ou un coin d'un bâtiment. Dans la robotique mobile ces points d'intérêts sont utilisés comme des amers par le robot pour naviguer et se repérer dans l'environnement à explorer. Le télémètre LASER peut donner une information sur la profondeur aussi bien que l'angle des points d'intérêts relatifs à la position du robot dans le repère local. On désigne par r la profondeur et ϕ par l'angle, le vecteur de position du point d'intérêt est donné par (1.4) :

$$f\left(z_t\right) = \left\{z_t^1, z_t^2, \ldots\right\} = \left\{\begin{pmatrix} r_t^1 \\ \phi_t^1 \end{pmatrix}, \begin{pmatrix} r_t^2 \\ \phi_t^2 \end{pmatrix}, \ldots\right\} \tag{1.4}$$

Le nombre des points d'intérêts détectés à chaque instant est variable et dépend du type de l'extracteur utilisé. Le modèle d'observation qu'on va utiliser suppose que la carte m construite par le robot s'agit d'un ensemble de points d'intérêts de telle sorte que $m = \{z_1, z_2, \ldots\}$. La position du point d'intérêt est notée par (x_f, y_f) où x_f et y_f sont les cordonnées du point dans le repère global. La figure 1.7 illustre le modèle de perception du robot. r, ϕ sont les mesures données par le capteur. Le modèle d'observation à base du capteur LASER est formulé en utilisant les lois géométriques classiques (1.5).

$$\mathbf{h}_l = \begin{pmatrix} r_t^i = \sqrt{\left(x_f - x\right)^2 + \left(y_f - y\right)^2} \\ \phi_t^i = \arctan\left(\frac{y_f - y}{x_f - x}\right) - \theta \end{pmatrix} + \begin{pmatrix} \epsilon_{\sigma_r^2} \\ \epsilon_{\sigma_\phi^2} \end{pmatrix} \tag{1.5}$$

où $s_t = (x, y, \theta)$ est la position du robot, $\left(\epsilon_{\sigma_r^2}, \epsilon_{\sigma_\phi^2}\right)$ est l'erreur Gaussienne moyenne respectivement sur la profondeur et l'angle mesurés par le capteur, avec une variance $\sigma_r^2, \sigma_\phi^2$.

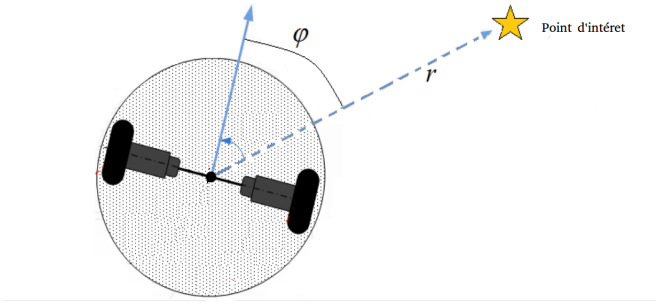


Figure 1.7: *Perception de l'environnement avec un capteur Laser*

Modèle d'observation à base d'une caméra monoculaire Il existe plusieurs méthodes de projection d'image pour modéliser une camera. Dans notre étude nous optons pour la projection perceptive pour sa meilleure adéquation à la réalité physique du capteur de vision. Le modèle sténopé (Pin-Hole en anglais) modélise une caméra par une projection perspective. Ce modèle transforme un point 3D de l'espace en un point dans l'image et peut se décomposer en trois transformations

élémentaires successives (1.8) : la transformation entre le repère du monde et celui de la caméra , la transformation entre le repère caméra et le repère capteur et la transformation entre le repère capteur et le repère image.

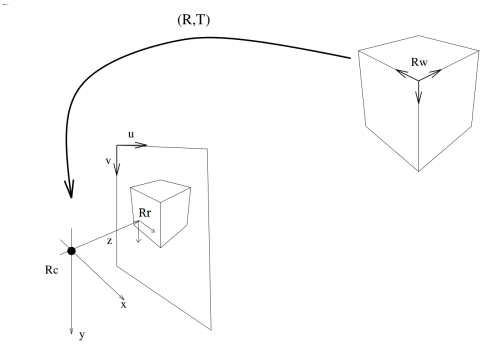


Figure 1.8: *modèle sténopé*

Le modèle d'observation (Pin-Hole) à base d'une camera décrit la relation entre les coordonnées d'un point (u, v) sur l'image et sa position $(x^{cam}, y^{cam}, z^{cam})$ dans le repère caméra. Il est donné par l'équation 1.7.

$$\begin{pmatrix} su \\ sv \\ s \end{pmatrix} = \begin{bmatrix} fk_u & fs_{uv} & c_u \\ 0 & fk_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \end{pmatrix} \tag{1.6}$$

Avec :

— (u, v) : la position du point sur l'image .
— f : la distance focale en mm.
— (k_u, k_v) : les facteurs d'agrandissement de l'image.
— (c_u, c_v) : les coordonnées de la projection du centre optique de la caméra sur le plan image en pixel.
— s_{uv} : la non-orthogonalité des lignes et des colonnes de cellules électroniques photosensibles qui composent le capteur de la caméra. Dans notre cas, cette variable est considérée comme nulle.
— $(x_{cam}, y_{cam}, z_{cam})$: la position de (u, v) dans le repère caméra.

En faisant l'hypothèse que $s_{uv} = 0$, le modèle Pinhole devient :

$$\mathbf{h}_c = \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} c_u + fk_u \frac{x_{f,cam}}{z_{f,cam}} \\ c_v + fk_v \frac{y_{f,cam}}{z_{f,cam}} \end{pmatrix} \tag{1.7}$$

1.4.3 Cœur SLAM

La présente hiérarchie du SLAM a été introduite premièrement dans les travaux de Lu et Milios [32] suivi par les travaux de Gutmann and Konolige [33]. Le SLAM en général est formulé comme un problème d'estimation postérieur et peut être représenté par un graphe pour mettre en évidence l'indépendance entre les variables. Supposons qu'on voudrait chercher une estimation d'une variable χ . Dans le contexte du SLAM, χ est une variable qui comporte la trajectoire du robot (comme un ensemble de points discrets) et la position des amers dans la scène. Étant donnée un ensemble des observations $Z = \{z_k; k = 1, \cdots, m\}$de tel sorte que chaque observation peut être exprimée en fonction

de χ $z_k = h_k(\chi_k) + \epsilon_k$, où h_k est une fonction connue (modèle d'observation) et ϵ_k est l'erreur de mesure. Dans la phase du traitement du cœur SLAM, on calcule χ de telle sorte que la probabilité $p(\chi|Z)$ soit maximale. Selon la théorème de Bayes on peut écrire l'équation (1.8) :

$$\chi = argmax\left(p\left(\chi|Z\right)\right) = argmax\left(p\left(Z|\chi\right)p\left(\chi\right)\right) \tag{1.8}$$

Dans l'équation (1.8), $p(Z|\chi)$ est la probabilité d'observation de la mesure Z étant donnée la variable χ, $p(\chi)$ est la probabilité antérieure de χ. La probabilité antérieure comporte toute information antérieure sur la variable χ. Dans le cas du manque d'information antérieure, $p(\chi)$ devient une distribution probabiliste uniforme et peut être exclu de la factorisation.

Supposons que l'observation Z est indépendante, l'erreur sur la mesure n'est pas corrélée et donc l'équation 1.8 peut être factorisée comme suivant :

$$\chi = argmax(p(\chi)) \prod_{k=1}^{m} p\left(z_k|\chi\right) = p(\chi) \prod_{k=1}^{m} p(z_k|\chi_k) \tag{1.9}$$

L'équation 1.9 peut être être interprétée comme un graphe factorisé [34]. Les variables correspondent aux nœuds du graphe, les deux termes $p(z_k|\chi_k)$ et $p(\chi)$ sont appelés des facteurs. Ces facteurs encodent les contraintes probabilistes tout au long d'une séquence de nœuds. Le graphe factorisé est une représentation graphique qui encode les dépendances entre les facteurs et la variable χ correspondante.

Un premier avantage de l'interprétation par graphe factorisé et qu'elle permet une visualisation perspicace du problème. La figure 1.9 montre une interprétation d'un problème SLAM simple par une représentation de graphe factorisé. La figure montre les différentes variables dans un système SLAM, notamment la position du robot, la position des amers, les paramètres de calibration de la camera et les facteurs qui imposent les contraintes entre les différentes variables. Les cercles bleus sont les positions du robot dans des instants consécutif $(x_1, x_2, ...)$. Les cercles verts sont les positions des amers (l_1,l_2..), le cercle rouge correspond aux paramètres intrinsèques associés à la caméra (K). Les facteurs de graphe sont les points en noir : la notation u est le facteur correspondant aux contraintes d'odométrie, la notation v est le facteur correspondant aux contraintes de la camera. Finalement, le facteur c correspond aux contraintes de la fermeture de boucle et p correspond au facteur antérieur. Comme deuxième avantage, le modèle de graphe factorisé sert à donner une représentation général d'un problème aussi complexe intégrant plusieurs facteurs et variables hétérogènes.

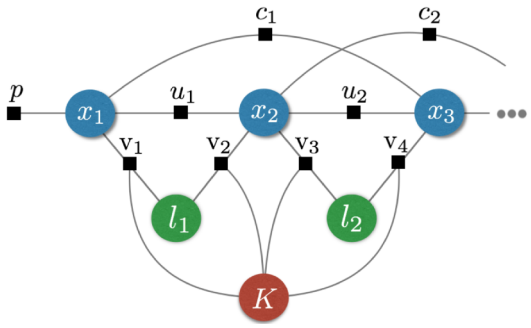


Figure 1.9: *Graphe factorisé d'un système de SLAM*

1.5 Résolution du SLAM

1.5.1 L'approche probabiliste

La plupart des solutions SLAM rapportées à ce jour sont basées sur des techniques probabilistes. Certaines d'entre elles sont : le filtre de Kalman étendu (EKF), la solution factorisée du SLAM (FastSLAM), la probabilité maximale (ML) et l'espérance de maximisation (EM) [35]. Les deux premières techniques sont les plus couramment utilisées parce qu'elles offrent les meilleurs résultats et minimisent les incertitudes de l'entité et la carte. Ces approches conviennent aux environnements à petites tailles, mais ont une capacité limitée pour explorer les grands environnements ou exécuter une fermeture de boucle.

Une méthodologie pour la construction de cartes de manière incrémentale (causale), a été présentée dans les travaux de Smith et al., [36]. Smith et coll. (1990) a introduit le concept de carte stochastique et a développé une solution précise pour le problème SLAM en utilisant le filtre de Kalman étendu. L'approche basée sur l'EKF est caractérisée par un vecteur d'état composé de la localisation des amers de la carte estimée de manière récursive à partir des modèles d'observation et d'évolution non linéaires. L'incertitude est représentée par une fonctions de densité probabiliste (PDF). L'EKF est particulièrement sensible aux mauvaises associations. Une mesure incorrecte peut conduire à la divergence du filtre en entier. La complexité de l'EKF est quadratique en fonction du nombre des amers sur la carte. Son utilisation est donc difficile dans des grands environnements. Dans la littérature, il existe différentes méthodes pour réduire cette complexité au moyen de certaines techniques telles que : Atlas Framework [37], EKF compressé (CEKF) [38], filtre d'information étendu (SEIF) [35] ou la méthode deviser pour régner proposée par [39].

FastSLAM a été proposée par Montemerlo et al., [40] et amélioré plus tard en [41]. Cette méthode gère la position du robot comme une distribution d'un ensemble de particules, où chaque particule représente une trajectoire précise, maintient sa propre carte construite à l'aide de l'EKF, possède une hypothèse sur l'identification d'un amer (appariement) et survit avec une probabilité définie (poids). L'algorithme se compose d'un processus de génération de particules et d'un processus de ré-échantillonnage afin d'éviter la dégénérescence des particules au fil du temps. La complexité de cet algorithme est logarithmique, $M\,(p \log\,(n))$, où M est le nombre de particules utilisées et n est le nombre des amers sur la carte. Le problème majeur de l'approche est qu'il n'y a aucun moyen de déterminer le nombre de particules nécessaires pour représenter avec précision l'état du système. Autrement dit, plusieurs particules nécessite une mémoire et un temps de calcul importants tendis que l'utilisation de quelques particules conduit à des résultats inexacts.

Davison [42] a été le premier à présenter un système probabiliste en temps réel appelé MonoSLAM. Cette technique de SLAM effectue simultanément une cartographie 3D des amers à partir de 30 images par seconde en utilisant seulement un appareil photo numérique firewire (IEEE-1394). Ce travail a des limitation : il ne peut fonctionner que dans des espaces confinés et des environnements intérieurs puisqu'il emploie l'EKF pour l'estimation des amers. Le système MonoSLAM utilise un modèle de mouvement linéaire constant et considère que la vitesse est angulaire. Ceci présente un inconvénient qui réside dans l'incapacité du modèle de traiter correctement les mouvements brusques, limitant la mobilité de la caméra.

Pour faire face à des mouvements erratiques de la caméra dans le MonoSLAM, [43] a mis au point une version optimisée, capable de fonctionner à 200 Hz en utilisant un modèle de mouvement étendu qui prend en compte l'accélération et la vélocité linéaire et angulaire. Cependant, ses performances en temps réel sont limitées seulement dans quelques secondes, car la taille de la carte et les coûts de

calcul augmentent extrêmement vite.

Pour augmenter le nombre des amers dans la carte sans perdre les performance temps réel, Eade et Drummond [44] ont utilisé une technique de filtrage particulaire inspirée de la méthode proposée par Montemerlo et coll. [40]. La méthode de Eade et Drummond est capable de suivre jusqu'à 30 amers par image et maintenir des milliers d'amers dans la carte. Clemente et coll. [45] proposent une solution alternative d'utiliser le MonoSLAM dans les grands environnements extérieur. Cette approche repose sur une technique de cartographie hiérarchique et un algorithme d'appariement robuste capable d'effectuer de grandes fermeture de boucles (250 m environ).

Comme mentionné ci-dessus, le problème majeur dans un SLAM visuel monoculaire réside dans l'initialisation des amers, car leur profondeur ne peut pas être calculée à partir d'une seule observation. Pour ce faire, Davison [46] utilise une technique d'initialisation avec retard, tandis que Montiel et coll. [47] proposent une technique appelée paramétrisation par inverse de profondeur qui effectue une initialisation non-retardée des amers dans un système SLAM.

1.5.2 Approche de structuration à partir du mouvement

La technique de structuration à partir du mouvement (SfM) permet de calculer la structure 3D de la position de la caméra et de la scène à partir d'un ensemble d'images [48]. Elle a ses origines dans les domaines de vision par photogrammétrie et ordinateur. La procédure standard consiste à extraire des amers à partir des images, les identifier par le processus d'appariement et d'effectuer une optimisation non linéaire appelée ajustement par faisceaux (Bundle Ajustement BA) afin de minimiser l'erreur de projection [49].

La technique SfM permet une localisation précise de la position de la caméra mais ne fournit pas nécessairement une représentation précise de la carte d'environnement. Malgré cela, plusieurs propositions ont été faites à l'aide SfM pour une localisation avec précision tout en créant une bonne représentation de l'environnement. Une des méthodes proposée pour résoudre le problème SLAM par SfM incrémental est l'introduction de l'odométrie visuelle par Nistér et al., [50]. L'odométrie visuelle consistent à déterminer simultanément la position de la camera à chaque image reçue et la position des amers en 3D, de façon incrémental et en temps réel. Mouragnon et coll. [51] utilise une odométrie visuelle semblable à la proposition de Nister, mais en y ajoutant une technique appelée ajustement par faisceau local, dans une trajectoire pouvant atteindre 500 m.

Klein et Murray [52] présentent une méthode monoculaire appelée suivi et cartographie parallèle (PTaM). Il utilise une approche basée sur des images avec deux tâches (threads) de traitements parallèles. Le premier thread exécute une tâche de suivi et localisation des amers, tandis que l'autre tâche produit une carte 3D corrigée par la technique BA. Le système PTaM souffre des échecs de suivi en présence des textures similaires et des objets mobiles.

[53, 54] et [55] proposent respectivement deux techniques appelées FrameSLAM et View-Based Maps. Les deux méthodes consistent à utiliser une représentation de la carte sous forme d'un squelette. Cette représentation s'agit d'un graphe de contraintes non linéaires entre les images (plutôt que des amers 3D individuels). Les auteurs utilisent une camera stéréo montée sur un robot à roues. Les performances du système sont prometteuse sur une longues trajectoire (approximativement 10 km) avec une variation de l'environnement.

Récemment Strasdat et al [56] a démontré que pour augmenter la précision d'un système SLAM monoculaire, il est recommandé d'augmenter le nombre des amers plutôt que le nombre d'images. Ainsi, des techniques d'optimisation par faisceau local sont nécessaires. Cependant, les filtres peuvent être bénéfique dans des situations où l'incertitude est grande. Un système SLAM idéal exploiterait les

avantages de la vision et les filtres probabilistes en même temps.

1.5.3 Approche bio-inspirée

Milford et coll.[57] utilisent des modèles de l'hippocampe (responsable de la mémoire spatiale) des rongeurs pour créer un système de localisation et de cartographie simultanée appelé RatSLAM. RatSLAM permet de générer des représentations cohérentes et stables d'un environnement complexe à l'aide d'une seule caméra. Des expériences menées dans ([58] ; [59]) montrent que le RatSLAM est capable d'opérer en temps réel dans des environnements interne et externe tout en donnant des localisation cohérentes. En outre, il a la capacité de fermer plus de 51 boucles de 5 km de longueur. Le travail dans [58] a présenté une étude approfondie sur le RatSLAM et d'autres systèmes biologiques de navigation chez les abeilles, les fourmis et les humains. Collett [60] examine le comportement des fourmis dans le désert pour analyser comment sont-ils guidés par des amers visuels. Bien que cette recherche se concentre sur la compréhension de comment les fourmis naviguent en utilisant des informations visuelles, l'auteur affirme que la solution proposée serait viable et facile à mettre en œuvre pour un robot.

1.6 Architectures de calcul embarqué pour le SLAM

Ces dernières années, des progrès majeurs ont été faits dans la conception des architectures pour le calcul des algorithmes SLAM. La plus-part de ces travaux s'intéressait à l'amélioration des performances de SLAM sans prendre en considération les problèmes qui s'imposent spécialement dans l'utilisation des architectures de faible puissance. Pour une implémentation efficace, il s'avère indispensable de prendre en compte l'aspect adéquation algorithme architecture. En effet, un système de SLAM consiste à utiliser une architecture embarquée récente aussi puissante que possible avec un algorithme SLAM optimisé et adéquat.

Les architectures massivement parallèles se développent de manière très importante depuis quelques années. Les GPUs (Graphics Processing Unit) et les FPGAs sont des exemples d'architectures massivement parallèles.

L'utilisation des GPUs, comme une plateforme pour le calcul générique de haute performance, est devenue populaire ces derniers décennies. Cependant, les GPUs reste très peu étudiés pour l'accélération des algorithmes SLAM. Les auteurs dans [61] ont présenté une méthodologie d'adéquation algorithme architecture pour implémenter la phase de calcul des poids d'importance et la mise à jour des particules sur un processeur NIOS II. Les auteurs dans [62] ont présenté une implémentation software du FastSLAM2.0 sur un processeur configurable et extensible VLIW (*Very Long Instruction Word*). [63] présenté une implémentation hétérogène d'un filtre particulaire adaptatif sur un FPGA et un CPU pour des applications de localisation et suivi de personnes. Cependant, il n'existe aucune implémentation complète et optimisée d'un algorithme SLAM sur une architecture hétérogène embarquée intégrant un CPU et un GPU. Dans l'état de l'art, on trouve [64] qui présente une implémentation parallèle du filtre particulaire sur une architecture multi-cœurs, tandis que [65] a proposé une architecture dédiée pour la phase de rééchantillonnage du filtre particulaire. Finalement, [66] a implémenté un filtre particulaire général sur un GPU de haute gamme. Le travail mené dans [67] s'est consacré à l'accélération du FastSLAM1.0 sur un GPU. Ils ont exploité le calcul générique sur un GPU NVIDIA pour aboutir à une implémentation parallèle. Cependant, ils ont juste proposé une implémentation parallèle de la phase de rééchantillonnage, le reste de l'algorithme s'exécute sur un CPU. En outre, les évaluations ont été réalisées sur un GPU de haute gamme (NVIDIA GeForce GTX

660) et un CPU (Intel Core i5-3570K).

Tertei en 2004[68] a accéléré un bloc de multiplication de l'EKF pour le SLAM visuel 3D. L'implémentation a été faite sur une architecture embarquée de faible puissance intégrant un processeur matériel reconfigurable. Le but de leur travail consiste à optimiser l'algorithme sur une architecture dédiée. Cependant, ils n'ont pas tenu compte du système SLAM dans sa globalité. En effet, l'algorithme EKF-SLAM n'est pas un choix adéquat pour résoudre le problème SLAM. La complexité de cet algorithme augmente linéairement avec le nombre des amers et n'est donc pas adapté pour l'utilisation en environnements larges. En outre, l'EKF-SLAM n'est pas adéquat pour tirer profit totalement de l'architecture parallèle massive du FPGA, comme la majeure de ses blocs sont séquentiels par nature.

Sileshi [69] a implémenté sur FPGA le FastSLAM1.0. Il a utilisé un jeu de données réelles d'un capteur laser avec des odomètres pour évaluer l'implémentation de l'algorithme sur FPGA. La plus part des blocs du FastSLAM1.0 sont implémentés sur un processeur Softcore Microblaze, y compris l'échantillonnage, le calcul du poids d'importance et le rééchantillonnage des particules. Les autres blocs sont accélérés sur le FPGA. Cependant, l'implémentation résultante n'est pas totalement accélérée sur FPGA. Certains blocs s'exécutent en séquentiel sur le Softcore. L'architecture hautement parallèle du FPGA n'est donc pas totalement exploitée. En outre, ils ont implémenté le FastSLAM1.0 qui souffre du problème de l'appauvrissement des particules [41]. Ainsi, un tel algorithme n'est pas un choix adéquat pour l'utilisation, plus particulièrement en environnement large.

Les auteurs dans [70] ont proposé un système de SLAM Visuel-Inertiel capable d'opérer dans des environnements larges. Ils ont implémenté l'algorithme résultant sur un GPU NVIDIA haut de gamme, TITAN NVidia, et un processeur Intel i7 quad-CPU de bureau. Leur contribution est intéressante puisqu'elle propose un nouveau algorithme capable de cartographier un environnement intérieur/extérieur à grande échelle. Cependant, la plus part des applications SLAM sont déployées dans des situations extrêmement difficiles où l'utilisation du calcul embarqué est obligatoire. Par conséquent, une évaluation d'un algorithme sur une architecture embarquée à faible puissance est nécessaire pour confirmer qu'un tel algorithme est capable de s'exécuter en temps réel.

Nardi et al [71] ont implémenté un algorithme de traitement d'images utilisé dans le SLAM. L'algorithme de prétraitement implanté est appelé Kinect Fusion. Ils ont utilisé différentes architectures embarquées de faible puissance pour évaluer l'implémentation de l'algorithme en utilisant des langages de programmation différents. Les auteurs se sont focalisés sur des processeurs ARM et des GPUs embarqués. Les architectures à base de FPGA n'ont pas été évaluées. Cependant, l'algorithme implanté ne peut être utilisé que pour le RGBD-SLAM, ce qui signifie que les résultats fournis sont spécifiques à ce type d'algorithme.

1.7 Synthèse

Le tableau 1.2 résume les systèmes récents de SLAM et leurs architectures de calcul. Depuis 2003, davison Davison [72] a introduit la première utilisation de la camera pour résoudre le problème de SLAM. Ce travail a montré la possibilité de réaliser le SLAM avec une vision monoculaire. Le système résultant est appelé le MonoSLAM et est capable d'opérer en temps réel sur un ordinateur de bureau. Ensuite, des modification algorithmique ont été introduite dans le MonoSLAM pour concevoir d'autres systèmes plus robustes. Jusqu'à 2006 ces algorithmes s'exécutaient toujours sur des stations de travail pour assurer l'aspect temps réel. En 2007 Klein et Murray [73], Klein a réalisé une implémentation du GraphSLAM sur un iphone. Ce travail a eu comme objectif de résoudre la problématique embarquabilité à l'aide des calculateurs bas-coût disposant de ressources limitées. Les systèmes de SLAM basés sur les approches bio-inspirées sont apparues depuis 2004 [57]. Leur implémentation est réalisée

toujours sur des stations de travail vu que les algorithmes sont complexe par nature. Un exemple d'implémentation en C est donnée dans [58] et une implémentation C++ sous ROS est décrite dans [3]. L'utilisation des architectures massivement parallèle est apparue depuis 2008. Ces architectures sont utilisées pour réaliser de nombreux calculs en parallèle. les auteurs dans Botero *et al.* [74] utilisent une architecture matérielle pour réaliser la détection des amers alors que les auteurs dans Bonato *et al.* [75] ont conçu une architecture pour paralléliser le calcul matriciel de l'EKFSLAM. [76] utilise un GPU pour accélérer le calcul du graphe SLAM basé sur des données laser. Plus récemment, [68] et [69] utilisent une architecture hétérogène intégrant un CPU et un FPGA pour accélérer des parties de l'algorithme SLAM. Tandis que [71] utilise des GPUs embarqués pour accélérer le prétraitement du RGBD SLAM.

Idris et al., [77] ont proposé une conception sur une architecture dédiée d'un multiplicateur matriciel pour le SLAM monoculaire. Toutefois, conformément à la Loi d'Amdal, un bloc qui représente un faible pourcentage du temps d'exécution global de l'algorithme, ne peut pas contribuer à une accélération significative même si ce bloc est 1000 fois plus rapide. En effet, le multiplicateur matriciel ne prend pas un temps significatif du temps total dans un système de SLAM monoculaire. Par conséquent, une réécriture algorithmique et des optimisations logicielles de l'algorithme sont nécessaires pour une implémentation efficace et temps réel.

1.8 Conclusion

Les algorithmes de localisation et de cartographie simultanées ont été largement étudiés pendant ces dernières années. Nous avons commencé par une présentation de la problématique générale de la localisation et de cartographie et nous avons défini l'architecture général d'un système de SLAM moderne.

Nous avons vu une variété de capteurs qui peuvent être utilisés dans les applications SLAM. Les capteurs proprioceptifs fournissent une estimation non-consistante de l'état future d'un système de SLAM. Les capteurs extéroceptifs, appelés aussi systèmes de perception, contribuent à l'amélioration de l'estimation de l'état future. Nous avons ensuite formulé les modèles mathématiques de la phase de prétraitement selon le type du capteur utilisé. Le modèle d'évolution ou bien cinématique interprète le mouvement du robot dans l'espace tandis que le modèle d'observation ou de perception décrit la manière dont le robot interagit avec son environnement. Finalement, nous avons cité les approches algorithmiques principales utilisées pour résoudre le cœur du SLAM. Les algorithmes basés sur des approches probabilistes semblent être plus utilisés pour la résolution du problème SLAM, tandis que ceux bio-inspirés sont peu étudiés bien qu'il existe des recherches en cours de réalisation afin d'améliorer leur consistance algorithmique et de faciliter leur mise en œuvre. Finalement, les algorithmes basés sur la structuration à partir du mouvement sont apparus récemment et commence à avoir lieu dans plusieurs applications robotiques.

La deuxième partie de cette étude bibliographique consiste à revoir les systèmes dédiés à l'exécution de ces algorithmes. En général les algorithmes SLAM sont coûteux et demandent des ressources importante lors de leur exécution. Les implantations embarquées de ces algorithmes ne sont pas nombreuses, en général on se limite à l'utilisation des machines performantes haute-de-gamme pour permettre leur exécution en temps réel. Des efforts ont été faits pour le calcul embarqué du SLAM, cependant, il paraît clairement qu'il n'existe aucune implémentation complète et efficace d'un algorithme SLAM sur une architecture embarqué.

Ce chapitre d'étude bibliographique permet de définir un axe de recherche important. Les implémentations embarquées qui ont été faites, ont fait ressortir l'intérêt principal d'une démarche scienti-

fique à mettre en place afin d'obtenir des performances satisfaisantes sur une architecture embarquée. Par conséquence, une démarche d'adéquation algorithme architecture est absolument nécessaire afin de choisir un algorithme adéquat, qui va tirer profit au maximum des capacités d'une architecture dédiée pour l'embarqué.

Auteurs	Capteurs	Coeur SLAM	Type de la carte	Architecture de calcul
Davison (2003) [46]	Caméra Monoculaire	MonoSLAM (EKF)	Métrique	Station de travail
Nistèr (2004) [50]	Stéreo/Caméra Monoculaire	Odométrie Visuelle	Métrique	Station de travail
Mouragnon (2006) [51]	Caméra Monoculaire+Odomètres	Odométrie Visuelle (OV) + BA	Métrique	Station de travail
Klein et al (2007) [78]	Caméra Monoculaire	GraphSLAM	Métrique	CPU embarqué : ARM A8
Milford (2008) [58]	Caméra Monoculaire + Odométres	RatSLAM	Topologique	Station de travail
Eade (2008) [44]	Caméra Monoculaire	FastSLAM2.0	Métrique	Station de travail
Gifford (2008) [79]	Télémètre infrarouge, gyromètres, odomètres	DP-SLAM	Métricque	CPU embarqué ARM
Bonato (2009)[80]	Caméra	EKF SLAM	Métrique	SoftCore + FPGA
Magnenat (2010) [81]	Télémètre laser rotatif	FastSLAM1.0	Métrique	CPU embarqué ARM
Botero (2012) [82]	Caméra	EKFSLAM	Métrique	Softcore MicroBlaze sur FPGA + CPU
Ratter et al 2013 [76]	Laser	Graph SLAM	Métrique	CPU + GPU haut de gamme
Tertei et al (2014) [68]	Caméra	Multiplicateur matriciel pour 3D EKF SLAM	-	HPS + FPGA
Nardi et al (2015) [71]	Caméra	Accélération de Kinect Fusion pour RGBD SLAM	-	GPUs embarqués haut de gamme CPU multicœurs
Sileshi et al. (2016) [69]	Laser + Odoméères	FastSLAM1.0	Métrique	Softcore Microblaze + FPGA
Ma et al. (2016) [70]	Caméra	Visuel-Inertiel SLAM	métrique	TITAN NVIDIA + Intel i7 Quad-CPU

Table 1.2: *Systèmes de SLAM et leurs architectures de calcul*

Chapitre 2

Méthodologie d'évaluation et de conception

Sommaire

2.1 Introduction

Les algorithmes du traitement du signal ou des images nécessitent une implémentation efficace afin de permettre une exécution en temps réel. Pendant ces dernières décennies, l'architecture des processeurs et des calculateurs ainsi que les outils du développement et de conception ont largement évolué. Ceci, a facilité l'implémentation des algorithmes et le développement des applications complexes. L'adéquation algorithme architecture (notée A3) est une approche globale formalisée qui consiste à étudier les deux aspects algorithmiques et architecturaux simultanément afin de réaliser de meilleurs implantations.

Après avoir défini un système (couple algorithme et architecture), une étude en simulation qui ne nécessite pas de ressources matérielles s'avère indispensable. Elle a l'avantage de valider le système en utilisant des scénarios extrêmes, souvent difficiles à tester en expérimentations. Ensuite, des évaluations expérimentales sont à mettre en place. Elles consistent à utiliser des séquences de données capteurs enregistrées par une plate-forme expérimentale dans un environnement réel. Elles permettent de tester la consistance et la robustesse d'un système SLAM dans des milieux réels internes/externes.

Dans ce chapitre, nous donnons en premier lieu une introduction de la méthodologie A3 suivie pour aboutir à une implémentation adéquate des algorithmes de SLAM. Les outils et les méthodes d'implémentation sont aussi décrits. La seconde partie est dédiée à la méthodologie d'évaluations de la consistance des résultats des algorithmes de localisation et de cartographie et la validation du SLAM sous un aspect système : partant du traitement des données capteurs jusqu'au traitement principal du cœur SLAM sur une architecture dédiée.

2.2 Adéquation Algorithme Architecture : Application au SLAM

Nous considérons ici des applications SLAM comprenant un calcul scientifique complexe et coûteux avec du traitement des images. Ces applications sont soumises à des contraintes du temps réel et d'embarquabilité et doivent avoir un comportement sûr. La complexité de ces applications, au niveau des algorithmes, de l'architecture matérielle et des interactions avec l'environnement sous la contrainte du temps réel, nécessite des méthodes pour minimiser la durée du cycle de développement, depuis la conception jusqu'à la mise en œuvre. Ceci est basé sur une méthodologie d'Adéquation Algorithme Architecture, appelée aussi "approche A3", qui formalise l'algorithme et l'architecture à l'aide de graphes. L'intérêt principal de ce modèle réside dans sa capacité à exprimer tout le parallélisme potentiel de l'algorithme, concrètement décrit sous la forme de schéma-blocs que l'on appellera plus tard des blocs fonctionnels. Il permet d'effectuer des optimisations précises prenant en compte des ressources matérielles programmables comme les processeurs et configurables tel que les circuits FPGA.

Un système se décompose en une partie matérielle, dont la structure est ci-après dénommée "*Architecture*", et une partie logicielle, dont la structure est ci-après dénommée "*Algorithme*". L'implantation consiste à mettre en œuvre l'algorithme sur l'architecture, c'est-à-dire à allouer les ressources matérielles de l'architecture aux opérations de l'algorithme, puis à compiler, charger et enfin lancer l'exécution du programme correspondant sur le calculateur. Enfin l'adéquation consiste à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée. Dans ce chapitre, on définit la méthodologie "Adéquation Algorithme Architecture" sur laquelle nous nous sommes basée pour aboutir à une implantation optimisée temps réel d'un algorithme SLAM sur une architecture homogène ou hétérogène. La méthodologie présentée est basée sur un modèle de

graphes factorisés qui utilise un ensemble d'opérations pour représenter un algorithme et un ensemble d'opérateurs pour représenter une architecture. Dans ce modèle, les optimisations sont exprimées en termes de transformations de factorisations.

2.2.1 Modèle et spécifications algorithmiques

La spécification algorithmique est le point de départ du processus de l'implantation matérielle d'une application sur une architecture. Un algorithme est le résultat de la transformation d'une spécification fonctionnelle, plus ou moins formelle, en une spécification logicielle adaptée à son traitement numérique par un calculateur. Autrement dit, un algorithme comme défini par Turing [83] est une séquence finie d'opération réalisables en un temps fini et avec un support matériel fini. La notion de l'algorithme peut aussi s'étendre pour prendre en compte d'une part l'aspect infiniment répétitif des applications réactive et d'autre part l'aspect parallèle nécessaire à leur implantation sur une architecture distribuée. Il existe deux approches principales pour spécifier un algorithme : celle orientée flot de contrôle et celle orientée flot de données. Dans les deux cas, on peut modéliser l'algorithme avec un graphe orienté acyclique [84] souvent appelé DAG (Directed Acyclic Graph). La formalisation mathématique du graphe est décrite dans l'annexe 7.2, section 7.2.3.

2.2.1.1 Graphe flot de contrôle

Dans un graphe flot de contrôle chaque sommet représente une opération qui consomme et produit des données dans des variables pendant son exécution, et chaque arc représente une précédence d'exécution [84]. L'ensemble des arcs définit un ordre d'exécution sur les opérations. Un arc est un contrôle de séquencement qui correspond soit à un branchement inconditionnel utilisé pour spécifier une itération finie ou infinie d'une opération, soit à un branchement conditionnel après évaluation d'une condition. La notion d'itération, inhérente à l'approche flot de contrôle, correspond à la répétition temporelle d'une opération ou d'un sous-graphe d'opérations. Il n'y a pas dans ce cas de relation entre l'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données. Si on veut faire apparaître du parallélisme potentiel dans l'approche flot de contrôle , il faut mettre en parallèle plusieurs graphe de flot de contrôle. Pour faire apparaître du parallélisme potentiel dans un graphe flot de contrôle qui n'en possède pas, on doit le transformer en un graphe flot de données par analyse des dépendances des données entre opérations à travers l'accès aux variables [85]. En revanche, cela s'avère complexe à cause du partage des variables.

2.2.1.2 Graphe flot de données

Dans un graphe flot de données [86], chaque sommet représente une opération qui consomme des données avant son exécution et produit des données après son exécution. Il introduit alors un ordre entre la lecture des données sur toutes les entrées et l'écriture des données résultats sur toutes les sorties. Au niveau de la spécification d'un algorithme la notion de variable n'existe pas. Chaque arc représente une précédence d'exécution induite par une dépendance de données entre une opération productrice et une ou plusieurs opérations consommatrices. Si deux opérations n'ont pas de données à s'échanger, elle ne sont pas connectées par un arc. L'ensemble des arc définit un ordre partiel d'exécution sur les opérations traduisant naturellement du parallélisme potentiel. Un arc est donc aussi un contrôle de séquencement. La donnée qu'elle véhicule est alors une donnée de contrôle utilisée pour gérer une itération ou du conditionnement. Il y a dans ce cas une relation entre l'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données.

En guise de conclusion, dans le cas du flot de contrôle les données sont indépendantes du contrôle, il revient au programmeur d'effectuer la spécification de l'algorithme et d'assurer l'ordre de consommation et de production des données. En revanche dans le cas du flot de données, celles-ci sont dépendantes du contrôle et de l'ordre d'accès aux données. Pour bien modéliser l'algorithme par un graphe flot de données, il est fondamental de maîtriser l'ordre dans lequel les données sont manipulées (cas des algorithmes de traitement d'image et notamment des applications SLAM). Dans la suite nous considérons ce type de graphe pour modéliser l'algorithme SLAM à implémenter sur une architecture embarquée.

2.2.2 Factorisation algorithmique

Certains algorithmes (dans notre cas le SLAM) impliquent des volumes de calculs suffisamment importants pour faire du traitement périodiques et identiques sur des données différentes, que l'esprit humain préfère spécifier sous forme de graphes factorisés. Ce type de spécification ne se limitent pas uniquement à réduire la taille de la spécification algorithmique, mais il permet aussi également de décrire plusieurs implantations plus ou moins séquentielles ou parallèles [87], chacune avec des caractéristiques différentes.

2.2.2.1 Sommets frontières de factorisation

La factorisation consiste à remplacer, dans un graphe un motif d'opérations répétitif par un seul exemplaire du motif, délimité par des sommets spéciaux appelés "frontières" [88]. La factorisation ne change pas les dépendances du graphe, elle permet simplement de réduire la taille de la spécification, tout en mettant en évidence ses parties répétitives. L'application SLAM qui nous intéresse, interagit avec l'environnement par une répétition infinie de la séquence acquisition-calcul-commande, qui correspond à un graphe de dépendances infiniment répétitif, dont la factorisation correspond à un graphe "flot de données". L'ensemble d'opérations d'un algorithme peut être spécifié sous différentes formes plus ou moins factorisées. La défactorisation, totale ou partielle, est une transformation trivial qui permet de produire automatiquement toutes les formes plus ou moins défactorisées d'une spécification. Chaque sommet frontière spécifie une forme différente de factorisation des arcs coupés par la frontière du motif. Les sommets frontières sont les suivants :[89]

- Sommet "*Diffuse*" : chaque répétition du sommet avant consomme la même valeur produite par le sommet amont, autrement dit, il diffuse la donnée en entrée aux motifs factorisés en sortie.
- Sommet "*Fork*" : chaque répétition du sommet avant consomme une partie différente du tableau produit par le sommet amont. Autrement dit il partitionne la donnée en entrée et distribue les parties aux motifs factorisés en sortie.
- sommet "*Join*" : chaque répétition du sommet amont produit une partie différente du tableau consommé par le sommet avant, autrement dit, il regroupe les données partitionnées en entrée par les motifs factorisés en un vecteur de sortie.
- sommet "*Iterate*" : marquant une dépendance de donnée inter-répétition (effet mémoire entre répétitions d'un motif), avec une seconde entrée initiale pour la première itération et une seconde sortie pour la dernière répétition. Autrement dit, il prend en entrée la sortie d'un des motifs factorisés ; sa sortie redirigée vers le motif factorisé suivant. Une valeur d'initialisation est nécessaire pour commencer le cycle d'itération.

Afin de mieux comprendre la notion de factorisation qui va être utilisé lors de la modélisation algorithmique de l'algorithme SLAM, nous présenterons en annexe 7.2, section 7.2.1 un exemple de factorisation d'algorithme en utilisant les sommets frontières de factorisation [90].

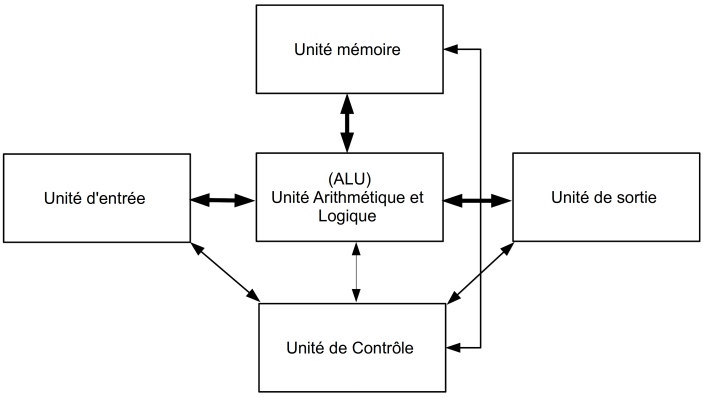


Figure 2.1: *Modèle classique d'une architecture de Von-Neumann*

2.2.3 Modèle Architecturale

2.2.3.1 Modèle classique d'architecture

Les modèles les plus souvent utilisés pour spécifier et modéliser des architectures parallèles ou distribuées sont les PRAM (Parallel Random Access Machines) et les DRAM (Distributed Random Access Machines) [91]. Le premier modèle correspond à un ensemble de processeurs communicant par mémoire partagée alors que le second correspond à un ensemble de processeurs à mémoire distribuée communicant par passage de messages. Ce modèle d'architecture de base est encore largement utilisé aujourd'hui. Ce modèle repose sur la coopération de cinq unités comme posé par Von Neumann (figure 2.1).

— L'unité mémoire contient des instructions et des données.
— L'unité de traitement effectue des calculs et transforme les données stockées en mémoire.
— Les unités d'entrée et de sortie permettent de transférer des données entre la mémoire et l'environnement.
— L'unité de commande contrôle l'ensemble de ces unités. Elle repose sur un séquenceur d'instructions qui lit les instructions dans la mémoire et les applique à l'unité de traitement. L'ensemble unité de commande - unité de traitement est souvent appelé processeur ou bien CPU *"Central Processing Unit"*.

Bien que toujours basés sur le principe de Von Neumann, les processeurs ont subit de nombreuses modifications architecturales dans le but d'améliorer leur puissance de traitement. IL est ainsi possible de les classer selon leur architecture interne [92].

2.2.3.2 Modèle d'architecture dans l'approche A3

Les modèles classiques sont suffisants pour décrire une architecture homogène. Cependant, ils ne permettent pas de prendre en compte des architectures hétérogènes comme cible pour des optimisations logicielles/matérielles. Pour cela, nous modéliseront l'architecture hétérogène distribuée par un graphe orienté (S, A), où S est l'ensemble des sommets de ce graphe et A l'ensemble de ses arcs. Chaque sommet est une machine à états finis (machine séquentielle) qui produit et consomme des données, chaque arc correspond à une connexion entre deux machines à états finis [93]. L'ensemble des sommets

S se décompose en cinq sous-ensembles disjoints qui correspondent chacun à un type de machine à états finis [94]. Les cinq types de sommets sont [95] :

- l'opérateur pour séquencer des opérations de calcul (séquenceur d'instructions).
- le communicateur pour séquencer des opérations de communications (ex : canal DMA)
- le bus/mux/démux avec ou sans arbitre pour sélectionner, diffuser et éventuellement arbitrer des données.
- la mémoire pour stocker des programmes

La mémoire peut aussi être considérée comme une machine séquentielle. Dans la méthodologie A3, il existe deux types de sommets mémoire :

- la mémoire RAM (Random Access Memory) pour stocker les données ou bien les programmes locaux à un opérateur
- la mémoire SAM, c'est une mémoire à accès séquentiel.

L'arbitre dans un bus/mux/démux constitue aussi une machine à états finis. Cet arbitre décide de l'accès aux ressources partagées que sont les mémoires. Les différents sommets ne peuvent pas être connectés entre eux de n'importe quelle manière, il est nécessaire de respecter un ensemble de règles. Par exemple, deux opérateurs ne peuvent pas être connectés directement, il en est de même pour deux communicateurs. Ces processeurs peuvent chacun être connecté à une RAM partagée ou à une SAM pour communiquer, en passant, ou non, par l'intermédiaire de communicateurs assurant le découplage entre calcul et communication. L'hétérogénéité d'une architecture signifie non seulement que les sommets peuvent avoir chacun des caractéristiques différentes (durée d'exécution des opérations, taille mémoire des données communiquées), mais aussi que certaines opérations peuvent n'être exécutées que par certains opérateurs, ce qui permet de décrire tout autant des composants programmable que des composants non programmables. Un processeurs dans la méthodologie A3 est décrit par un sous-graphe contenant un seul opérateur, une ou plusieurs RAM de données et de programmes locaux. Ce processeur n'ayant qu'un seul opérateur, n'est pas efficace pour la description d'une architecture à base de FPGA nécessitant généralement plusieurs machines à états finis ou opérateurs pour exprimer le parallélisme architectural de ce processeur [96]. Un exemple de modélisation architecturale est donné dans l'annexe 7.2, section 7.2.2.

2.2.4 Modèle d'implémentation

A partir d'un graphe d'algorithme et d'un graphe d'architecture, il est possible de construire un graphe correspondant au graphe d'implantation au moyens de la composition de trois relations : le routage, la distribution et l'ordonnancement. L'implémentation d'un algorithme sur une architecture consiste à réaliser, en tenant compte des contraintes, une distribution et un ordonnancement des opérations de l'algorithme sur une architecture donnée. Autrement dit, la distribution est souvent appelée placement ou répartition.

La distribution consiste tout d'abord à effectuer une partition du graphe de l'algorithme initial (que l'on appellera plus tard partitionnement en blocs fonctionnels des algorithmes de SLAM) en autant ou moins d'éléments de partition qu'il y a d'opérateurs dans le graphe de l'architecture. Il faut ensuite affecter chaque élément de partition, c'est-à-dire chaque sous-graphe correspondant du graphe de l'algorithme initial, à un opérateur du graphe de l'architecture. Puis il faut affecter chaque transfert de données du graphe de l'algorithme reliant des opérations appartenant à des éléments de partition différents, à une route. Chaque route est un chemin dans le graphe de l'architecture qui connecte un couple d'opérateurs. Ce chemin est formé d'une liste de moyens de communication comportant chacun un couple de communicateurs. Les communicateurs assurent l'acheminement des données routées dans

un processeur. Une route peut se réduire à un seul moyen de communication si on a une communication directe entre deux opérateurs. (Annexe 7.2, section 7.2.4).

Les opérateurs de calculs et les opérateurs de communication sont des machines séquentielles à états finis. Ils ont à exécuter chacun un sous-graphe qui doit être un ordre total. L'ordonnancement consiste à rendre total, par ajout d'arcs, l'ordre partiel associé à chaque sous-graphe de l'algorithme formé d'opération, d'opérations de communication, de sommets identité et d'allocations, affectés à un sommet respectivement opérateur, communicateur, bus/mux/démux, et mémoire du graphe de l'architecture. (Annexe 7.2, section 7.2.5.

2.2.5 Transformation de graphe et adéquation

Le problème d'optimisation s'agit tout d'abord de respecter les contraintes temps réel qui sont de deux types : latence et cadence. La première contrainte concerne la durée d'exécution d'une réaction (la durée du chemin critique du graphe de l'algorithme). La seconde contrainte concerne la durée qui s'écoule entre deux réactions. Dans l'approche A3, le graphe de l'algorithme spécifié par l'utilisateur est normalement transformé par des outils logiciels de CAO avant d'effectuer l'adéquation. Cette dernière consiste à trouver la meilleure distribution et le meilleur ordonnancement d'un graphe flot de donnée sur une architecture cible. Dans le cas général, la meilleur implantation (distribution et ordonnancement) demeure une solution approchée que l'on recherche et non un solution optimale que l'on ne peut obtenir dans un temps raisonnable que pour des cas simples. Pour cela, des heuristiques sont utilisées pour essayer de nombreuses variantes d'implantation et de réaliser automatiquement la défactorisation pour choisir enfin l'implantation la plus adéquate et qui respecte les contraintes de temps réel. Dés qu'une implantation (distribution et ordonnancement) a été choisie et déterminée, des exécutifs sont générées automatiquement par ces outils logiciels [93].

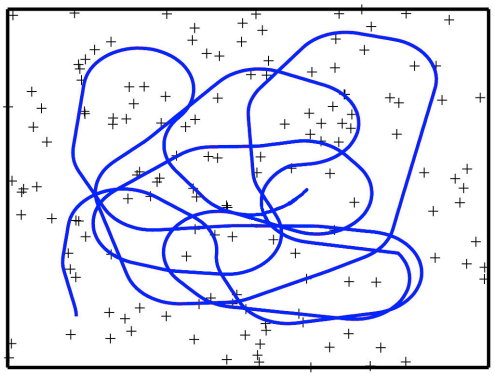
Bien que dans cette thèse l'étude porte sur des algorithmes de SLAM qui sont par nature des algorithmes hyper-complexes, nous étudierons lors de leur optimisation, non seulement l'aspect algorithmique et architectural mais aussi l'aspect logiciel. Nous étudierons plusieurs implantations avec différents langages de programmation pour tenir en compte leur impact sur l'implémentation résultante. Pour cela, la génération automatique des exécutifs en utilisant des heuristiques ne nous concerne pas. Les transformations de graphe de l'algorithme en fonction du graphe d'architecture sont faites à la main pour générer le graphe d'implantation adéquat. Ainsi, l'implantation matérielle des algorithmiques consiste à développer des programmes (C++, OpenGL, OpenCL) correspondant à ces graphes.

2.3 Outils de Prototypage et méthodologie d'évaluation

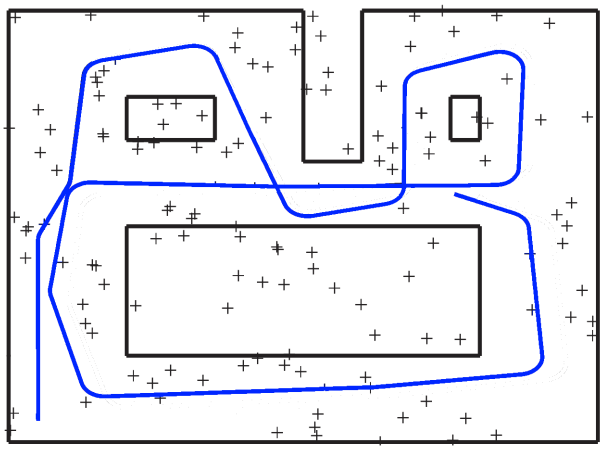
2.3.1 Outils de simulation

Dans cette thèse nous utiliserons un simulateur d'environnement [97], modifié et adapté à notre application, afin d'évaluer l'algorithme SLAM en simulation. Ce simulateur produit plusieurs environnements de tailles différentes qui contiennent un ensemble de couloirs et des points d'intérêts. La réalité terrain est représentée par un ensemble de points de passage reliés ente eux. Au fur et à mesure du trajet, le simulateur fournit à chaque étape les données de l'encodeur de la roue droite et gauche en fonction du déplacement du mobile. Il fournit aussi l'angle et la profondeur des points d'intérêts de la carte avec leurs identificateurs. Il est capable aussi de fournir la position de l'amer dans l'image. Le simulateur, développé sous Matlab, simule un environnement de taille variable, constitué d'un ensemble de mur et d'amers permettant de modéliser des environnements structurés tel que des couloirs ou des

étages complets de bâtiments. Les amers sont disposés aléatoirement dans l'environnement. Le trajet suivi par le mobile est composé de points de passage, reliés par un ensemble de courbes et de droites. Pour les données proprioceptives, durant le trajet, le simulateur renvoie les données odomètriques issues du parcours du mobile. Ces données peuvent être altérées en utilisant différents bruits (gaussien, uniforme, biaisé). Pour les données extéroceptives, le simulateur modélise une caméra sur le robot, il calcule la projection de chaque amer sur l'image de la caméra. Puis, il envoie le numéro de l'amer détecté ainsi que sa position sur l'image. Le simulateur modélise aussi un capteur Laser, il calcule l'angle et la profondeur de l'amer dans la scène. L'association entre un amer nouvellement détecté et un amer précédemment détecté est supposée connue grâce au numéro des amers. Le simulateur ne gère pas les problèmes d'appariement. Les algorithmes de SLAM ont des comportements différents en fonction de plusieurs paramètres de l'environnement. Les performances d'un algorithme SLAM dépendent de plusieurs paramètres liés à l'environnement tel que la taille et le nombre des amers visibles. Le résultat de la localisation dépend du nombre d'amers présents dans l'environnement mais aussi de la taille de ce dernier ou encore de la visibilité des amers. On définit deux cartes de test illustrées dans la figure 2.2. Ils représentent des situations différentes : l'exploration d'une petite salle et l'exploration d'une grande salle avec un ensemble de couloirs. Les deux situations présentent des difficultés diverses pour un algorithme de SLAM. En particulier, elles mettent en évidence le problème de la taille de l'environnement ou celui de la visibilité des amers.[97],



(a) *Environnement 1*



(b) *Environnement 2*

Figure 2.2: *Définition des environnements et trajets de simulation*

2.3.2 Données Expérimentales

Après l'évaluation des algorithmes en simulation, il faut les évaluer et les valider expérimentalement. L'évaluation expérimentale peut se faire de deux manières. Elle consiste soit d'utiliser un jeu de données réelles, soit d'instrumenter une plateforme expérimentale.

2.3.2.1 Jeu de données réelles

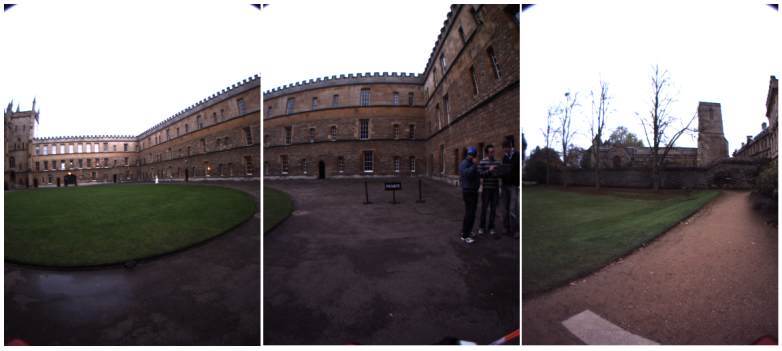
L'étude, la conception et la commercialisation des robots autonomes repose principalement sur le fait que les développeurs et le groupe des chercheurs disposent, ou se procurent facilement, des outils pour tester et valider les algorithmes de localisation et de cartographie. On peut démarquer ces outils selon les catégories suivantes :

— Jeu de données capteurs pour tester l'algorithme SLAM dans un environnement réel, interne ou externe.
— Des méthodologies pour l'évaluation quantitative et la comparaison avec d'autre algorithmes
— Un ensemble des algorithmes qui ont été testés et validés, constituant alors une ébauche pour développer d'autre solution et pour la comparaison des performances.

Howard et Roy [98], un groupe universitaire, a fait les investissements nécessaires pour créer un jeu de donnée contenant quelques outils permettant de valider un système de SLAM. Cependant, les outils fournis restent très limités en termes de performances et de versatilité. Dans des domaines pas si éloignés des sujets d'intérêt pour les roboticiens, comme la vision par ordinateur par exemple, il est un peu plus fréquent de se référer à des outils et données (Dataset), habituellement disponibles sur le web, en tant que source de données pour tester expérimentalement des propositions scientifiques innovantes. Comme exemple, on trouve Group [99] pour l'évaluation des performances pour un système de trafic, et Scharstein et Szeliski [100] pour la reconstruction 3D. Les chercheurs, et éventuellement les entreprises, qui s'intéressent au domaine de la robotique, trouvent un grand dissuasif qui se reflète dans la difficulté et le coût de se procurer des outils permettant de valider les performance d'un algorithme. Notamment, le moyen pratique pour avoir de tels outils, est de mettre en place un projet de recherche lourdement coûteux pour construire une plateforme permettant l'acquisition des données et d'outils d'expérimentations. Sans oublier bien sur le temps de développement de tel projet. Si un jeu de données réelles (permettant de faire la conception, les tests de validation et d'évaluation des performances des algorithmes de SLAM) seraient disponibles au grand public, ils auraient facilité et accéléré à son tour les activités de recherches dans le domaine de la robotique. Il aurait aussi encouragé les chercheurs à rejoindre le progrès des nouvelles technologies de recherche dans le domaine de la robotique. Dans cette thèse, nous utiliserons différents jeux de données pour valider l'aspect fonctionnel des algorithmes de SLAM étudiés. Nous utiliserons Rawseeds Andrea Bonarini et Tardos [101] Simone Ceriani [102], NewCollege [103] et KITTI [104]. Ces jeux de données réelles participent au processus du développement des algorithmes de localisation et de cartographie en offrant un ensemble des outils et des données capteurs disponibles gratuitement aux communautés de robotique. La figure 2.3 montre des séquences d'images de l'environnement où les données capteurs ont été collectées.

(a) *Jeu de données Rawseeds*

(b) *Jeu de données Kitti*



(c) *Jeu de données Oxford*

Figure 2.3: *Bicocca : Environnement interne où les séquences ont été enregistrées*

Les données capteurs (y compris les données visuelles) incluses dans les jeux de données sont des données brutes. Ils ne sont pas sujettes au pré-traitement ou compression à priori. Ces données sont issues de l'exploration de quelques environnements (interne et/ou externe) par un robot équipé de plusieurs capteurs de prédiction et de perception de haute/moyenne qualité pour répondre aux besoins des utilisateurs. Ces équipements sont choisis pour répondre aux critères de performances et de qualité des robots autonomes et pour fournir à la communauté des jeux de données brutes utiles pour une extraction des informations de haut niveau de l'environnement exploré. Le robot utilisé pour

la récolte des données est équipé d'un système de capteurs de :

— perception (extéroceptive) : capteur de profondeur laser allant jusqu'à 180 m, des caméras stéréoscopiques, caméra omnidirectionnelle, caméra couleur, capteurs sonores, GPS.

— prédiction (proprioceptives) : un système d'odométrie, gyroscopes et accéléromètres.

Durant l'exploration, l'acquisition des données est faite en utilisant un PC interfacé à plusieurs cartes d'acquisition. Parfois, la récolte des données capteurs est souvent sujette aux bruits. Pour résoudre ce problème des méthodologies extensives de validation des données capteurs sont mises en places pour vérifier et certifier la consistance des données fournies par chaque capteur. Des analyses statistiques sont aussi offertes afin de s'assurer de la cohérence des données produites avec la réalité terrain. Dans nos tests, nous avons utilisé deux types de capteurs (proprioceptives et extéroceptives) dont les modèles mathématiques ont été développés dans le premier chapitre. On a utilisé les capteurs odomètriques pour l'étape de prédiction des algorithmes SLAM probabilistes. Pour la perception d'environnement, nous avons utilisé un capteur de profondeur laser et une caméra monoculaire .

2.3.2.2 Réalité terrain

La réalité terrain de la position et l'orientation du robot est nécessaire pour l'évaluation des résultats de localisation et de cartographie des algorithmes SLAM. Le GPS différentiel D-GPS est un outil intéressant utilisé pour générer la réalité terrain, il est capable de générer la position et l'orientation du robot avec haute précision. Notons que dans des environnements internes, un tel dispositif n'est pas possible. Des technologies différentes sont donc utilisées pour générer la réalité terrain tel que la technologie de localisation sans fil ou bien l'utilisation des caméras intelligentes pourvues des algorithmes de traitement d'images et de vision par ordinateur. Ceci permet la génération de la réalité terrain dans des zones réduites comme les environnements internes. Cette réalité terrain sera considérée comme référence pour évaluer la consistances et les performances des algorithmes SLAM.

2.3.3 Validation Hardware In the Loop

Le SLAM ne se limite pas à un algorithme seulement mais nécessite un système incluant des capteurs, des calculateurs et une plate-forme. L'évaluation d'un système de SLAM ne s'arrête pas à la validation de l'algorithme en tant que tel, mais s'étend au système dans son ensemble d'où l'intérêt de la validation par la méthode HIL (hardware in the loop) [105] [106]. La validation HIL est couramment utilisée pour valider un système matériel dans une chaîne de traitement logiciel. Dans notre cas, cette méthode nous permet d'évaluer les résultats de localisation d'un algorithme SLAM en utilisant un simulateur comme source de données et une architecture matérielle, dédiée pour les applications embarquées, pour le traitement. La figure 2.4 illustre l'intégralité des possibilités du système d'évaluation "Hardware In the Loop". Le schéma différencie la phase d'acquisition des données capteurs de la phase de traitement. Deux sources sont disponibles pour fournir des données capteurs :

— Le simulateur : il peut être utilisé comme source de données capteurs. Le simulateur fournit des données proprioceptives (odomètriques) et des données extéroceptives (la position des amers sur l'image dans le cas d'une caméra, la profondeur et l'angle de l'amer dans le cas d'un capteur Laser). Les données du simulateur sont transmises par liaison série. La liaison RS232 a été choisie car le volume de données à transmettre est beaucoup moins important que pour la transmission d'une image complète.

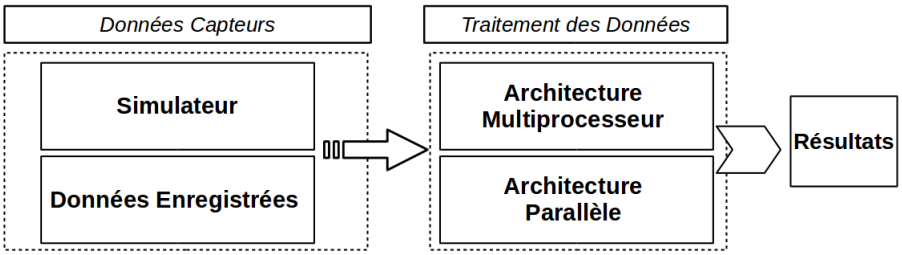


Figure 2.4: *HIL*

— Le rejeux de données : après avoir effectué une expérimentation, il est possible de rejouer les données capteurs. Ce mode de fonctionnement est principalement utilisé pour mettre au point et évaluer les algorithmes SLAM. Un prétraitement des données peut être nécessaire.

Deux cibles sont disponibles pour traiter les données capteurs :

— Des architectures multiprocesseurs : des cartes pour l'embarqué peuvent être interfacées avec le système de validation HIL . Grâce à ce système, on peut évaluer les différents calculateurs embarqués.

— Des architectures massivement parallèle (GPU ou FPGA) : dans le chapitre 5, nous proposons une implémentation sur des architectures hétérogènes adaptées aux applications SLAM. Cette méthode de validation permet de valider l'intégralité du système.

Le système d'évaluation mis en place permet de tester à la fois les résultats des algorithmes en utilisant des données simulées mais aussi des données réelles. Il évalue des systèmes en interfaçant directement des architectures matérielles au simulateur .

Afin d'évaluer l'implémentation des algorithmes SLAM sur les architectures embarquées, nous avons développé une plat-forme de validation HIL. La figure " Fig.2.5 " détaille la plate-forme développée pour l'évaluation. L'émulateur logiciel est utilisé comme source de données capteurs simulées ou réelles. Dans ce dernier cas, le jeux de données enregistré est étiqueté temporellement. Il est ensuite rejoué par l'outil logiciel émulateur (développé sous Matlab). Ce dernier envoie les données correspondantes à la profondeur, l'angle des objets dans la scène et les valeurs des encodeurs optiques par une liaison série vers le calculateur. L'algorithme SLAM optimisé est implémenté sur une l'architecture embarquée cible. Les résultats de localisation et de cartographie sont renvoyés vers l'émulateur par la même liaison série. L'outil permet de tracer la trajectoire estimée, la trajectoire de référence et la carte reconstruite à chaque instant d'échantillonnage.

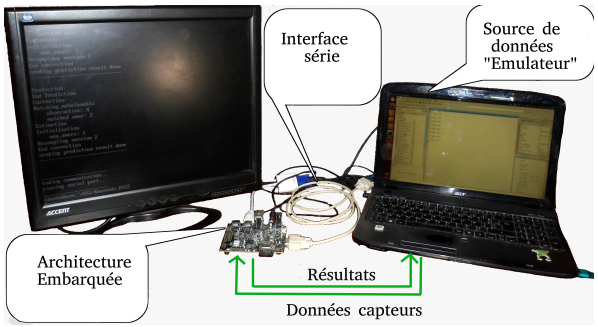


Figure 2.5: *Plate-forme de validation HIL*

2.3.4 Critères d'évaluation

Pour évaluer un algorithme ou un système, des critères sont définis pour quantifier ses performances. Les algorithmes de SLAM localisent un mobile et cartographient un environnement. Il faut évaluer la localisation du robot mais aussi la reconstruction de l'environnement. Plusieurs critères sont couramment utilisés comme l'erreur absolue sur la trajectoire, la distance euclidienne ou bien le NEES.

2.3.4.1 Erreur absolue sur la trajectoire (ATE)

L'erreur absolue sur la trajectoire notée ATE (*Absolut Trajectory Error*) est une mesure obligatoire. Son calcul est nécessaire pour juger la précision des résultats de localisation et pour s'assurer qu'un algorithme peut être considéré comme consistant. ATE est souvent utilisé à chaque fois qu'un algorithme SLAM est appliqué pour estimer la position du robot. Le but de l'ATE est l'évaluation des erreurs sur la reconstruction de trajectoire introduites par l'algorithme SLAM . A chaque instant, et pour chaque position de référence spécifiée par la réalité terrain, une comparaison est effectuée entre cette position et la position du robot estimée par l'algorithme de SLAM. L'ATE est donc une description cumulée des erreurs de localisation. La trajectoire reconstruite par un algorithme SLAM est souvent calculée par rapport à une référence locale, autrement dit un repère associé au robot. Cependant, le calcul de l'ATE exige que la trajectoire reconstruite par le SLAM soit dans le même système de cordonnées que la référence qui est à son tour générée par rapport à un repère fixe dans l'environnement.

L'erreur absolue sur la trajectoire est calculée à travers les étapes suivantes :

— Pour chaque instant de temps par rapport à la réalité terrain : Évaluer la position du robot à l'instant t estimée par le SLAM. Des interpolations pourrait être nécessaires si le temps d'acquisition de la position estimée ne correspond pas à celui de l'acquisition de la référence. On calcule après l'erreur de translation d_i entre la position de référence notée x_{GT}^t et la position estimée notée x_t, $d_j = ||trans\,(x_j) - trans\left(x_j^{GT}\right)||$.

— On calcule les valeurs numériques suivantes : la moyenne de l'erreur de translation $\overline{d}_j$, la déviation standard de l'erreur de translation σ_{d_j}, les deux extrêmes $a_{d_j,,3\sigma}$ et $b_{d_j,3\sigma}$ de l'intervalle de confiance de l'erreur de translation d_j.

— L'erreur absolue sur la trajectoire est finalement donnée par l'équation : $\mathbf{ATE} = \left[\overline{d}_j, \sigma_{d_j}, a_{d,3\sigma}, b_{d_j,3\sigma}\right]^T$

2.3.4.2 Distance euclidienne

Pour évaluer la qualité de la localisation du mobile, la distance euclidienne entre la position réelle du mobile et sa position estimée peut être calculée. Cette distance représente l'erreur introduite par l'algorithme à chaque instant de la trajectoire. La distance euclidienne est calculée donc entre la position de référence $\mathbf{x}_k = (x_{k,ref}, y_{k,ref})$ du robot et sa position estimée $s_t = (x_t, y_t, \theta_t)$ à chaque instant t.
La distance est calculée à l'aide de la formule suivante :

$$d = \sqrt{(x_{t,GT} - x_t)^2 + (y_{t,GT} - y_t)^2} \tag{2.1}$$

Cette distance représente l'erreur produite par l'algorithme relative à la localisation du mobile.

2.3.4.3 NEES

Le couloir d'incertitude renseigne sur la consistance de la localisation. Cependant, le couloir ne prend pas en compte la forme de l'ellipse d'incertitude : son orientation n'est pas réellement utilisée. Pour combler ce manque, on utilise le NEES défini pour N expérimentations par :

$$\epsilon_{rob} = \frac{1}{N}\sum_{j=1}^{N}(\hat{\mathbf{x}}_k - \mathbf{x}_k)^T \hat{\sigma}_{\hat{\mathbf{x}}_k}(\hat{\mathbf{x}}_k - \mathbf{x}_k) \tag{2.2}$$

Pour 3 dimensions et N = 50, la zone de 95% de probabilité est définie par l'intervalle $[2.36, 3.72]$. Si ϵ est supérieur à la borne supérieure de l'intervalle alors le filtre est optimiste : la localisation du robot n'inclut pas la position réelle. Si ϵ est inférieur à la borne inférieure alors le filtre est conservateur : la zone de localisation du robot est plus grande que la normale.

2.3.5 Évaluation des temps d'exécution

Après avoir évalué les résultats de localisation et de cartographie d'un algorithme SLAM, il faut évaluer son temps d'exécution. Une première approche consiste à calculer le temps d'exécution de l'intégralité de l'algorithme. Ces temps sont calculés en milliseconde à l'aide d'un temporisateur physique (Timer) intégré dans les architectures de test. Une deuxième approche consiste à calculer le nombre de cycles comme une métrique commune pour évaluer l'implémentation sur des architectures de différentes natures. Généralement, le temps d'exécution d'un algorithme de SLAM n'est pas constant et dépend de plusieurs paramètres. Notre méthodologie d'évaluation des temps d'exécution consiste à analyser l'algorithme et ses dépendances. Ensuite, on identifie les tâches de traitement qui ont besoin d'un temps de calcul considérable. En découpe ensuite l'algorithme sous forme de blocs fonctionnels ayant des temps de traitement bornés.

Les temps d'exécution des blocs fonctionnels (BFs), dépendent de plusieurs paramètres de l'algorithme et de l'environnement. Cette dépendance détermine le nombre d'occurrences d'un bloc fonctionnel par itération de l'algorithme. Afin de tenir en compte cette dépendance durant notre évaluation, les temps d'exécution sont calculés par rapport à la moyenne d'occurrences par itération (MOI). Si un bloc fonctionnel BF_x est exécuté $\mathrm{N}_{\mathrm{BF}_x}$fois dans une itération, MOI est donnée par l'équation (2.3). Avec n est le nombre d'itération de l'algorithme.

$$\mathrm{MOI} = \frac{1}{n}\left(\sum_{i=1}^{n} \mathrm{N}^{\mathrm{i}}_{\mathrm{BF_x}}\right) \tag{2.3}$$

Par conséquence, le temps d'exécution moyen t'_{BF_x} du bloc fonctionnel BF_x est donné par l'équation (2.4). Avec t_{BF_x} est le temps d'exécution (moyenne de 10 exécutions de l'algorithme pour 500 itérations) d'un bloc fonctionnel BF_x.

$$t'_{\mathrm{BF}_x} = \mathrm{MOI} * t_{\mathrm{BF}_x} \tag{2.4}$$

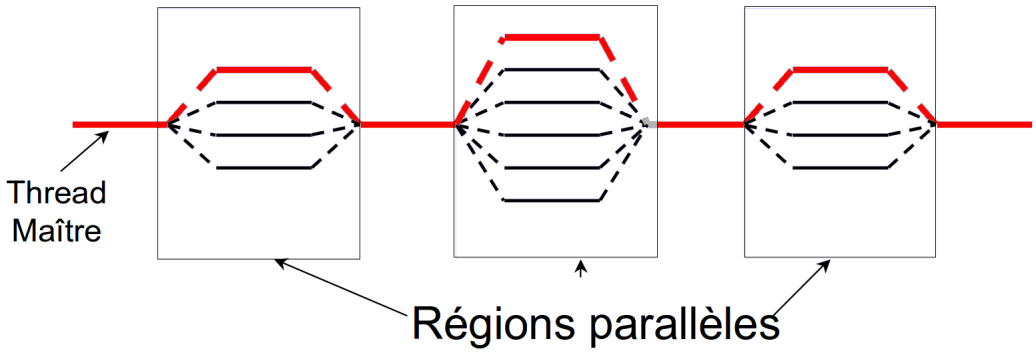


FIGURE 2.6: *OpenMP : Modèle d'exécution*

2.4 Outils de développement et de conception

Dans cette thèse on a utilisé une variété d'architectures embarquées grand publique pour évaluer les algorithmes étudiés. Une maîtrise tous d'abord des outils de développement et de conception sur ces architectures embarquées s'avère essentiel. Dans cette section nous présenterons les différents outils utilisés dans cette thèse pour le développement sur des architectures mono-cœurs CPU, multi-cœurs CPU, des architectures hétérogènes, des architectures massivement parallèles telles que les GPUs et des architecture programmables FPGA.

2.4.1 Optimisation logicielle sur architectures multicœurs

OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est supportée par de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Elle se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement. OpenMP est portable et permet de développer rapidement des applications parallèles à petite granularité . La parallélisation sur une architecture homogène est facilitée par l'utilisation de librairies OpenMP qui permettent de répartir facilement le traitement réalisé sur les différents processeurs. De plus, ce type de librairies peut gérer les différents accès concurrents à la mémoire partagée. OpenMP permet principalement de paralléliser les boucles les plus coûteuses en temps et distribuer leurs itérations sur plusieurs tâches. La figure 2.6 montre le principe de parallélisation avec OpenMP. Le thread Maître lance un ensemble de threads selon les besoins (région parallèle). Le parallélisme est introduit de façon incrémentale et le programme séquentiel évolue vers un programme parallèle.

Un programme OpenMP peut être compilé par un compilateur non OpenMP. Ces compilateurs offrent généralement la possibilité de compiler un programme sans interpréter les directives OpenMP ce qui permet la comparaison rapide entre parallélisation automatique et parallélisation par directives.

2.4.2 Calcul sur GPU

Les GPUs ont tellement évolués que de nombreuses applications industrielles utilisent aujourd'hui leur puissance de calcul pour le calcul.

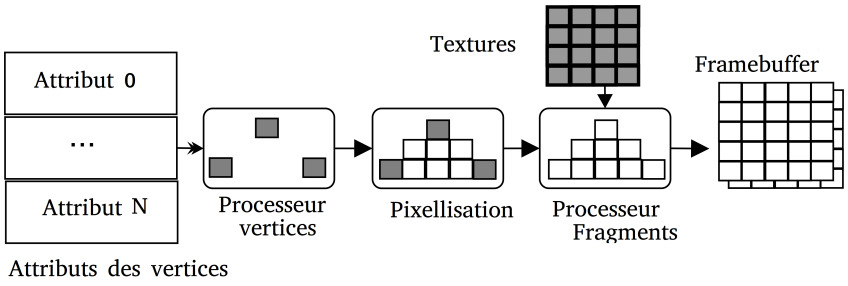


Figure 2.7: *Représentations de l'architecture du GPU avant l'unification des shaders*

2.4.2.1 Programmation CUDA

Comme son nom l'indique, CUDA (Compute Unified Device Architecture) représente une révolution du modèle architectural des cartes graphiques Nvidia. Les cartes graphiques NV20 (2001, GeForce 3 family) et NV40 (2004, GeForce 6800 family) étaient constituées d'un pipeline graphique composé de trois principales étapes : le vertex shader, le rasterizer et le fragment shader (Figure 2.7). Depuis la configuration G80 (2006, GeForce 8800 family), dans la perspective de transformer le pipeline graphique en une machine parallèle, Nvidia a unifié l'architecture des processeurs du vertex shader et du fragment shader. Son successeur, apporte surtout des améliorations au niveau de la puissance délivrée et des ressources allouables. Peu de modifications bousculent l'architecture de la G80 mis à part le nombre de Streaming Multiprocessor (SM) par Thread Processing Cluster (TPC) qui passe de 2 à 3. Avec la GT300 (ou GF100 ou Fermi), Nvidia fournit davantage de fonctionnalités : une augmentation du nombre de processeurs de calcul flottant à double précision, un développement de la hiérarchie des caches entre la mémoire globale et les processeurs scalaires, une protection ECC contre les erreurs éventuelles pendant les transactions de données, une unification de l'espace d'adressage des espaces mémoires de la carte et du GPP permettant d'exécuter du code C++ directement sur le GPU et la possibilité d'exécuter 16 kernels simultanément (limité a un auparavant). Avec l'architecture Fermi, les GPU Nvidia sont devenus des architectures parallèles dédiées au calcul scientifique à part entière. Depuis, une dernière génération de cartes a fait irruption, la Kepler, améliorant la souplesse d'utilisation des GPU dans le calcul parallèle sur les GPU. [107]

2.4.2.2 Programmation OpenCL

OpenCL (*Open Computing Language*) est un standard récent, ratifié par le groupe Khronos, pour programmer des ordinateurs hétérogène. . OpenCL est constitué d'un langage de programmation adapté aux accélérateurs et d'une API pour orchestrer l'appel aux kernels OpenCL sur les accélérateurs à partir de la plateforme. Le langage de programmation est basé sur le C99, avec une philosophie et une syntaxe similaire à CUDA exploitant le modèle Single Program Multiple Data (SPMD) pour la programmation parallèle. Le parallélisme de tâches est également supporté par le lancement de tâches multiples sous forme de kernels mono-thread, exprimant le parallélisme sur des vecteurs de données. Le choix d'exécuter les kernels en pile de commandes permet une exécution des tâches dans le désordre tout en exprimant le parallélisme de tâches à granularité fine. Les tâches exécutées dans le désordre sont synchronisées par barrières ou explicitement en spécifiant les dépendances. La synchronisation entre piles de commandes réparties sur différents accélérateurs est aussi explicite. Le standard définit une charte d'utilisation pour une utilisation conforme d'OpenCL, comme il a été le cas pour OpenGL. Des extensions optionnelles sont également définies, incluant les fonctions double précision et les fonctions atomiques. Apple a déjà incorporé OpenCL dans Mac OS X Snow Leopard et les deux constructeurs

de cartes graphiques Nvidia et AMD ont mis sur le marché des compilateurs beta [108].

2.4.2.3 Programmation OpenGL

OpenGL (*Open Graphics Library*) a été crée premièrement pour la création graphique. Pourtant, il peut être utilisé pour le calcul générique de haute-performance. Pour cela nous adoptons les principes suivants :

Les textures sont équivalent aux tableaux Par défaut, pour le CPU, les données sont stockées dans des tableaux d'une seule dimension, tandis que pour le GPU, les données sont stockées dans des tableaux à deux dimensions. Deux ou trois dimensions sont aussi supportées par le GPU, mais ils ne peuvent pas être utilisées directement avec les techniques que nous adoptions dans notre implémentation. Les tableaux dans la mémoire du GPU sont appelées des textures. Les valeur stockées dans les textures sont appelées des texels (texture d'un pixel), ils sont accessibles en utilisant les cordonnées textures. Chaque élément dans la texture peut stocker jusqu'à 4 valeurs de type RGBA (Rouge, Bleu, Vert, Alpha) qui permettent au GPU de traiter 4 données simultanément. Il existe plusieurs type de texture supportées par la majorité des GPU. Dans notre implémentation, nous choisissons d'utiliser une texture de type rectangle qui est la plus adaptée au type de calcul que nous utiliseront. Nous choisissons également d'utiliser RGBA comme format de texture pour stocker quatre valeur flottante par texel afin d'accélérer d'avantage le calcul. Afin de contrôler l'accès exacte aux valeurs stockées dans les texels, nous utiliserons une projection orthogonale et un angle de vue appropriée. Pour assurer une exécution performante, nous utiliseront les textures pour stocker les données d'entrée/sorties.

Les Shader sont équivalent aux kernels La différence majeure entre le CPU et le GPU réside dans le fonctionnement du kernel ou du shader. Une bonne compréhension du principe de calcul parallèle facilite la programmation des shaders. Pour exécuter des calculs parallèles, nous utiliserons le shader des fragments dans le GPU. Il se compose de plusieurs éléments de traitement qui se comportent comme un processeur vectoriel type SIMD. Les unités des fragments traitent les éléments stockés dans les textures en parallèle. Pour cela, le shader des fragments est exécuté en parallèle pour chaque élément de texture.

Le dessin est équivalent au calcul Le calcul parallèle souhaité est déclenché en dessinant une primitive géométrique adéquate. Pour s'assurer que le shader des fragments est exécuté pour chaque élément de donnée dans la texture, chaque élément doit être transformé en fragment. Pour cela, nous dessinons un rectangle rempli qui couvre tous les éléments de données qu'on veut traiter en parallèle. Les cordonnées des texture des quatre sommets du rectangle sont définis comme des attributs, la phase de pixellisation utilise une interpolation bilinéaire pour générer un fragment pour chaque pixel de la texture d'entrée. Cette phase se comporte comme un générateur de flux de données pour le shader de fragment.

2.4.3 Outils de synthèse et de conception sur FPGA

2.4.3.1 Conception avec un langage de description matérielle

VHDL est un langage de description de matérielle destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. En VHDL, tout composant (dans le sens logiciel) est décrit sous deux aspects : interface avec le monde extérieur, décrite dans une section dénommée entity et l'implémentation elle-même, décrite dans une section dénommée architecture.

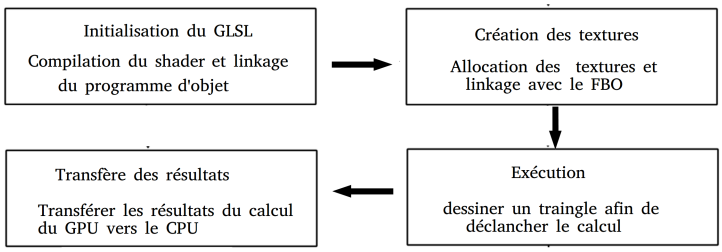


Figure 2.8: *Représentation de l'architecture du GPU avant l'unification des shaders*

La section architecture contient la description de la fonction matérielle désirée soit sous forme de description structurelle précise de l'architecture matérielle ; soit sous forme de comportement attendu ou bien orienté flot de données. Le VHDL reste un langage difficile à mettre en œuvre surtout pour les algorithme présentant une complexité et des dépendances de données.

2.4.3.2 Conception de haut-niveau HLS (HLS : High-Level Synthesis)

La synthèse de haut niveau HLS (High-Level Synthesis) est un processus de conception automatisé qui interprète une description algorithmique d'un comportement souhaité et crée une architecture matérielle numérique implémentant ce comportement. Bien que la synthèse logique utilise une description RTL de la conception, la synthèse de haut niveau fonctionne à un niveau d'abstraction plus élevé, en commençant par une description algorithmique dans un langage de haut niveau tel que SystemC, OpenCL, C/C++. Le concepteur développe généralement les fonctionnalités du module et les protocoles d'interconnexion. Les outils de synthèse de haut niveau transforment un code fonctionnel en une implémentation RTL entièrement chronométrée, créant automatiquement le détail cycle par cycle pour la mise en œuvre du matériel. Les implémentations (RTL) sont ensuite utilisées directement dans un flux de synthèse logique classique pour créer une implémentation au niveau matériel.

2.5 Bilan

La définition d'un système SLAM nécessite une démarche scientifique à mettre en place. Son développement ainsi que sa validation requirent des outils et méthodologies pour évaluer le bon fonctionnement de l'ensemble.

Durant ce chapitre, nous avons défini l'approche d'adéquation algorithme architecture. Elle consiste tout d'abord à définir un graphe orienté flot de donné correspondant à l'algorithme à implémenter. Ce graphe sert à donner une représentation compact ou non de l'algorithme selon le type de factorisation utilisée afin de faire apparaître les dépendances de données ou bien un parallélisme potentiel. Ensuite, on définit le graphe correspondant à l'architecture cible. Ce graphe fait apparaître les opérateurs de calcul et du traitement d'une architecture ainsi que leur interconnexion. Le graphe d'implémentation est ensuite définit en projetant le graphe d'algorithme sur le graphe d'architecture. Ce graphe d'implémentation est ensuite transformé en des exécutifs à l'aide des outils d'aide à l'implémentation . L'évaluation des algorithmes et des architectures matérielles requiert plusieurs outils. Un simulateur a été définit pour évaluer les différents systèmes et algorithmes dans toutes les situations possibles. De plus, un système HIL a été développé pour évaluer le système SLAM en incluant le matériel dans la chaîne de traitement. Finalement, les métriques, les méthodes de programmation, les techniques d'optimisation et les outils de synthèse utilisés ont été présentés. Nous allons voir l'application de cette démarche et les outils associés dans les deux chapitres qui suivent.

Chapitre 3

Étude algorithmique et choix des architectures

Sommaire

3.1 Introduction

Les algorithmes SLAM peuvent être classifiés en deux catégories : le SLAM en ligne et le full SLAM. Le SLAM en ligne consiste à calculer à chaque instant la position actuelle du robot par rapport aux mesures capteurs. Le full SLAM consiste à construire à chaque instant l'ensemble de trajectoire du robot. Au sein de ces deux catégories, les algorithmes SLAM sont basés soit sur une approche probabiliste, sur une structuration à partir du mouvement (SfM) ou bien ils sont bio-inspirée. L'approche probabiliste consiste à représenter l'incertitude du robot en utilisant une distribution probabiliste dans l'espace. Elle définit un ensemble des hypothèses sur la position du robot. L'approche SfM implique des traitements d'images pure pour l'estimation de la position de la caméra à partir de certaines observations multiples. L'approche bio-inspirée repose sur des concepts d'apprentissage de la nature et les appliquer à la conception des systèmes SLAM . Généralement deux types de capteurs extéroceptifs sont utilisés dans les algorithmes SLAM. Des capteurs à grand prix comme un capteur laser ou bien bas-coût comme une caméra. Les odomètres sont des capteurs proprioceptifs utilisés pour estimer la position du robot.

Notre première considération pour le choix des algorithmes SLAM à étudier, était de s'assurer à ce qu'ils couvrent une large gamme d'applications en robotique. Par conséquent, les algorithmes SLAM ont été choisis de tel sorte à ce que :
- ils abordent le problème du SLAM en ligne ou le full SLAM
- ils sont basés sur une approche probabiliste ou bien bio-inspirée
- ils utilisent soit des capteurs à coût élevé ou bien à faible coût.

Avec une telle méthodologie de choix à l'esprit, nous avons choisi quatre algorithmes de SLAM à évaluer dans notre étude : le FastSLAM2.0 , l'ORB SLAM, le RatSLAM et le SLAM linéaire.

Dans ce chapitre nous présenterons une étude algorithmique de chaque algorithme SLAM. Nous donnerons des évaluations fonctionnelles en utilisant un ensemble de données réelles. Ensuite, nous allons étudier la promesse tenue par des architectures embarquées bas-coût pour l'implémentation de ces algorithmes. Pour ce faire, nous utiliserons plusieurs architectures embarquées récentes afin de sélectionner l'algorithme le plus adapté pour un système SLAM répondant aux contraintes de temps réel, de performance et de consistance.

3.2 Prétraitement des données capteurs

3.2.1 Extraction des amers

La détection des amers est une phase de prétraitement importante pour tous les algorithmes SLAM. Les détecteurs des amers détectent un nombre minimal des amers à partir d'un grand flux de données brutes d'un capteur extéroceptif. Il existe dans la littérature plusieurs types d'amers en fonction du capteur extéroceptif utilisé. Pour les télémètres Laser, les amers sont : soit des lignes, des coins ou bien des maximas locaux qui correspondent à des mures ou des objets. Pour une camera, les amers sont : soit des contours, des objets de différentes apparences ou bien des amers sous forme de places comme des couloirs ou des intersections.

3.2.1.1 Détection à partir d'un flux de données Laser

Ils existe plusieurs méthodes pour l'extraction des amers à partir d'un flux de données Laser. Dans notre implémentation, les amers sont extraits en utilisant une segmentation et un filtre d'information (Information Filter IF en anglais). Le filtre d'information est un filtre Gaussien qui représente la densité

probabiliste par une représentation canonique. Autrement dit, la densité probabiliste est décrite par une matrice d'information et un vecteur d'information, ceci est équivalent à la matrice de covariance et le vecteur d'état dans le filtre de Kalman. La procédure d'extraction des amers à partir d'un flux de données laser est donnée en Annexe-7.3, section 7.3.1.

3.2.1.2 Détection à partir d'un flux de données images

Plusieurs algorithmes de détection de points d'intérêts existent : SURF [109], SIFT [110], Harris [111], Shi et Tomasi [112]. L'algorithme FAST (Features from Accelerated Segment Test) [113] détecte rapidement des points d'intérêts et produit des résultats suffisamment stable pour notre implémentation. De plus, des études ont démontrées que SURF donne les meilleurs résultats de détection, cependant son temps de traitement est incompatible avec une implantation embarquée temps-réel. De ce fait, nous avons choisi le détecteur FAST qui permet une détection rapide et fiable des amers. FAST utilise une image au niveau de gris. La procédure d'extraction des amers par FAST à partir d'une image monoculaire est décrite en Annexe 7.3, section 7.3.2.

3.2.2 Appariement des amers

La plus part des algorithmes de SLAM supposent que l'association entre l'observation et l'amer est connue. En pratique, et dans la plus part des cas l'association est inconnue, ce qui exige la mise en place d'un algorithme de mise en correspondance. Les amers sont utilisés par les algorithmes de SLAM pour cartographier l'environnement exploré et localiser le robot. Il est nécessaire donc de redétecter des amers ayant déjà été vus. Ceci est assuré par la phase de mise en correspondance entre des amers connus (déjà existant dans la carte) et de nouvelles observations. La redétection des amers permet au robot de se relocaliser dans l'environnement et d'améliorer la carte reconstruite. La mise en correspondance de points d'intérêts est un domaine très étudié et dépend fortement du type du capteur extéroceptif utilisé pour la perception de l'environnement. Classiquement, on utilise des méthodes de mesure de similarité pour la mise en correspondance. Pour le SLAM à base des télémètres Laser, on utilise souvent des méthodes probabiliste telle que le calcul de la distance de Mahalanobis ou bien la probabilité maximale. Pour le SLAM monoculaire à base d'une caméra on calcule des distances spécifiques à un type bien défini des descripteurs. Plus de détail sur le formalisme mathématique de la phase d'appariement est donné dans l'annexe7.3, section 7.3.3.

3.2.3 Étape d'initialisation

Les algorithmes SLAM ont besoin de connaître la position initiale d'un amer pour cartographier l'environnement exploré. L'initialisation d'un amer n'est pas une étape évidente, surtout dans le cas où une seule observation est insuffisante pour calculer la position du nouvel amer dans toutes les dimensions. Autrement dit, la méthode d'initialisation dépend du type du capteur extéroceptif utilisé. Pour les capteurs Laser, le modèle d'observation est inversible, du coups une seule observation suffit pour ajouter l'amer dans la carte [35] [114] [115]. Contrairement au capteur Laser, l'initialisation pour une caméra nécessite la mise en place d'un algorithme d'estimation de profondeur. Plus de détail sur les méthodes d'initialisation en fonction du capteur est donné dans l'annexe 7.3, section 7.3.4

3.3 Algorithme FastSLAM2.0

3.3.1 Approche de l'indépendance conditionnelle

Contrairement aux autres approches de SLAM basées sur l'estimation de la probabilité postérieure $p\left(s_t, m_t | z_t, u_t\right)$, le FastSLAM2.0 calcule une quantité légèrement différente donnée par $p\left(s^t, m_t | z_t, u_t\right)$. Cette subtile différence permet la factorisation de la probabilité postérieure sous forme d'un produit de termes simples. La probabilité que le FastSLAM2.0 essaie de résoudre peut être factorisée sous forme de produit de probabilités (3.1).

$$p\left(s_t, m_t | z_t, u_t\right) = p\left(s_t | z_t, u_t\right) \prod p\left(m_t | s_t, z_t, u_t\right) \tag{3.1}$$

où s_t est la position du robot à l'instant t, m est la carte stochastique, z_t est l'observation, et u_t est la loi de contrôle. Cette factorisation montre que la probabilité postérieure peut être divisée en un produit de probabilité de position $p\left(s_t | z_t, u_t\right)$ et N autres probabilités qui représentent l'estimation de la localisations des amers $p\left(m_t | s_t, z_t, u_t\right)$.

3.3.2 Structure de données

L'algorithme FastSLAM2.0 estime le premier terme de l'équation (3.1) par le filtre particulaire. Les N autres probabilités restantes sont estimées par le filtre de Kalman étendu (EKF). La structure de l'algorithme basée sur la forme factorisée (3.1) implique que l'estimation de la localisation des amers est conditionnellement indépendante. Les amers ne sont plus corrélés les uns aux autres, donc la taille de la matrice de covariance de l'EKF est réduite, puisque il estime la position d'un seul amer. Chaque particule dans le filtre particulaire est sous la forme :

$$\mathbf{S}_t^m = \{s_t^m, \mathrm{X}_1^m, C_1^m, \ldots, \mathrm{X}_n^m, C_n^m\} \tag{3.2}$$

m désigne l'indice de la particule dans le filtre, s_t^m est la position de la m-ième particule, X_1^m, C_1^m sont respectivement la localisation et la matrice de covariance du n-ième amer conditionné par la position de la particule s_t^m. Ces paramètres constituent la m-ième particule de l'ensemble $\boldsymbol{S}_t$ où M est le nombre total des particules. Le FastSLAM2.0 calcule l'ensemble $\mathbf{S}_t$ à l'instant t à partir de l'ensemble $\mathbf{S}_{t-1}$à l'instant précédent, ce qui engendre la génération de nouvelles particules tout en intégrant les données des capteurs proprioceptifs et extéroceptifs.

3.3.3 Échantillonnage des particules

La première étape de l'algorithme FastSLAM2.0 est la génération probabiliste d'un ensemble de particules à l'instant t. L'étape de prédiction exploite les données du capteur extéroceptif (odomètre dans notre cas) pour calculer la position future de toutes les particules dans le filtre. Autrement dit, la génération d'un ensemble $\boldsymbol{S}_t$ temporaire à partir de $\boldsymbol{S}_{t-1}$. Ceci est obtenu en utilisant le modèle d'évolution (3.3). Chaque odomètre retourne la distance (en nombre de pas) parcourue par une roue. On désigne par (d_r, d_l) respectivement la distance parcourue par la roue droite et gauche du robot mobile. Ce modèle nécessite une intégration d'un modèle de bruit qui reflète les erreurs de déplacement dans le système.

$$\boldsymbol{f}\left(s_t^m\right) = s_{t-1}^m + \begin{pmatrix} d_c \cos\left(\theta + \frac{\delta_\theta}{2}\right) \\ d_c \sin\left(\theta + \frac{\delta_\theta}{2}\right) \\ \delta_\theta \end{pmatrix} \tag{3.3}$$

Avec :

— d_r la distance parcourue par la roue droite.
— d_l la distance parcourue par la roue gauche.
— d_b la distance de la base qui sépare les deux roues motrices
— $d_c = \frac{d_r + d_l}{2}$ le déplacement longitudinale du centre de l'essieu arrière du robot.
— $\delta_\theta = \frac{d_r - d_l}{d_b}$ le déplacement angulaire du centre de l'essieu arrière du robot.
— s_t^m le vecteur de position de la m-ième particule.

3.3.4 Mise à jour de la position des particules

La précision du FastSLAM2.0 est reliée à la diversité des particules dans le filtre. Plus le filtre est riche en particules plus il est précis. Il peut corriger la position du robot et des amers si une observation est vue dans des instants un peu plus tard. Si le modèle d'évolution est bruité par rapport à l'observation, la trajectoire des particules prédite diverge. Ces particules vont recevoir un poids faible et seront donc supprimées dans la phase de rééchantillonnage ce qui conduit au phénomène de l'appauvrissement des particules. Le FastSLAM2.0 [116] résout ce problème en introduisant une modification dans le calcule de la distribution proposée (ensemble des particules générées après intégration des observations). Si un amer a été redétecté, l'ancienne distribution proposée résultante de la phase de prédiction est mise à jours. En effet, pour éviter de supprimer un nombre important des particules dans la phase de rééchantillonnage on calcule une nouvelle moyenne de distribution $\boldsymbol{\mu}$ et sa matrice de covariance $\boldsymbol{\Sigma}$ en se basant sur l'observation faite de manière incrémentale et pour chacune des particules dans le filtre.

3.3.5 Estimation des amers

Chaque particule possède sa propre carte modélisée par un arbre binaire (section 3.3.6). Chaque amer est représenté par un filtre de Kalman étendu de dimension réduite. Les coordonnées de l'amer sont modélisées par une simple paramétrisation (xyz) si le capteur extéroceptif utilisé est un capteur Laser, et par une paramétrisation par inverse de profondeur pour la camera monoculaire. La phase d'estimation corrige les amers appariés avec les observations dans la phase de mise en correspondance par le filtre de Kalman EKF. Le filtre de Kalman corrige les paramètres initiales des amers et leurs incertitudes pour chaque particule du filtre. Chaque filtre est de dimension réduite. La taille de la matrice de covariance est fixée puisque les amers ne sont pas corrélés. Le poids d'importance pour chaque particule est ensuite calculé par rapport à chaque amer apparié. Cette grandeur exprime la probabilité d'observer un amer à partir d'une particule bien déterminée.

3.3.6 Gestion de la carte et représentation par arbre binaire

Le FastSLAM2.0 a une complexité en $O(N.M.K)$, N est le nombre de particules, M est le nombre d'amers et K est le nombre d'amers observés. La complexité en M est inévitable, vu qu'il faut à chaque étape traiter toutes les particules dans le filtre. La complexité en N vient de l'étape de ré-échantillonnage. En effet, chaque amer présent dans la carte doit être mis en correspondance, et s'il

est apparié avec une observation, il doit être mis à jour pour chaque particule. De plus, lors du rééchantillonnage, il faut copier entièrement la carte de l'environnement, puisque les particules sont dupliquées. Une implémentation naïve de la carte peut augmenter considérablement le temps d'exécution de l'algorithme. Montemerlo *et al.* [117] proposent une implantation optimisée de l'algorithme en représentant la carte de l'environnement, de chaque particule, sous la forme d'un arbre binaire. (voir Annexe 7.3, section 7.3.5).

3.3.7 Rééchantillonnage

Cette étape reflète les observations sur la position des particules. En effet, on effectue un rééchantillonnage systématique on se basant sur les poids des particules calculés dans la phase d'estimation. Les particules qui ont reçu un poids faible sont supprimées tandis que ceux qui ont reçu un poids important sont gardées et dupliquées pour maintenir une cohérence dans le filtre. Si une particule est dupliquée, sa carte aussi est copiée pour constituer un nouveau ensemble de particules qui vont être utilisées pour les prochaines itérations de l'algorithme.

3.4 Algorithme ORB SLAM

3.4.1 Présentation du système

3.4.1.1 Les blocs majeur de l'ORB SLAM

L'algorithme ORB SLAM en tant que système se compose de trois tâches concurrentes : le suivi, la cartographie locale et la fermeture de boucle, Figure 3.1.

La tâche de suivi s'occupe de la localisation de la camera à chaque réception d'image et décide de rajouter ou non cette image au système. Il réalise un appariement entre l'image précédente et l'image courante et calcule la position de la caméra par un modèle d'évolution. Dans le cas ou le suivi est perdu, la phase de reconnaissance de place est lancée pour réaliser une relocalisation globale. Si le suivi est réussi et qu'on dispose d'une première estimation de la position de la caméra plus un ensemble des amers appariés, une carte locale est construite en utilisant le graphe de covisibilité. Une deuxième phase d'appariement est réalisée pour identifier des amers dans la carte locale en utilisant une procédure de projection, ensuite la position de la caméra est optimisée avec les amers appariés. Finalement, la tâche de suivi décide soit de sauvegarder l'image dans la scène soit de l'abandonner. [2]

La tâche de cartographie locale traite les nouvelles images acquises et exécute une compensation par faisceaux local (CF) pour aboutir à une reconstruction optimale de la carte. Une phase d'appariement est réalisée pour chercher des correspondances dans les images connectées les uns aux autres dans le graphe de covisibilité afin de permettre leur triangulation. Après l'initialisation des amers, une procédure de sélection est réalisée pour ne conserver que les points de haute qualité on se basant sur certaines informations récoltées par la tâche du suivi. [2]

La tâche de fermeture de boucle cherche des boucles potentielles à chaque fois qu'une image est acquise. Si une fermeture de boucle est détectée, on calcule une transformation de similarité qui donne une information sur le degré de glissement accumulé dans la boucle. Ensuite, les deux boucles sont alignées et les amers dupliqués sont fusionnés. Enfin, une optimisation du graphe de position est réalisée pour aboutir à une consistance globale.

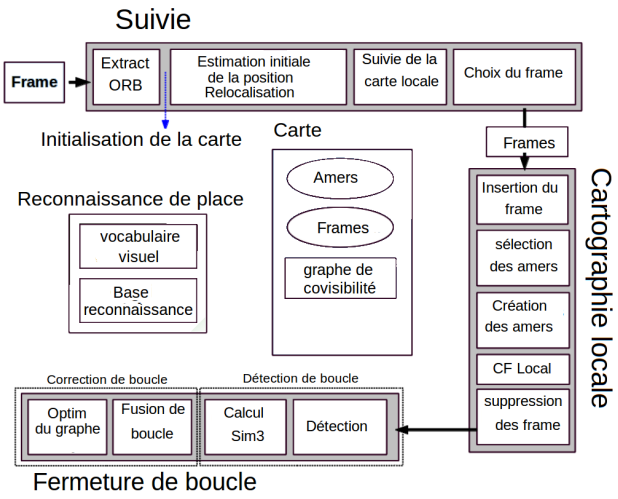


Figure 3.1: *Les différentes tâche du l'ORB SLAM*

3.4.1.2 Échantillonnage des amers et des images

La structure des amers dans le système ORB SLAM est composée de [2]

- $X_{W,i}$ la position 3D des amers dans le repère du monde
- L'angle de vue n_i , il s'agit d'un vecteur moyen de tous les angles de vues.
- Un descripteur représentative type ORB noté D_i, ce descripteur est choisit de telle sorte que la distance Hamming est minimale.
- d_{max} , d_{min} sont respectivement la distance maximale et minimale avec lesquelles l'amer peut être observé.

La structure d'une image notée K_i dans le système ORB SLAM est composée de :

- La position de la caméra notée T_{iw} , (transformation du repère du monde vers le repère de la caméra).
- Les paramètres intrinsèques de la caméra.
- Tous les amers avec leurs descripteurs ORB détectés dans cette image.

Les amers et les images sont choisis à partir d'une sélection stricte et rigoureuse en utilisant un mécanisme de suppression pour détecter les images dupliquées et les mauvais appariements des amers. Ceci permet une reconstruction flexible tout au long de la phase d'exploration qui améliore de plus la phase de suivi dans des conditions difficiles telle que la rotation et les mouvements brusques de la caméra.

3.4.1.3 Graphe de covisibilité et graphe essentiel

L'utilisation du graphe de covisibilité qui connecte les différentes images acquises est importante surtout pour un système de SLAM purement visuel [118]. Chaque nœud du graphe représente une image, les transitions du graphe lient deux images partageant au moins 15 amers. Pour corriger une boucle fermée, une optimisation de graphe est réalisée [119]. Pour éviter d'inclure toutes les transitions du graphe qui sont nombreuses et ne pas alourdir la représentation du graphe, l'ORB SLAM utilise un graphe essentiel qui maintient plusieurs images et moins de transition. Le système construit de manière incrémentale un arbre à partir des images initiales qui fournissent un sous graphe du graphe de covisibilité avec un minimum de transition. Si une image acquise est rajoutée dans le système, elle est incluse dans l'arbre et liée avec les autres images qui partagent certaines observations. Si une image est supprimée par la procédure de suppression, le système met à jour les transitions affectées par

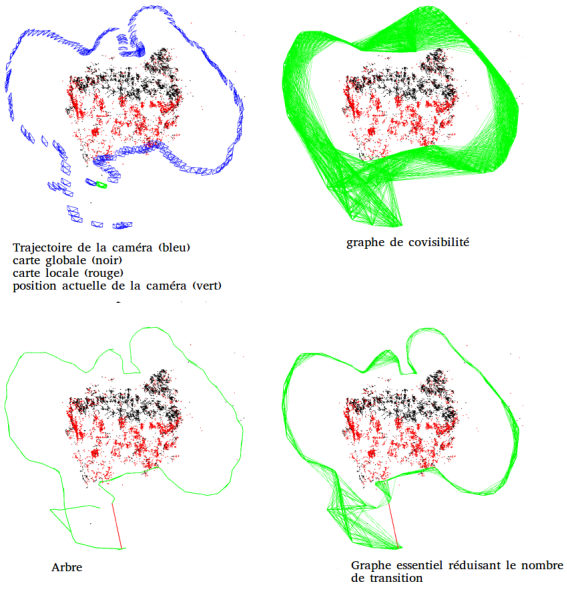


Figure 3.2: *Graphe de covisibilité, graphe essentiel et l'arbre correspondant [1]*

l'image supprimée. La figure 3.2 montre un exemple du graphe de covisibilité, l'arbre correspondant et le graphe essentiel.

3.4.1.4 Ensemble des mots et reconnaissances des places

L'ORB SLAM embarque un ensemble de mot utilisé pour la reconnaissance de place basé sur le module DBoW [120]. Ce module est utilisé principalement pour la détection de boucle et pour réaliser une relocalisation. Le mot visuel, s'agit d'une discrétisation de l'espace du descripteur. Ce vocabulaire est crée en mode hors ligne en utilisant le descripteur ORB et à partir une séquence très large des images. L'ensemble des mots sert à améliorer l'efficacité de la mise en correspondance des amers, comme mentionné dans [120]. Autrement dit, pour calculer la correspondance entre deux amers identifiés par le descripteur ORB, on peut limiter la zone de recherche pour les amers qui appartiennent au même nœud de l'arbre vocabulaire ce qui permet d'accélérer le processus de mise en correspondance.

3.4.2 Tâche de suivi

3.4.2.1 Extraction par ORB

Cette étape extrait et détecte les amers dans l'image courante. La détection des amers est effectuée en utilisant le détecteur FAST. Pour assurer une détection fiable et une distribution cohérente des observations, l'image est divisée en grilles pour détecter des observations dans chaque cellule. Pour chaque amer détecté, on calcule l'orientation et le descripteur ORB pour identifier l'amer pour la phase de l'appariement.

3.4.2.2 Estimation initiale de la position et relocalisation

Cette étape calcule une estimation initiale de la position de la caméra ou bien effectue une relocalisation selon l'état courant du système. En effet, Si le suivi de l'image courante est réussi, la position de la caméra est prédite en utilisant un modèle du mouvement à vitesse constante. Dans le cas où le

suivi de l'image courante est perdu, un processus de relocalisation est mis en place pour retrouver la position de la caméra.

3.4.2.3 Suivie de la carte locale

Une fois la position initiale de la caméra est estimée, la carte est projetée dans l'image courante pour chercher plusieurs correspondances. Afin de minimiser la complexité, les correspondances sont recherchées en utilisant juste la carte locale. La carte locale contient l'ensemble des images K_1 qui partagent des amers avec l'image courante et un ensemble des images K_2 qui contient des amers voisins de l'ensemble K_1 dans le graphe de covisibilité. La carte locale contient aussi une image de référence $K_{ref} \in K_1$ qui partagent la plus part des amers avec l'image courante. Le suivi est assuré par une méthode de recherche active (voir annexe 7.3, section 7.3.7).

3.4.2.4 Décision d'ajout d'une nouvelle image

La dernière étape de la tâche du suivi est de décider soit de considérer l'image courante dans le système où non. Comme il existe déjà une étape dans la phase de cartographie locale qui supprime les images courantes non conformes, l'image est rajoutée dans le système le plutôt possible pour rendre la tâche du suivi plus robuste contre les mouvements brusques de la caméra, typiquement les rotations.

3.4.3 Cartographie locale

3.4.3.1 Insertion d'une nouvelle image

Tous d'abord, le graphe de covisibilité est mis à jour, on y rajoute la nouvelle image tout en mettant à jour les transitions du graphe. Ensuite on met à jour l'arbre en liant l'image courante K_i avec l'image qui partage la plus part des amers observés. Après on calcule l'ensemble des mots correspondant à la nouvelle image. Ceci renforce la robustesse de la phase d'appariement pour trianguler les nouveaux amers.

3.4.3.2 Sélection des amers

Pour ajouter un nouveau amer à la carte du système, il doit satisfaire certaines conditions. Il doit être suivi dans au moins 25 images dans lesquelles il est supposé d'être visible. En plus, il doit être observé dans au moins 3 dernières images consécutives. Ceci pour assurer que les amers soient initialisés correctement.

3.4.3.3 Création des nouveaux amers

Un amer est initialisé dans la carte par sa triangulation à partir des images liées à l'image K_c dans le graphe de covisibilité. Pour chaque amer non apparié dans l'image courante K_i, on cherche une correspondance avec des amers non appariés dans d'autre images. Pour initialiser un amer dans la carte, sa profondeur doit être positive. La procédure d'initialisation est décrite dans l'annexe7.3, section 7.3.4.2.

3.4.3.4 Compensation par faisceaux

Durant le traitement d'une image donnée, toutes les images qui lui sont associées dans le graphe de covisibilité et les points observés dans ces images sont optimisées en utilisant un processus de

compensation par faisceaux local. Ce processus d'optimisation sert à annuler toutes les données et observations aberrantes.

3.4.3.5 Suppression des images

Afin de maintenir une reconstruction dense et compacte, les images redondantes sont détectées et supprimées. Les images dont 90% de leurs amers ont été précédemment vus dans les trois dernières images sont alors rejetées et supprimées.

3.4.4 Fermeture de boucle

3.4.4.1 Détection de fermeture de boucle

Dans un premier temps on calcule la similarité entre le vecteur de l'ensemble des mots de l'image K_i avec celui des images voisines de K_i dans le graphe de covisibilité, et on retient la valeur minimal s_{min}. On récupère la base de reconnaissance et on exclut toutes les images dont leur similarité est inférieur à s_{min}. En outre, toutes les images qui sont directement liées à K_i sont exclues du test. Pour accepter une fermeture de boucle éventuelle, il faut détecter trois fermetures de boucle consécutives consistantes (les trois images fermant la boucle doivent être connectées dans le graphe de covisibilité). Ceci pour éviter toute ambigüité puisqu'ile se peut qu'il y aie plusieurs emplacements avec une apparence similaire à l'image K_i.

3.4.4.2 Calcule de la transformation de similarité

Dans un système SLAM monoculaire, la carte peut glisser par rapport à 7 facteurs possibles : trois translations, trois rotations et un facteur d'échelle. Par conséquence, pour fermer la boucle il faut calculer une transformation de similarité à partir de l'image courante K_i vers l'image candidate K_l fermant la boucle. Cette transformation donne une information sur l'erreur accumulée dans la boucle.

3.4.4.3 Fusion de boucle

La première étape dans la phase de correction de boucle est de fusionner les amers dupliqués et de rajouter une nouvelle transition dans le graphe de covisibilité qui va lier la fermeture de boucle. Premièrement, la position T_{iw} de l'image courante est corrigée en utilisant la transformation de similarité S_{il}. Cette correction est propagée dans toutes les images voisines de K_i. Ceci assure que les deux côtés de boucle sont alignées les uns aux autres. On effectue ensuite une projection des amers observés dans l'image candidate sur l'image courante K_i, et on effectue une recherche de correspondance. Les amers appariés sont fusionnés. Toutes les images concernées par la procédure de fusion ont leur transition mise à jour dans le graphe de covisibilité afin de créer des transitions qui interprètent efficacement la fermeture de boucle.

3.5 Algorithme Bio-inspiré : RatSLAM

Dans cette section nous présenterons un nouveau algorithme, RatSLAM, qui dérive du modèle complexe hippocampique chez les rongeurs [57] . Le système RatSLAM utilise un modèle de calcul approximatif basé sur les réseaux à attracteurs compétitifs. Celui-ci, suit d'autres modèles récents de calcul de l'hippocampe des rongeurs qui utilisent les réseaux à attracteurs compétitifs comme base pour la représentation [121]. Le paquet de l'activité du réseau à attracteurs compétitifs représente la distribution probabiliste du robot en ce qui concerne sa propre pose. Le mouvement du robot module

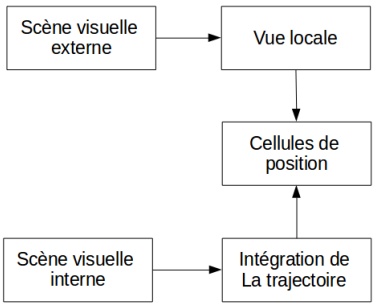


Figure 3.3: *La position est illustrée par une activité dans la cellule de position. Cette dernière est mise à jour de façon continue*

la dynamique du réseau en provoquant un changement du paquet de l'activité et par conséquent une mise à jour de l'estimation de pose. Les signaux sensoriels sont associés à des paquets de l'activité. Une fois que les associations entre les signaux sensoriels et l'estimation de position sont appris, les signaux sensoriels auront une influence sur la position du paquet de l'activité pour mettre à jour l'estimation de la pose du robot. En utilisant une structure du réseau à attracteurs compétitifs, le RatSLAM génère une représentation qui combine une partie grille et une autre partie topologique. Les éléments qui sont proches du réseau risquent d'être proches dans l'espace, mais la connectivité actuelle et le sens du réseau sont définis par le comportement du robot entre les éléments. En outre, le système possède l'une des forces des systèmes basés sur les amers visuels. RatSLAM peut prendre une entrée visuelle ambiguë puis maintient et propage plusieurs hypothèses de position simultanément. La dynamique des réseaux permet à ces hypothèses de rivaliser entre elles jusqu'à ce que l'entrée visuelle au cours de la compétition peut renforcer la distribution probabiliste en une ou plusieurs hypothèses de position possibles.

3.5.1 L'architecture du RatSLAM

3.5.1.1 Le système en général

La figure 3.3 représente le modèle de base du RatSLAM. La position du robot est représentée par l'activité dans un réseau à attracteurs compétitifs appelés les cellules de position. Les informations issues des encodeurs de roue effectuent une intégration du chemin en injectant l'activité dans les cellules de position et en déplaçant ainsi les paquets d'activité actuelle. Les informations visuelles sont converties en une représentation de vue locale. Dans le cas où cette représentation est connue, elle injecte l'activité dans les cellules particulières qui sont associées à ce point de vue local spécifique [57]
.

3.5.1.2 Les cellules de position

Les cellules de position sont implémentées comme un réseau à attracteur compétitif, un type de réseau neuronal qui est conçu pour converger vers un modèle stable d'activation dans l'ensemble de ses unités. Les unités du réseau peuvent être organisées dans de nombreuses configurations, mais généralement chaque unité va exciter les unités qui lui sont proches et inhibent celles plus loin, ce qui conduit à un ensemble d'activité connu sous le nom "paquet d'activité éventuellement à dominer". L'activité qui sera injectée dans le réseau près du paquet dominant aura tendance à déplacer ce paquet vers elle. Par contre, celle qui sera injectée loin du paquet dominant, elle créera un autre paquet qui

rivalise avec l'original. Si l'activité est injectée suffisamment, le nouveau paquet peut "dominer" et le vieux paquet disparaît.

Le système RatSLAM utilise des cellules de position qui représentent simultanément les distributions probabilistes sur l'emplacement et l'orientation du robot. L'intégration de l'emplacement et l'orientation dans un seul réseau diffère sensiblement des autres modèles de l'hippocampe des rongeurs [122]. Les expériences qui ont été menées sur des rongeurs réels ont montré que certaines cellules répondent au maximum quand un rat est à un certain endroit (cellules de code de lieu) et que d'autres réagissent lorsqu'il est orienté dans une direction donnée (cellules de la direction de la tête). Ces résultats ont incité l'utilisation des réseaux à attracteurs compétitifs distincts pour le code de lieu (x, y) et la direction de la tête (téta). Ces systèmes ont une limite fondamentale : ils ne peuvent pas représenter et maintenir des distributions probabilistes multiples en position pendant un laps de temps. Ce phénomène a été précédemment illustré dans [123].

En représentant (x, y, θ) dans le même réseau à attracteurs compétitifs, le système peut gérer en même temps plusieurs distributions probabilistes de position au cours du temps. Nous arrangeons les cellules de position dans un arrangement (x, y, θ) afin de faciliter la visualisation bien qu'il n'y a aucune justification biologique pour ce genre d'arrangement ordonné. Cet arrangement simplifie également l'affectation de poids pour l'intégration du chemin.

3.5.1.3 Intégration du chemin

L'intégration du chemin d'accès n'est pas censée être strictement cartésienne. La relation entre la distance, l'angle et les différentes unités n'a été que partiellement réglée afin d'aider à la visualisation et l'affectation des poids plutôt que d'assister le fonctionnement du réseau. Chaque cellule occupe environ $0.25\ m \times 0.25\ m$ dans la zone et environ $9°$ en rotation. La pondération des connexions entre les unités au sein du réseau de cellules de position est décrite dans la section suivante.

Le caractère important de la représentation des cellules de position signifie que l'intégration du chemin d'accès basée uniquement sur le réseau des cellules de position est inférieure à celle obtenue avec la simple intégration de l'odométrie. Ce sont les propriétés topologiques ainsi que la relation de la pose et les amers visuels qui maintiennent la cohérence et la stabilité dans la représentation de la position [57].

3.5.1.4 Vue locale

La caméra et le module du traitement de vision du robot peuvent voir des cylindres colorés et indiquer la distance et la rotation relative au cylindre ainsi que les incertitudes associées [124]. Une matrice tridimensionnelle des cellules de vue locale encode la couleur du cylindre (type), la distance et la rotation. Les vues locales activées des cellules sont constamment associées aux cellules de position qui sont fortement activées à ce moment là grâce au renforcement des connexions pondérées entre elles. Bien que l'un des paramètres visuels est la distance vers un cylindre, il n'y a aucune interprétation géométrique de la distance à un amer visuel dans le système.

3.5.2 La dynamique du RatSLAM

3.5.2.1 Processus d'association visuelle

Le processus d'association visuelle est la clé pour maintenir des représentations de position cohérentes face aux représentations incohérentes qui se présenteront dans le processus d'intégration du

chemin important. Les forces de connexion entre les cellules de vue locale et les cellules de position sont renforcées grâce à l'apprentissage Hebbien, par l'équation 3.4.

$$\beta^{t+1}_{(ijk)(lmn)} = \beta^{t}_{(ijk)(lmn)} + \eta P_{lmn} V_{ijk} \tag{3.4}$$

Le taux d'apprentissage n'est pas critique au fonctionnement et a été arbitrairement fixée à 0.05. Étant donné que seul un petit pourcentage de toutes les connexions aura des poids non nuls, nous encodons ces poids de façon superficielle.

Une connectivité complète entre l'implémentation actuelle est environ 700 cellules de vue locale et 180000 cellules de position qui exigerait 1.3 x 10^8 connexions. Cependant, lors d'une expérience en temps réel, une heure exige seulement environ 800.000 connexions qui ont des poids non nuls qui demandent une mémorisation importante dans le calcul. Quand une scène familière est rencontrée, les cellules de vue locale activées projettent une énergie tout au long des connexions pondérées dans les cellules de pose (3.5). La quantité d'énergie projetée à partir des cellules de vue locale est limitée en fournissant une limite stricte sur le changement de chaque unité des cellules de pose, σ .

$$\triangle P_{ijk} = min\left(\sum_{l=0}^{x}\sum_{m=0}^{Y}\sum_{n=0}^{Z}\beta_{(ijk)(lmn)}V_{lmn}, \sigma\right) \tag{3.5}$$

3.5.2.2 Intégration du chemin

Le processus d'intégration du chemin projette l'activité des cellules de pose dans des cellules légèrement décalées par rapport à celles qui sont actuellement activées. Dans le cas où le robot effectue une activité de translation, l'activité est décalée dans le plan (x, y) sinon si le robot effectue une activité de rotation, l'activité cette fois-ci est décalée dans la direction thêta. La translation de la direction du mouvement de l'activité dépend de la position de la cellule dans la direction de thêta. L'amplitude du mouvement dans le plan (x, y) est fonction de la vitesse de translation v.

3.5.2.3 La dynamique des attracteurs compétitifs

Après les processus visuel et intégration du chemin, les cellules de position subissent le processus dynamique interne de l'attracteur compétitif . La dynamique des attracteurs compétitifs veille à ce que l'activité totale dans les cellules de position reste constante. Ceci est cohérent avec l'interprétation des cellules de position comme une distribution de probabilité de pose. Les paquets d'activité situés près de chaque mouvement vers l'autre rassemblent des représentations de pose similaires. Les paquets d'activité séparés représentent plusieurs hypothèses de position en concurrence les uns avec les autres.

1) Mise à jour de la couche interne X-Y Une distribution gaussienne discrète à deux dimensions a été utilisée pour créer les poids des excitateurs, Les connexions pondérées projettent l'activité de chaque cellule P aux autres cellules dans le N_x par la couche N_y.

$$\triangle P_{jk} = \sum_{a=0}^{N_x}\sum_{b=0}^{N_y}\epsilon_{(a-i)(b-k)}P_{ab} \tag{3.6}$$

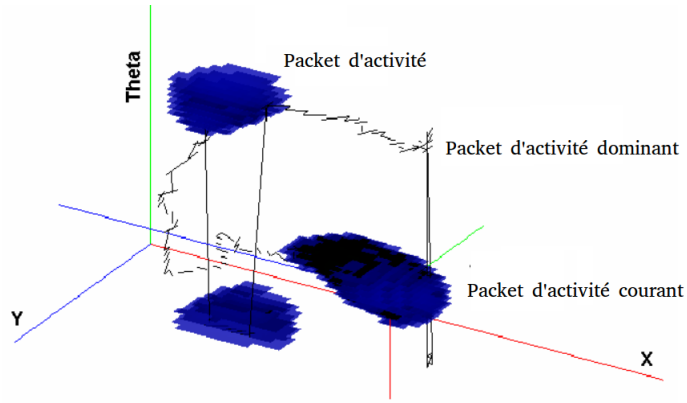


FIGURE 3.4: *Dynamique des attracteurs compétitifs*

2) Mise à jour de l'inter-couche Une distribution gaussienne à une dimension est utilisée pour former les poids de δ , qui provoque une excitation entre les couches. Le champ d'influence d'une couche est d'environ 45° (ou deux couches de chaque côté) - fixé par γ.

$$\triangle P_{ijk} = \sum_{c=k-\gamma}^{k+\gamma} \delta_{(c-k)} P_{ijc} \tag{3.7}$$

Les connexions entre les couches représentent des liens entre les cellules avec des orientations angulaires similaires. Ainsi, il existe une enveloppante de connexions dans la direction de thêta - la couche en haut excite à la fois les couches directement en dessous et les couches en «bas» du diagramme.

3) L'inhibition globale Bien que plusieurs hypothèses de position (représentées par plusieurs paquets d'activité) ont besoin de temps pour concurrencer et être renforcée par une autre entrée visuelle, l'inhibition est relativement douce et les paquets rivaux peuvent coexister pendant de longues périodes de temps. Le niveau d'inhibition diminue avec l'augmentation de l'activation des cellules. L'inhibition des contrôles constants ϕ du niveau d'inhibition globale est fixée à 0.004. Les niveaux d'activation sont limités à des valeurs non négatives.

$$P_{ijk}^{t+1} = max\left[P_{ijk}^{t} + \phi\left(P_{ijk}^{t} - max\left(P\right)\right), 0\right] \tag{3.8}$$

4) Normalisation La dernière étape est une normalisation qui maintient le niveau de l'activation totale après l'entrée visuelle et l'intégration du chemin.

$$P_{ijk}^{t+1} = \frac{P_{ijk}^{t}}{\sum_{x=0}^{N_x}\sum_{y=0}^{N_y}\sum_{z=0}^{\theta} P_{xyz}^{t}} \tag{3.9}$$

Un exemple de la dynamique des attracteurs compétitifs en fonctionnement est illustré sur la Fig. 3.4.

3.6 Le SLAM Linéaire

Le SLAM à large échelle nécessite à adhérer un ensemble des sous-cartes locales de manière séquentielle ou bien à travers la technique « Diviser et Régner ». Ceci peut être obtenu à travers la résolution d'une séquence des problèmes à moindre carré linéaires. Le SLAM Linéaire est une approche linéaire destinée à résoudre le problème SLAM à large échelle. Il peut être appliqué aux SLAM qui construisent des cartes d'environnement sous forme de graphe ou bien sous forme d'un ensemble des amers. Il s'agit d'une nouvelle approche qui consiste à fusionner un ensemble de sous-cartes locales en résolvant des problèmes à moindre carré linéaires et en faisant des changements de repère [125].

3.6.1 Structure des cartes locales

Chaque carte locale est définie par un vecteur d'état $\hat{\mathrm{X}}_{\mathrm{L}}$ et une matrice de covariance $\mathrm{P_L}$. Le vecteur d'état contient la position du robot finale estimée $\left(x_r^L, y_r^L, \theta_r^L\right)$ et l'ensemble des amers observés $\left\langle x_1^L, y_1^L, ..., x_n^L, y_n^L \right\rangle$ (3.10). La matrice de covariance reflète la corrélation entre les différents éléments de la carte (3.11). $\mathrm{P_r}$ est la covariance de la position du robot, $\mathrm{P_{f_1}}$ la covariance de la position du premier amer, $\mathrm{P_{rf_1}}$ représente la corrélation entre la position du robot et la position de l'amer et $\mathrm{P_{f_1 f_2}}$ représente la corrélation entre les amers de la carte.

Le système des cordonnées d'une carte locale est défini par la position du robot au début de la construction, le mouvement du robot commence à l'origine du repère de la carte locale. Étant donné que le robot commence la construction de la carte locale $k+1$ une fois la construction de la carte locale k est finie, la position finale du robot de la carte locale k est considérée comme la position initiale de la carte locale $k+1$. Ceci est illustré dans la figure 3.5. La génération des sous cartes locales peut se faire à travers différentes techniques de SLAM existantes. Généralement les cartes locales sont obtenues en utilisant le filtre de Kalman étendue (EKF).

$$\mathrm{X_L} = \left(\mathrm{X_r}, \mathrm{X_1^L}, ..., X_2^L\right) = \left(x_r^L, y_r^L, \theta_r^L, x_1^L, y_1^L, ..., x_n^L, y_n^L\right) \tag{3.10}$$

$$\mathrm{P_L} = \begin{bmatrix} \mathrm{P_r} & \mathrm{P_{rf_1}} & \mathrm{P_{rf_2}} \\ \mathrm{P_{rf_1}^T} & \mathrm{P_{f_1}} & \mathrm{P_{f_1 f_2}} \\ \mathrm{P_{rf_2}^T} & \mathrm{P_{f_1 f_2}^T} & \mathrm{P_{f_2}} \end{bmatrix} \tag{3.11}$$

3.6.2 Principe de fusion de deux cartes locales

La procédure traditionnelle de fusion de deux cartes locales est illustrée dans la figure 3.6. La carte 1 est construite par rapport au système de cordonnées définie par la position initiale P_0, elle contient un ensemble d'amers et la position finale du robot P_1. La carte 2 est construite dans le système de cordonnées défini par la position initiale P_1 et contient un ensemble d'amers et la position finale du robot P_2. La position finale du robot dans la carte 1 est la même que la position initiale dans la carte 2 (les deux sont P_1). Le système de cordonnées de la carte globale est défini par P_0. La fusion de ces deux cartes s'appelle le problème d'optimisation non-linéaire [126]. Contrairement à la méthode traditionnelle pour la fusion de deux cartes, le SLAM linéaire construit la carte (1,2) comme suivant :

la carte 1 est construite en prenant P_2 (la position finale du robot dans la carte locale 2) comme origine du système de cordonnées et contient un ensemble d'amers et la position initiale du robot P_0.

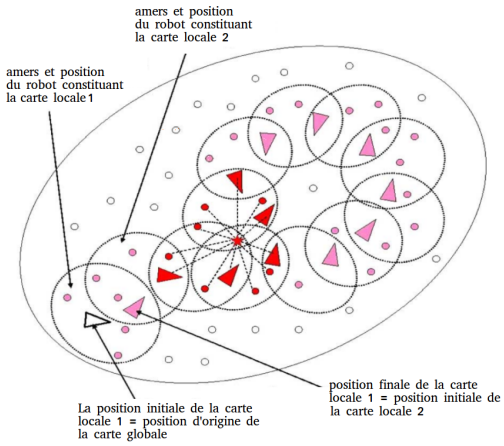


Figure 3.5: *Structure des cartes locales : la grande ellipse représente l'environnement exploré, les petites ellipses contiennent la position du robot et les amers partageant la même sous carte. La carte globale finale contient tous les amers détectés et la position du robot correspondante.*

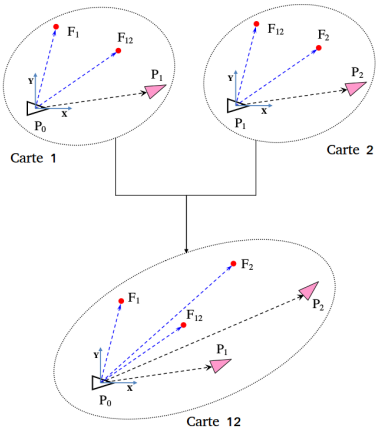


Figure 3.6: *La méthode traditionnelle pour la fusion de deux cartes locales*

La carte 2 est construite en prenant P_1(position initiale de la carte 1) comme origine du repère, elle contient les amers observés et la position finale du robot P_2. Contrairement à la méthode traditionnelle, l'origine du repère de la carte résultante 12 est fixé par P_2 (la position finale du robot dans la carte locale 1), figure 3.7. Avec ces modifications, la nouvelle méthode de fusion des deux cartes 1 et 2 (figure 3.7), demeure un problème d'optimisation non-linéaire. Il est équivalent à une optimisation à moindre carré linéaire plus des changements de repère non-linéaires.

3.6.3 Fusion d'une séquence de cartes locales

Le SLAM linéaire se base sur la solution optimale linéaire (détaillée dans l'annexe 7.3, section 7.3.6) pour la fusion de deux cartes locales. Il existe deux méthodes généralement utilisée par le SLAM linéaire pour résoudre le problème SLAM à large échelle à savoir : la méthode séquentielle ou bien la méthode « Diviser et Régner ».

La méthode Diviser et Régner La méthode diviser et régner proposée par [125] pour fusionner deux cartes locales est illustrée dans la figure (3.8). Notons que :

— Les deux cartes locales 1 et 2 sont construites dans le même système de cordonnées. Les deux

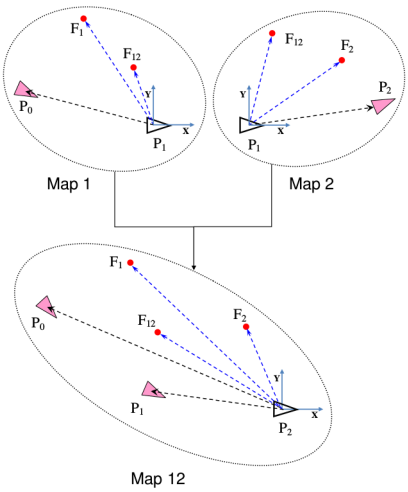


Figure 3.7: *Nouvelle approche dans le contexte SLAM linéaire pour la fusion de deux sous-cartes locales*

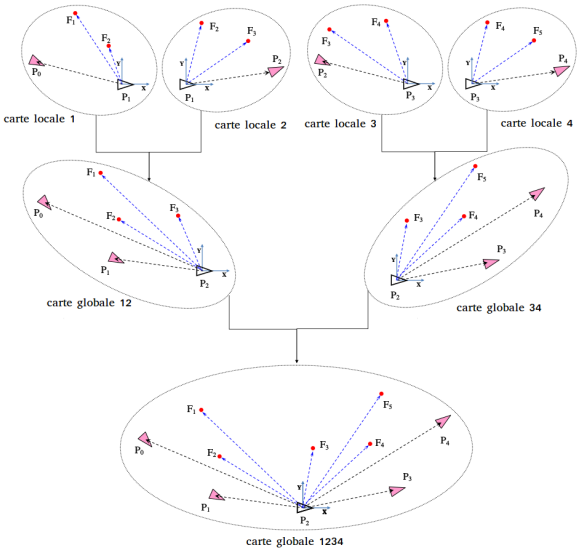


Figure 3.8: *La méthode "Diviser et Régner" pour la fusion de deux cartes*

cartes 3 et 4 sont aussi construites dans le même système de coordonnées.

— La carte globale 12 est obtenue en fusionnant les deux cartes locales 1 et 2. Cette carte est construite par rapport au repère dont l'origine est la position finale de la carte locale 2.

— La carte globale 34 est obtenue en fusionnant les deux cartes locales 3 et 4. Cette carte est construite par rapport au repère dont l'origine est la position initiale de la carte locale 3.

— Les deux cartes globales résultantes 12 et 34 sont dans le même système de cordonnées et peuvent être fusionnées par le SLAM linéaire.

La méthode séquentielle La méthode séquentielle est illustrée dans la figure 3.8, mais cette fois ci, la carte locale 3 est construite dans le système de cordonnées défini par sa position initiale. Ensuite, toutes la cartes locales (1,2,3,4) sont fusionnées de manière séquentielle. Notons que :

— La carte locale 1 est construite par rapport au système de cordonnées défini par sa position finale au lieu de sa position initiale.

— La carte globale 12 résultante de la fusion des deux cartes locales 1 et 2 est dans le repère défini

par la position finale de la carte locale 2.

— La carte globale 12 peut être fusionnée par le SLAM linéaire avec la carte locale 3.

3.7 Évaluation de l'aspect fonctionnel des algorithmes

Après avoir étudié le formalisme mathématique de chaque algorithme SLAM, nous présenterons dans cette section les évaluations fonctionnelles en utilisant des jeux de données réelles. Les algorithmes étudiés se basent sur différentes approches algorithmiques et utilisent une variété de capteurs. Nous utiliserons différents jeux disponibles dans la littérature pour évaluer ces algorithmes. Le FastSLAM2.0 et le SLAM linéaire ont été évalués en utilisant les données Laser et odométriques des encodeurs des roues fournies par le jeu de données Rawseeds [101] . L'ORB SLAM a été évalué en utilisant des images monoculaire fourni par le jeux de données KITTI [104] , la localisation de la caméra est assurée par odométrie visuelle. Finalement, le RatSLAM a été évalué en utilisant les données des images monoculaire et odométriques fournies par le jeu de données Oxford [103]. Les figures 3.9, 3.10, 3.11 et 3.12 montrent les résultats de localisation respectivement du FastSLAM2.0, le SLAM linéaire, l'ORB SLAM et le RatSLAM. Le FastSLAM2.0 utilisant des données Laser est capable de cartographier un environnement tout en produisant des résultats de localisation correcte et qui demeure stable par rapport à la référence. Cependant, la version monoculaire du FastSLAM2.0 nécessite une analyse algorithmique approfondie pour l'adapter aux environnements larges. La figure 3.10 montre les résultats de localisation et de cartographie fournis par le SLAM linéaire. Les résultats sont obtenus en utilisant 800 sous-cartes locales comme entrée pour l'algorithme. Ces cartes locales sont construites à partir des amers observés et les positions estimées du robot. La technique utilisée pour la construction des cartes locales est le filtre du Kalman étendu avec une matrice de covariance de taille réduite. Les résultats de localisation obtenus montrent que le SLAM linéaire est capable de fournir une carte globale consistante de l'environnement en réalisant le full SLAM par fusion des sous-cartes locales. Les résultats obtenus par l'ORB SLAM sur le jeu de données KITTI confirme la possibilité de cartographier un large environnement en réalisant un SLAM visuel monoculaire tout en produisant une localisation correcte de la caméra par odométrie visuelle. La trajectoire est corrigée d'avantage par un processus de fermeture de boucle fiable et robuste grâce à la mise en place de la reconnaissance des places. Les résultats du RatSLAM sur le jeu de données Oxford montrent qu'une approche bio-inspirée est capable de fournir une bonne localisation du robot en environnement large en produisant une carte topologique de l'environnement exploré qui pourra contribuer à la planification de la trajectoire.

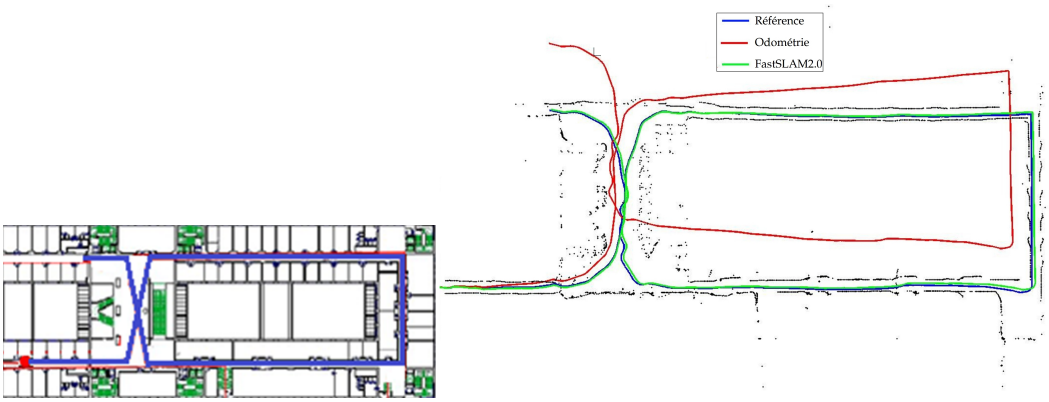


Figure 3.9: *Résultats de cartographie et de localisation du FastSLAM2.0 Laser avec le jeu de données Rawseeds*

Figure 3.10: *Résultats de cartographie et de localisation du SLAM linéaire avec le jeu de données Rawseeds*

Figure 3.11: *Résultats de cartographie et de localisation de l'ORB SLAM avec le jeu de données KITTI [2]*

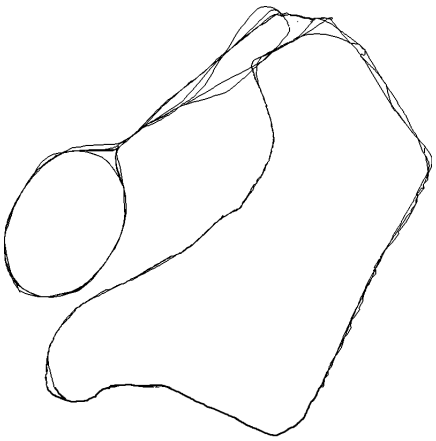


Figure 3.12: *Résultats de de localisation du RatSLAM avec le jeu de données Oxford [3]*

3.8 Découpage en blocs fonctionnels

En général, les algorithmes SLAM sont des processus dynamiques qui n'ont pas un temps d'exécution fixe et qui peut être prévu à l'avance. Afin d'évaluer leur temps d'exécution, ces algorithmes ont

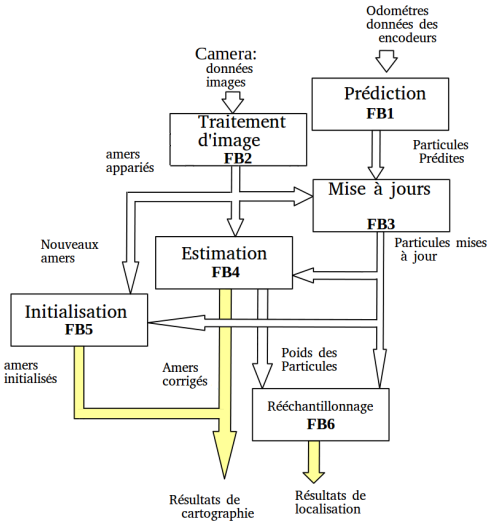


Figure 3.13: *Organigramme des blocs fonctionnels du FastSLAM2.0*

été analysés en termes d'instructions et d'ordre d'opérations. Cette étude nous a permis de deviser les algorithmes en sous-parties appelées blocs fonctionnels (BFs). Cette étude permet l'identification des tâches nécessitant un temps de calcul important. Il est basée sur plusieurs étapes : nous analysons tous d'abord le temps d'exécution des tâches et leur dépendances des données et des paramètres de l'algorithme. Un seuil est fixé pour chaque paramètre. L'algorithme est ensuite découpé sous forme de blocs fonctionnels exécutant un calcul bien défini. Chaque bloc est ensuite évalué pour déterminer son temps de traitement. Les figures 3.13, 3.14, 3.15 et 3.16 représentent la répartition en blocs fonctionnels des algorithmes SLAM étudiés.

Définition des blocs fonctionnels du FastSLAM2.0 Le FastSLAM2.0 est découpé en 6 blocs fonctionnels (BFs). Le bloc de prédiction (FB1) est le premier bloc exécuté à chaque itération et après chaque acquisition des données odométriques. Il exploite ces dernières pour calculer la position future du robot partant d'une position initiale et en se basant sur le modèle d'évolution. Le bloc FB2 traite les images acquises à chaque itération de l'algorithme, il est exécuté dans deux phases différentes à savoir la détection et la mise en correspondances. Le bloc FB3 calcule une nouvelle distribution probabiliste pour chacune des particules à partir d'une matrice de covariance initiale P_m calculée dans le bloc FB1. Cette distribution est calculée de façon incrémentale et pour chaque amer apparié. Une fois la distribution est calculée, une nouvelle position et une nouvelle incertitude des particules sont échantillonnées. Le bloc FB4 effectue une mise à jour de la carte reconstruite tout au long de la trajectoire explorée. En effet, la position des amers appariés dans l'image est calculée en utilisant le modèle Pinhole. L'innovation est ensuite calculée entre la position image des amers appariés et leur correspondances observées. Les équations du filtre de Kalman sont ensuite appliquées pour mettre à jour la position des amers et leurs incertitudes. Ce bloc aussi calcule le poids de chaque particule par rapport aux amers appariés. Le bloc FB5 augmente la taille de la carte et ajoute de nouveaux amers. En effet, pour les amers qui ne se sont pas appariés dans le bloc FB2, leurs positions et leurs incertitudes dans l'environnement sont calculées. Plus précisément, ce bloc calcule les paramètres du nouvel amer correspondant à une initialisation par inverse de profondeur. L'initialisation des amers est effectuée pour chaque particule qui dispose de sa propre carte. Le dernier bloc FB6, réalise un rééchantillonnage des particules pour générer un nouveau ensemble qui tient en compte des amers observés.

Définition des blocs fonctionnels de l'ORB SLAM L'ORB SLAM dispose de trois tâches principales qui s'exécutent en parallèles : le suivi, la cartographie locale et la fermeture de boucle. Chaque tâche est découpée en blocs fonctionnel exécutant un calcul bien spécifique. La tâche du suivi est découpée en trois blocs fonctionnels. L'extraction des amers dans l'image courante (FB1), l'estimation initiale et relocalisation (FB2), le suivi de la carte locale et la décision d'ajout d'une nouvelle image (FB3). La tâche de cartographie locale est découpée en cinq blocs fonctionnels. L'insertion d'une nouvelle image (FB4), la sélection des amers (FB5), la création des nouveaux amers (FB6), la compensation par faisceaux (FB7) et la suppression des images (FB8). La dernière tâche concernant la fermeture de boucle est découpée en deux blocs fonctionnels. La détection de fermeture de boucle (FB9) et la correction de boucle (FB10).

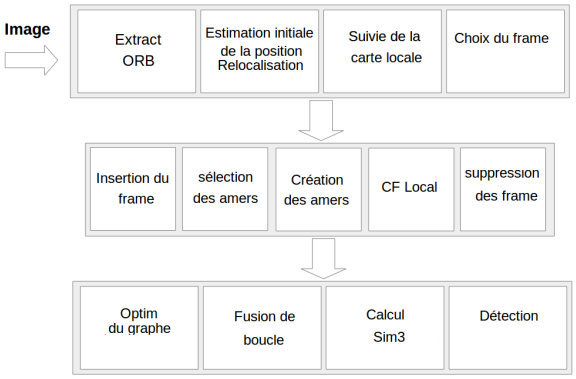


Figure 3.14: *Organigramme des blocs fonctionnels de l'ORB SLAM*

Définition des blocs fonctionnels du RatSLAM Le RatSLAM est découpé en trois blocs fonctionnels principaux exécutant la tâche de localisation et de cartographie. La vue locale est la mise en correspondance (FB1). Dans ce bloc, les cellules de vue locale représentent un tableau des unités avec une taille variable. Chaque unité représente une scène visuelle distincte de l'environnement. Si une nouvelle scène visuelle est observée, une nouvelle unité est crée contenant les coordonnées des pixels de la scène dans l'environnement. Si une scène visuelle est ré-observée par le robot, la cellule de vue locale correspondant est activée et injecte une activité dans la cellule de position. Dans le bloc (FB2), réseaux de cellules de position, cellules constituent un réseaux continu d'unités connectées par des connexions inhibitrices. Ces liaisons ressemblent à un réseau de neurones de navigation qui se trouve chez certains mammifères appelé une cellule de la grille. Les cellules sont reliées à des cellules voisines par des connexions excitateurs enveloppées dans toutes les extrémités du réseau. La dimension du tableau de cellules correspond à un robot mobile dont l'état est représenté par une position de trois dimension (x, y, θ) . La dynamique du réseau de cellules de position sont de telles sorte que l'état stable est un seul groupe d'unités activées, appelé un paquet d'activité ou un paquet d'énergie. Le barycentre de ce paquet représente la meilleur estimation de la position du robot. Les données odoométriques engendrent une activité dans les cellules de position pour représenter le mouvement du robot on se basant sur une échelle spatiale nominale pour chaque cellule de position. L'injection d'une activité par les cellules de vue locale fournit un mécanisme pour effectuer un processus de fermeture de boucle. Le dernier bloc (FB3), carte d'expérience, estime une position unique du robot tout en combinant les informations provenantes des cellules de position et des cellules de vue locale. Une nouvelle expérience est créée si l'état de l'activité courante du robot dans les cellules de position n'a pas été appariée avec un état associé à une expérience déjà vue. Quand le robot navigue entre les expériences, une liaison

est créée entre l'ancienne expérience active déjà observée et la nouvelle expérience. Le robot se base sur ces informations temporaires pour se localiser et cartographier un chemin à partir de sa position courante.

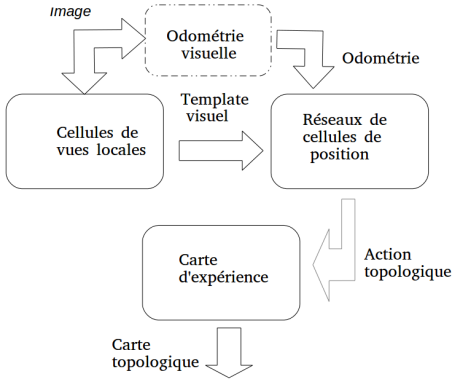


Figure 3.15: *Organigramme des blocs fonctionnels du RatSLAM*

Définition des blocs fonctionnels du SLAM Linéaire Le SLAM linéaire a été découpé en trois blocs fonctionnels exécutant la fusion des sous-cartes globales. Le bloc d'appariement et initialisation des amers (FB1), l'appariement des amers dans l'algorithme SLAM linéaire consiste à chercher l'amer dans la carte locale qui est déjà inclus dans la carte globale. Ceci est effectué comme suite : tous d'abord on détermine un ensemble potentiel des sous-cartes locales qui se chevauchent entre elles, une recherche des correspondances potentielles des amers est ensuite réalisée. Finalement, une mise en correspondance statistique est faite pour retrouver la correspondance (voir annexe 7.3, section 7.3.3.1). Les amers qui ne sont pas appariés sont considérés comme nouveaux amers, ils sont ensuite initialisés et ajoutés dans la carte globale (voir annexe 7.3, section 7.3.4.1). La mise à jours de la carte globale (FB2) : la carte locale fournit une estimation consistance de la position relative entre la position initiale de départ et la position finale. Cette carte est considérée comme une observation de la position réelle relative. Cette observation est utilisée pour mettre à jour le vecteur et la matrice d'information de la carte globale. Le vecteur d'état global est ensuite réorganisé pour rendre efficace le calcul de la factorisation de Cholskey (FB3). Si le vecteur d'état global n'a pas été réorganisé après la phase de mise à jour, l'ancienne factorisation de Cholesky est utilisée pour construire une nouvelle factorisation. Autrement, si le vecteur d'état global a été réorganisé après la mise à jour, une factorisation de Cholesky directe est effectuée pour construire la nouvelle factorisation. Une équation linéaire creuse est résolue en utilisant la factorisation de Cholesky pour récupérer le vecteur d'état global.

3.9 Choix des architectures et évaluation temporelle

3.9.1 Choix des architectures

L'implémentation d'un algorithme dépends très fortement de l'architecture cible. Chaque architecture dispose de spécificités propres qui permettent une implantation efficace. Dans notre étude, on propose d'étudier les implantations des algorithmes sur des architectures multi-processeurs et embarquables. Ces architectures sont aujourd'hui très répandues et largement distribuées en particulier dans les applications de drones ou des smartphones. Notre étude s'intéresse donc à plusieurs types

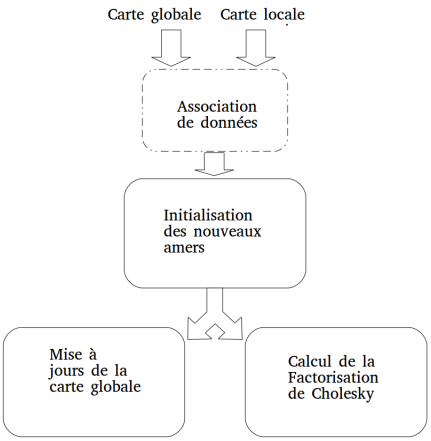


Figure 3.16: *Organigramme des blocs fonctionnels du SLAM Linéaire*

Table 3.1: *Architectures utilisées dans l'évaluation*

	Station de travail	PC Portable	Architectures pour l'embarqué				
			Tegra X1	Tegra K1	ODROID XU4	i.MX6 Sabre Lite	Pandaboard ES
CPU	Core 2 Quad Q6600	Dual-Core T4300	4x ARM Cortex A57 + 4x ARM Cortex A53	Quad-Core ARM Cortex-A15	Exynos 5422	Quad-Core ARM Cortex A9	Dual-Core ARM Cortex A9
CPU Cores	4	2	8	4+1	8	4	2
CPU GHz	2.40	2.10	1.9	2.3	1.8	1.2	1.2
Ubuntu OS	Precise, 12.04 LTS	Precise, 12.04 LTS	Linux de la Tegra 14.04	Linux de la Tegra 12.04	Precise, 12.04	Linaro, 13.04	Precise, 12.04 LTS

d'architectures afin de permettre une évaluation globale consistante. La première évaluation est faite sur quatre architectures embarquées récentes et sur deux machines performantes. Ces architectures sont décrites en détail dans le tableau 3.1.

Tegra K1 La Tegra K1 est un système sur une puce (*System on Chip SoC*) fabriqué par NVIDIA pour les appareils mobiles et les applications multimédia. Cette architecture est basée sur un processeur K1. Ce processeur dispose de quatre cœurs plus un cœurs ARM Cortex A15, cadencé à une fréquence de 2.3 Ghz. Le cœur Cortex-A15 est le processeur mobile le plus avancé dans le monde. IL contient un noyau de batterie innovant de troisième génération économiseur de l'énergie, pour offrir plus de performance . Le processeur K1 est basé principalement sur une architecture symétrique multi-traitement (SMP). Cette architecture permet l'utilisation des quatre cœurs ARM au maximum en cas de besoin. Chacun des quatre cœurs peuvent être utilisé indépendamment et automatiquement activé ou désactivé en fonction du type de traitement. Ce processeur est associé à 8 Go de mémoire DDR3L et LPDDR3.

Tegra X1 Tegra X1 est le dernier système sur une puce fabriqué par NVIDIA pour toutes les applications mobiles. C'est une architecture plus avancée que la TK1. Elle est destinée premièrement à accélérer les applications automobiles et les systèmes d'aide à la conduite. Cette architecture est basée sur le processeur X1. Ce processeur dispose de quatre cœurs ARM Cortex A57 et quatre cœurs ARM Cortex A53, cadencés à une fréquence de 1.9 GHz. Le processeur A57 est de 64 bits, de capacités

vidéo 4K incomparables, de performances importantes et d'un rendement énergétique sans précédent. C'est le compagnon idéal des applications mobiles les plus exigeantes. Ce processeur est associé à 8 Go de mémoire LPDDR4.

ODROID XU4 L'odroid XU4 est un système sur puce fabriqué par Hardkernel qui fait partie de la nouvelle génération des architectures embarquées dédiées pour les applications mobiles et multimédia. Cette architecture est basée sur le Samsung Exynos 5422. L'Exynos 5422 dispose de quatre cœurs ARM Cortex A15 cadencés à une fréquence de 2.0 GHz et quatre cœurs ARM Cortex A7 cadencés à une fréquence de 1.4 GHz. Ce processeur est dédié principalement pour le calcul hétérogène multi-traitement à faible consommation d'énergie. Ce processeur est associé à 2 Go de mémoire LPDDR3.

i.MX6 Sabre Le i.MX6 Sabre lite est un système sur puce de développement bas-coût fabriqué aux états unis qui cible principalement les applications mobiles sur des appareils à faible consommation. Cette architecture est basée sur le processeur i.MX6 qui dispose de quatre cœurs ARM Cortex A9 cadencés à 1.2 GHz. Ce processeur permet le développement rapide bas-coût des applications multimédia pour différents systèmes tel que l'Android et Linux. Ce processeur est associé à 1 Go de mémoire DDR3.

Pandaboard ES La pandaboard ES est une architecture conçue par Texas Instruments dédiée aux applications multimédias bas-coût. Cette architecture est basée sur un processeur OMAP4460 qui dispose de deux cœurs ARM Cortex A9, cadencés à une fréquence de 1.2 Ghz et disposant d'un coprocesseur vectoriel SIMD NEON. Le cœur Cortex-A9 fournit un excellent niveaux de performance. Ce processeur est une solution idéale pour les systèmes nécessitant des performances élevées et une consommation électrique faible. Aujourd'hui, les Cortex-A9 sont parmi les cœurs les plus performants fabriqués par ARM et disponibles pour le grand public. Ce processeur est associé à 1 Go de mémoire DDRAM.

Machines performantes Afin de permettre une évaluation consistante, nous utiliseront dans nos évaluations des machines performantes haute-gamme qui constitueront une référence pour évaluer les performances des implémentations sur les architectures embarquées. Pour cela, nous considérons une station de travail qui intègre un processeur hôte de quatre cœurs cadencés à 2.45 GHz. Afin d'être aussi représentatif que possible sur les plate-formes d'évaluation, nous incluons un ordinateur portable haute-de-gamme qui intègre un processeur hôte de deux cœurs cadencés à 2.10 GHz.

3.9.2 Évaluation des temps d'exécution

Les algorithmes SLAM sont développés principalement pour des tâches d'exploration des environnements réels. Afin de rester aussi représentatif que possible, nous utiliseront un jeu de donnée réelles comme source de données capteurs pour les algorithmes SLAM. Pour ce faire, nous utiliseront les données expérimentales décrites dans le chapitre 2. Notre évaluation s'intéresse aussi au SLAM en tant que système. Pour cela, nous adoptons une évaluation par la méthode HIL décrite dans le chapitre 2, section 2.3.3. La figure 3.17 décrit la méthode d'évaluation des quatre algorithmes SLAM sur les différentes architectures. Le système embarqué représente les différentes architectures décrites auparavant (Tegra X1, K1, XU4, i.MX6 ou Panda ES). La machine hôte est utilisée pour fournir tous d'abord les données capteurs (flux laser, images et encodeur) selon le type de l'algorithme à tester et pour la visualisation des résultats de localisation et analyser les performances et les temps d'exécution. Nous utiliserons différentes liaison pour transmettre les données capteurs (des messages ROS,

liaison série ou Ethernet) et aussi pour récupérer les résultats. Pour une évaluation cohérente, tous les algorithmes on été exécutés pour 100 itérations et le temps moyen d'exécution est ensuite calculé.

Figure 3.17: *Méthode d'évaluation des algorithmes de SLAM sur les architectures embarquées*

3.9.2.1 Implémentation mono-cœur du FastSLAM2.0

Le Tableau 3.2 montre les résultats de l'implémentation mono-cœur de l'algorithme FastSLAM2.0 sur les six architectures. Le temps d'exécution global sur la station de travail est de 35.45 ms, 60.71 ms sur le PC portable. Sur les architectures embarquées, le temps d'exécution global est de 63.88, 72.28, 91.4 et 130.26 ms respectivement sur TX1, XU4, iMX6 et Panda ES. Les résultats obtenus sur les architectures embarquées peuvent donner une première impression que le FastSLAM2.0 peut tourner en temps réel sur les architectures embarquées bas-coût. Ceci est vrai si et seulement si l'algorithme est exécuté sur une trajectoire courte (100 itérations) où 100 particules sont suffisantes pour obtenir des résultats consistants [7]. Cependant, il reste des défis importants à confronter pour que le FastSALM2.0 monoculaire puisse cartographier un environnement large. Le nombre de particules dans un tel système est nécessaire pour maintenir une estimation raisonnable de position et d'incertitude des amers comme cité dans [127]. Pour explorer et cartographier un environnement large, une estimation consistante de l'incertitude est exigée. Par conséquence, le nombre de particules doit être augmenté en fonction de la complexité de l'environnement. Un tel algorithme qui fonctionne avec un grand nombre de particules va sans doute augmenter le temps d'exécution puisque tous les blocs fonctionnels dépendent principalement du nombre de particules. Dans ce cas, une implémentation naïve de l'algorithme sur un système embarqué n'est pas efficace, des optimisations sont alors nécessaires pour répondre aux contraintes du temps réel.

FBs	Station de travail	PC portable	Architectures pour l'embarqué			
			TX1	XU4	i.MX6 Sabre	Panda-board ES
Prédiction	0.101	0.212	5.32	4.22	0.30	0.46
Détection /Appariement	2.39	5.01	0.313	0.33	4.24	6.49
Mise à jour de la position	8.83	18.49	19.50	23.56	28.61	43.5
Estimation	22.56	34.59	35.53	40.34	55.03	75.4
Initialisation	1.53	2.35	3.16	3.22	3.17	4.35
Rééchantillonnage	0.04	0.06	0.052	0.61	0.05	0.065
Total (ms)	35.45	60.71	63.88	72.28	91.4	130.26

Table 3.2: *Temps d'exécution du FastSLAM2.0 (avec 100 particles sur environnement réel)*

3.9.2.2 Implémentation mono-cœur de l'ORB SLAM

Le Tableau 3.3 montre les résultats de l'implémentation de l'algorithme ORB SLAM sur les six différentes architectures. Le temps d'exécution global sur la station de travail est de 346.76 ms, 727.76 ms sur le PC portable. Sur les architectures embarquées, le meilleur temps d'exécution est de 880.27 sur le TX1. Comme vu auparavant, l'ORB SLAM prend en entrée des images pour cartographier l'environnement et construire le trajet du robot. Un tel système doit traiter les images dès leur acquisition pour fournir des résultats stables et consistants en termes de de localisation. Certains blocs

fonctionnels nécessitent un temps d'exécution considérable comme la compensation par faisceaux et la création des amers. Dans ce cas, l'ORB SLAM doit être exécuté au maximum dans 346.76 ms pour pourvoir traiter les images en temps réel. Cet algorithme doit être exécuté sur des machines performantes de haute gamme pour assurer l'aspect temps réel [2]. L'ORB SLAM est exécuté dans environs une ou deux secondes sur les architectures embarquées. Ce qui signifie que le système est incapable de traiter les images en temps réel. Les données expérimentales utilisées dans l'évaluation fournissent 30 images par second (30 fps), ce qui signifie que certaines images ne seront pas traitées. Ceci provoque une perte d'information et donc le système fournit des résultats instables et inconsistants. Dans ce cas, des optimisations s'avèrent donc indispensables, mais cela exige de revoir la structure algorithmique car l'ORB SLAM n'est pas parallèle par nature. Certains blocs fonctionnels sont séquentiels exigeant une transformation algorithmique pour pouvoir l'implémenter sur une architecture embarquée.

FBs	Station de travail	PC portable	Architectures pour l'embarqué			
			TX1	XU4	i.MX6 Sabre Lite	Panda-board ES
Extraction ORB	12.539	19.28	123.16	190.94	224.64	343.25
Estimation de la position initiale	5.57	11.17	33.58	42.16	61.26	93.61
suivi de la carte locale / décision d'ajout d'une image	8.97	19.64	17.60	20.43	32.11	48.81
Insertion d'une nouvelle image	12.38	22.30	33.37	36.45	60.88	92.55
Sélection des amers / Création des amers	96	214.53	251	284.68	292.04	400.24
Compensation par faisceaux	200.12	415.75	368.44	521.41	672.82	921.77
Suppression des amers	4.53	7.92	4.44	5.32	8.11	10.06
Détection de boucle / correction de boucle	6.66	17.84	18.68	20.64	22.34	29.28
Total (ms)	346.76	727.76	880.27	1122.03	1374.2	1939.57

Table 3.3: *Temps d'exécution de l'ORB SLAM sur environnement réel*

3.9.2.3 Implémentation mono-cœur du RatSLAM

La complexité de l'algorithme RatSLAM augmente linéairement avec le nombre des templates visuels et les nœuds de la carte d'expérience. En effet, les cartes d'expériences sont liées de façon creuse, ce qui augmente linéairement le temps d'exécution. Les performances temps réel du RatSLAM dépendent de trois facteurs. Le premier facteur est la phase de l'appariement : la taille du template visuel détecté augmente le temps de calcul de la similarité entre le template visuel courant et celui détecté auparavant. Deuxièmement, dans le bloc fonctionnel e cellule de position, la complexité de calcul augmente linéairement avec la taille des cellules de position dans l'environnement. Finalement, dans le dernier bloc fonctionnel, le calcul est aussi linéairement dépendant du nombre de boucles à corriger dans chaque itération. Le Tableau 3.4 montre les résultats de l'implémentation de l'algorithme RatSLAM sur les six architectures. Le temps d'exécution global sur la station de travail est de 119.06 ms, 170.75 ms sur le PC portable. Sur les architectures embarquées, le temps d'exécution global est de 270, 312.3, 356.32 et 520.73 ms respectivement sur TX1, XU4, iMX6 et la panda-board ES. Le RatSLAM ne peut pas tourner en temps réel sur les architectures embarquées comme il ne peut pas traiter les images en temps réel. Des optimisations sont alors indispensables. Cependant, dans les approches bio-inspirées, certaines optimisations peuvent augmenter la complexité du code en particulier si des transformations algorithmiques sont nécessaires.

FBs	Station de travail	PC portable	Architectures pour l'embarqué			
			TX1	XU4	i.MX6 Sabre	Panda-board ES
Vu locale et appariement	97.80	136.92	140.05	166.3	179.01	279.26
Réseaux de cellules de Position	18.87	30.16	120.8	135.04	166.10	227.56
Carte d'expérience	2.39	3.67	9.35	10.96	11.21	13.91
Total (ms)	119.06	170.75	270.2	312.3	356.32	520.73

Table 3.4: *Temps d'exécution du RatSLAM sur environnement réel*

3.9.2.4 Implémentation mono-cœur du SLAM Linéaire

Le SLAM linéaire résout le problème du full SLAM. La carte globale et la trajectoire sont reconstruite à partir d'un certain nombre de sous-cartes locales. Les cartes locales peuvent être construites en utilisant différentes techniques de SLAM. Dans notre évaluation, nous avons utilisé l'EKF pour construire les cartes locales on se basant sur les données du flux Laser et des encodeurs fournie par le jeux de données réelles. Le tableau 3.5 montre les résultats de l'implémentation du SLAM Linéaire sur les différentes architectures. Le temps total donné dans le tableau 3.5 est le temps de traitement global pour construire la carte globale après 100 itérations de l'algorithme. Comme vu auparavant, le SLAM linéaire est une solution du problème du full SLAM. Il consiste à joindre un certain nombre des sous-cartes locales en utilisant deux méthodes différentes telle que la méthode séquentielle (SEQ) et "Diviser pour régner" (DC). Dans notre évaluation, nous donnons les temps d'exécution du SLAM linéaire par les deux méthodes. Ainsi, les temps 23.2, 38.32, 42.34 et 58.21 secondes sont les temps globaux sur les architectures embarquées, TX1, XU4, iMX6 et la panda-board ES, pour construire la carte globale en utilisant la méthode séquentielle (SEQ). Par conséquence, une telle méthode prend énormément de temps et donc n'est pas convenable pour une application temps réel sur un système embarqué. En utilisant la méthode diviser pour régner, le temps d'exécution globale est de 1.514, 2.25, 2.77, 4.24 s respectivement sur la TX1, XU4, i.MX6 et panda ES. Sur la station de travail et le PC portable le temps global est de 0.74 et 1.12 s. Le temps nécessaire pour construire une carte locale dépend de la complexité de la méthode utilisée. En utilisant l'EKF, cette reconstruction prend environ $1s$ sur les architectures embarquées. Pourtant, le SLAM linéaire n'a pas été encore utilisé dans un SLAM actif. Par conséquent, l'algorithme est à réécrire pour permettre des optimisations matérielles et assurer des performances en temps réel et réduire la complexité de calcul.

FBs	Station de travail		PC portable		Architectures pour l'embarqué							
					TX1		XU4		i.MX6		Panda-bord ES	
	DC	SEQ	DC	SEQ	DC	SEQ	DC	SEQ	DC	SEQ	DC	SEQ
carte locale	0.42	0.42	0.63	0.63	0.56	0.56	0.72	0.72	1.03	1.03	1.58	1.58
SLAM Linèaire	0.32	5.83	0.49	8.93	0.954	22.64	1.53	37.6	1.74	41.31	2.66	56.63
Total (s)	0.74	6.25	1.12	9.56	1.514	23.2	2.25	38.32	2.77	42.34	4.24	58.21

Table 3.5: *Temps d'ecxécution du SLAM Linéaire (100 iterations) sur environnement réel*

3.10 Comparaison des performances

Les figures 3.18, 3.19, 3.20 et 3.21 montrent les évaluations des bloc fonctionnels respectivement du FastSLAM2.0, ORB SLAM, RatSLAM et le SLAM linéaire sur les différentes architectures.

FastSLAM2.0 Les deux blocs FB4 (estimation) et FB3 (mise à jour de la position des particules) sont des blocs conséquents dans l'algorithme FastSLAM2.0. Ils dépendent du nombre des amers à traiter qui augmente le nombre des itérations des blocs. Cependant, ces résultats ont été obtenus avec 100 particules sur une petite partie de la trajectoire. Pour des environnements larges, des modifications algorithmiques sont nécessaires pour adapter l'algorithme avec les contraintes imposées par l'environnement exploré. Ceci modifiera le pourcentage du temps de calcul par bloc, plus particulièrement la phase de prédiction (FB1). Le bloc de traitement d'image (FB2) utilise le détecteur FAST qui n'est pas significatif en temps de calcul. De plus ce bloc ne dépend pas du nombre de particules car la phase d'appariement est faite par rapport à la particule qui a le plus grand poids. Le traitement par amer dans ce bloc peut être facilement parallélisable sur une architecture multi-cœurs. Les autres blocs fonctionnels de l'algorithme dépendent du nombre de particules traitées, le calcul est indépendant pour chacune des particules. Ce qui rend l'algorithme valable aux optimisations parallèles sur des architectures embarquées.

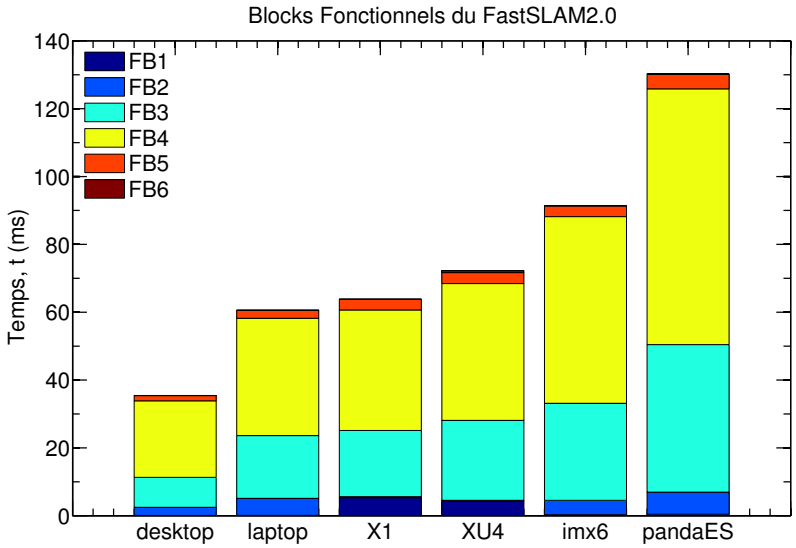


Figure 3.18: *Temps d'exécution des blocs fonctionnels du FastSLAM2.0*

ORB SLAM Le bloc FB6 (compensation par faisceaux) est le bloc le plus occupant de l'algorithme. Il consomme un temps considérable, environ 200 ms sur une machine aussi performante et jusqu'à une seconde par itération sur une architecture embarquée. Ce temps peut augmenter d'avantage pour des environnements plus complexes. Des optimisations parallèles du bloc FB6 ne sont pas évidentes. Dans le domaine de la vision par ordinateur, plusieurs travaux ont réussi à accélérer certains algorithmes de vision sur des architectures parallèles : la détection et l'appariement des amers, la construction par stéréo-vision et les systèmes SFM (structure à partir du mouvement) [128]. Cependant, la phase de compensation par faisceaux s'exécute toujours sur un CPU mono-cœur. Des travaux ont été menés pour une accélération logicielle du bloc FB6 sur des architectures parallèles tel que le GPU. [129] a proposé une implémentation hybride du bloc FB6 dans laquelle le calcul est partagé entre le CPU et le GPU. La matrice Hessienne et le complément de Schur sont construits sur le GPU. Malheureusement,

ceci n'est pas pratique pour des environnements aussi large où il y a beaucoup des images à traiter en temps réel, spécialement les images qui ont un complément de Schur plus dense. Même une machine performante intégrant un GPU haute-gamme contient peu de mémoire RAM qui ne lui permet pas de sauvegarder la matrice Hessienne où le complément de Shur. Même si on dispose d'une RAM suffisante, la construction du complément de Schur reste très coûteuse. Par conséquence, ces optimisations ne sont pas faisable sur un système embarqué intégrant un GPU avec une mémoire insuffisante. Certains blocs fonctionnels de l'ORB SLAM peuvent avoir des optimisations parallèles plus particulièrement les tâches de traitement par amers (le calcul de l'orientation et le descripteur ORB de toutes les observation détectées (FB1), l'appariement entre les amers de la carte et les observations détectées (FB1, FB2, FB3) et la triangulation des nouveaux amers (FB5)). Cependant, ces blocs fonctionnels ne sont pas gourmands en temps de calcul par rapport à la compensation par faisceaux (FB6). Selon la loi d'Amdahl, un bloc fonctionnel qui occupe un pourcentage minimal du temps d'exécution global, ne peut contribuer à aucune accélération globale significative, même si l'accélération de ce bloc est 1000 fois plus rapide. Par conséquence, l'ORB SLAM ne peut pas bénéficier complètement des optimisations sur des architectures parallèles embarquées.

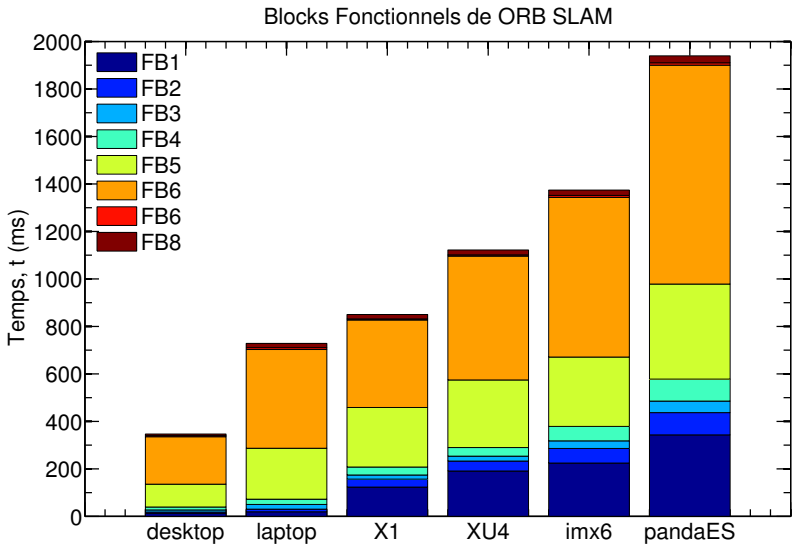


Figure 3.19: *Temps d'exécution des blocs fonctionnels de l'ORB SLAM*

RatSLAM La figure 3.20 montre la distribution du pourcentage du temps de calcul des blocs fonctionnels du RatSLAM. Le bloc de vue locale et d'appariement sont gourmands en temps de calcul par rapport aux blocs FB2 et FB3. Le bloc FB1 est exécuté à chaque acquisition d'image, il implémente des algorithmes de traitement d'images coûteux en temps de calcul pour transformer l'image en des templates visuels. La recherche des correspondances dans ce bloc est aussi coûteuse en temps de calcul et la complexité augmente avec le nombre des templates visuels détectés. Le calcul de la distance SAD est fait pour tous les templates visuels courants avec ceux déjà vus dans la carte. Une optimisation parallèle de ce bloc est possible en parallélisant le calcul pour chaque template visuel. Le RatSLAM ne dispose pas d'un modèle de mouvement explicite, il repose principalement sur des activités injectées à partir du bloc FB1 dans le cas où il y a des correspondances entre les templates visuels. L'injection des activités dans le bloc FB2 augmente la taille des cellules de position. Vu que la représentation des cellules de position est dense, ceci augmente la complexité de calcul. La complexité de calcul de ce bloc ne peut être réduite qu'avec une représentation creuse des cellules de position, mais ceci augmentera d'avantage la complexité du code. Le bloc carte d'expérience (FB3) exploite les actions reçues pour

créer de nouveaux nœuds et liaisons. Ce bloc consomme peu de temps par rapport au temps global du RatSLAM.

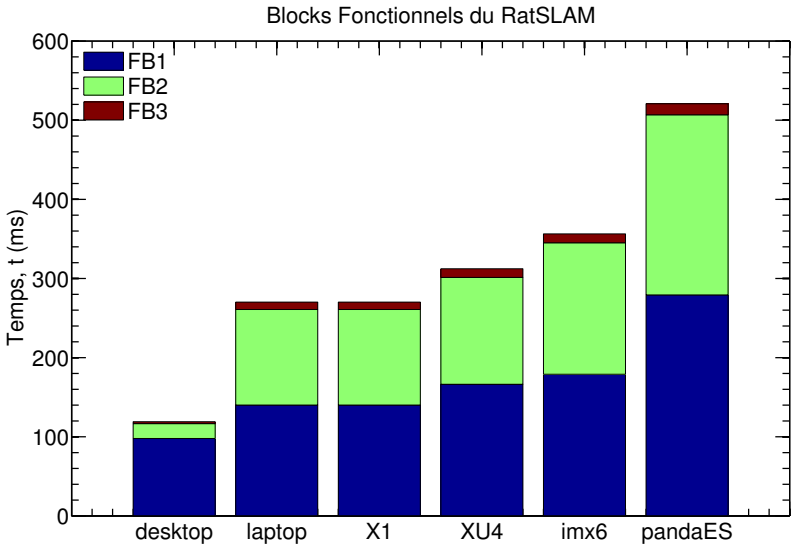


Figure 3.20: *Temps d'exécution des blocs fonctionnels du RatSLAM*

SLAM linéaire La figure 3.21 montre les temps d'exécution des blocs fonctionnels du SLAM linéaire par la méthode "Diviser et régner". La construction des sous-cartes locales par l'EKF SLAM (FB1) et la construction de la carte globale (FB3) sont des tâches gourmandes en temps de calcul. La carte globale d'une partie de la trajectoire ne peut être construite (FB3) qu'à partir d'un certain nombre des sous-cartes locales construites par l'EKF (FB1). Au fur et à mesure que le robot avance dans l'environnement, le nombre des amers augmente, ce qui augmente la taille de la matrice de covariance et le temps de traitement du bloc FB2. Le temps de traitement du bloc FB3 augmente d'avantage avec le nombre des sous-cartes locales intermédiaires et avec le nombre des amers estimés par l'EKF dans chacune des sous-cartes. La complexité du bloc FB4 (la factorisation de Cholesky) dépend du nombre des éléments introduits dans la matrice qui dépend à son tour de l'environnement et de la trajectoire du robot. Ce qui influence et augmente le temps d'exécution de la factorisation de Cholesky et le calcul des équations linéaires. La complexité des deux blocs FB1 et FB3 ne peuvent être réduite qu'avec la méthode "Diviser et régner" implémentée dans notre évaluation [130]. Cependant, cette méthode reste coûteuse en terme de calcul. Elle est basée principalement sur l'EKF, ce dernier n'est pas facilement parallélisable, sa nature n'est pas adéquate avec une architecture embarquée massivement parallèle.

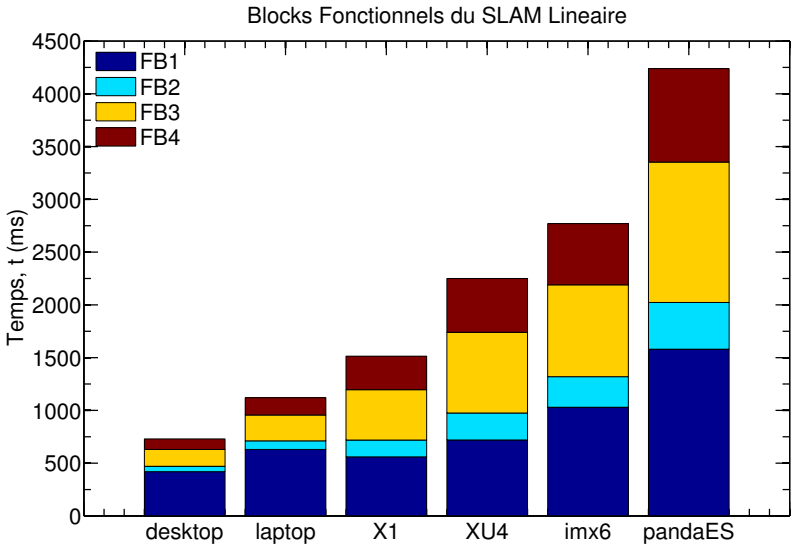


Figure 3.21: *Temps d'exécution des blocs fonctionnels du SLAM linéaire*

Les performances absolues d'un SLAM embarqué dépendent principalement de la nature de l'algorithme lui même, mais aussi de l'architecture du système embarqué cible. Dans ce cas, des optimisations logicielles et matérielles sont alors indispensables pour permettre une exécution temps réel des ces algorithmes sur un système embarqué. Notre méthodologie, assume que la conception d'un système embarqué dédié aux applications SLAM est basée sur plusieurs étapes. On réalise d'abord une étude des algorithmes SLAM répandus et qui peuvent résoudre la problématique du SLAM ainsi que les architectures embarquées capables de porter ces algorithmes. Ensuite, il faut faire un choix adéquat d'un algorithme et une architecture qui peuvent constituer ensemble un système performant en termes de résultats de localisation. Dans ce contexte, la figure 3.22 montre le temps d'exécution global des algorithmes étudiés sur les architectures embarquées choisies. L'algorithme FastSLAM2.0 semble être un bon candidat pour un système SLAM embarqué temps réel. En outre, le FastSLAM2.0 monoculaire est un algorithme qui est parallélisable par nature. Ces raisons en font un bon candidat pour des optimisations parallèles sur des architectures embarquées.

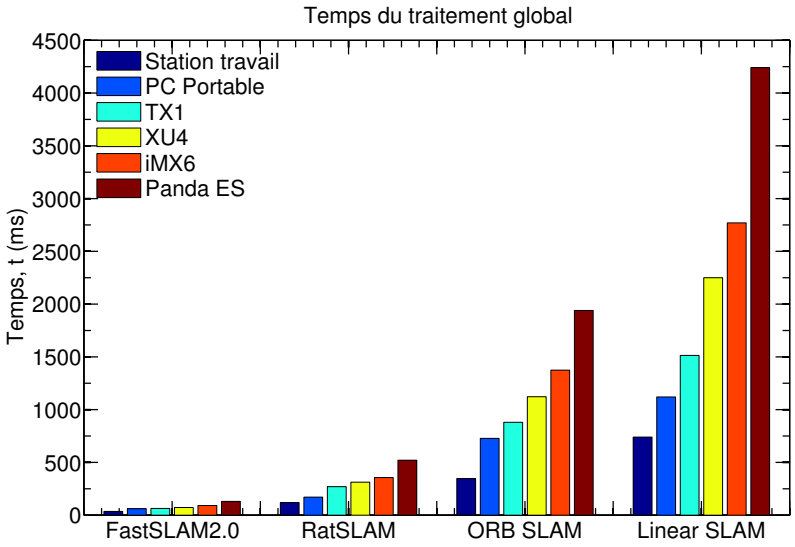


Figure 3.22: *Temps d'exécution globaux des algorithmes SLAM sur les différentes architectures*

3.11 Bilan

Dans ce chapitre nous avons réalisé une étude algorithmique de quatre algorithmes SLAM de différentes natures.

Le FastSLAM2.0 monoculaire est une approche probabiliste. Il est basé sur le filtre particulaire. L'incertitude du robot est modélisée par un ensemble de particules dont chacune dispose de sa propre carte de l'environnement. Un tel algorithme est parallèle dans sa nature de traitement, et implique un traitement d'image au niveau de l'extraction et de l'appariement des amers. Cet algorithme nécessite des ressources de calcul importante à cause du traitement parallèle de plusieurs particules. Par conséquence, il s'avère très intéressant d'étudier son embarquabilité sur une architecture embarquée. S'il s'avère qu'il est susceptible d'être portable, il sera sans aucun doute souvent utilisé dans plusieurs applications de la robotique mobile et la navigation autonome.

L'ORB SLAM est un algorithme basé sur l'approche SfM. Cet algorithme repose sur la vision monoculaire pour cartographier des grands ou petits environnements. L'algorithme utilise des images pour la perception de l'environnement et l'odométrie visuelle pour fournir une estimation correcte de la position du robot dans l'environnement. Il embarque une base de donnée pour la reconnaissance de place afin de réaliser une fermeture de boucle. En outre, la nature de cet algorithme en fait un candidat qui mérite d'être étudié, puisqu'il implique le traitement d'image lourd qui nécessite des ressources de calcul importantes.

Le RatSLAM est un algorithme basé sur l'approche probabiliste. L'algorithme fournit une carte topologique de l'environnement. L'algorithme est basé principalement sur le modèle de calcul des réseaux de neurones. Cet approche utilise une combinaisons des apparences de la scène, des réseaux compétitifs et une représentation topologique de la carte de l'environnement. En particulier, le RatSLAM est capable de donner des résultats consistants même avec des capteurs de faible résolution. L'étude de cet algorithme est intéressante afin d'évaluer la portabilité des approches bio-inspirée sur des architectures embarquées.

Finalement, le SLAM linéaire, est une approche probabiliste capable d'adresser le problème du full SLAM. Ce dernier est traité à travers la résolution d'un système de moindre carré. Il est utilisé principalement pour des environnements larges et basé sur le principe de fusion d'un ensemble de sous-cartes locales. Il utilise une carte globale pour reconstruire le trajet complet du robot. Les sous-cartes locales sont construites en utilisant des techniques SLAM existantes. Ceci en fait une autre étude de cas intéressante pour évaluer la portabilité des algorithmes type full SLAM sur des architecture bas-coût.

Nous avons évalué de manière quantitative les temps d'exécution des quatre algorithmes SLAM sur plusieurs architectures embarquées . Des comparaisons ont été faites avec des implémentations sur des machines performantes haute gamme. La plupart des implémentations de SLAM dans l'état de l'art ont utilisées des machines puissantes. Ces algorithmes, dans leur états actuels, ne sont pas encore prêts à être implémentés sur des architectures embarquées.. Dans ce cas, une méthodologie d'adéquation algorithme architecture est obligatoire afin d'aboutir à des performances qui répondent aux exigences du temps réel pour les applications SLAM embarquées.

La puissance du calcul d'un système embarqué augmente grâce aux architectures multi-cœurs et hétérogènes. Ces architectures peuvent être exploitées afin d'améliorer de plus l'exécution temps réel du calcul embarqué pour le SLAM qui n'était pas possible il y a quelque années. Nous avons vu que l'algorithme FastSLAM2.0 est un candidat qui satisfait le compromis entre la consistance des résultats de localisation et les temps d'exécution. La nature des traitement de données lui permet de bénéficier de plus de certaines spécificités des architectures embarquées. Dans le chapitre suivant nous allons

explorer les optimisations du FastSLAM2.0 sur une architecture massivement parallèle à base de GPU. Nous explorerons aussi la conception d'une architecture programmable à base de FPGA. L'étude sera poussée par des comparaisons de performances entre les différentes architectures GPU et FPGA afin de converger vers une solution adéquate permettant d'embarquer un algorithme SLAM à large échelle.

Chapitre 4

FastSLAM2.0 : Vers une implantation embarquée

Sommaire

4.1 Introduction

Dans le chapitre précédent, nous avons défini et évalué quatre algorithmes de SLAM sur plusieurs type d'architectures grand public. Après avoir évalué le temps de calcul des différents blocs fonctionnels constituant chacun de ces algorithmes, il est apparu que le FastSLAM2.0 représente un bon candidat pour bénéficier des optimisations sur des architectures parallèles. De plus, le FastSLAM2.0 résout le problème de corrélation en utilisant un filtre particulaire pour estimer la position du mobile. Il décorrèle l'estimation de la position du robot et celle de chaque amer, ce qui limite les dépendances du temps de traitement à la taille de l'environnement exploré. En outre, nous avons vu précédemment que l'algorithme tel qu'il a été développée n'est pas apte à opérer en temps réel, en particulier pour des environnements large et complexe. Nous avons constaté qu'une optimisation de l'implantation de l'algorithme permettait d'obtenir des résultats de temps de traitement intéressants. Cette optimisation est absolument nécessaire pour obtenir un système efficace, opérant en temps réel. En outre, nous avons vu que l'évolution des architectures des systèmes embarqués permettait de réaliser ces optimisations grâce à l'évolution des systèmes à architectures hétérogènes, multi-cœurs et massivement parallèles.

Dans ce chapitre, nous effectuerons tout d'abord une étude algorithmique du FastSLAM2.0 en analysant ses contraintes et paramètres de dépendance. Ensuite, nous effectuerons une première optimisation parallèle sur une architecture homogène multi-cœurs. Nous présenterons après les modifications algorithmiques nécessaires pour adapter la version monoculaire du FastSLAM2 aux environnements larges. Nous définirons ensuite deux modèles d'optimisation sur des architectures hétérogènes massivement parallèles. Le premier modèle explore les optimisations logicielles sur les architectures de calcul graphique tandis que le deuxième modèle explore les optimisations matérielles sur une architecture programmable dédiée. Ces deux modèles seront définis en se basant sur une démarche d'adéquation algorithme architecture.

4.2 Définition du problème

Nous avons défini dans le chapitre 2 notre méthodologie d'évaluation d'un algorithme SLAM. Nous avons découpé l'algorithme FastSLAM2.0 en blocs fonctionnels exécutant des tâches de calcul bien spécifiques. Dans le chapitre 3, nous avons évalué les temps d'exécution des blocs fonctionnels du FastSLAM2.0 sur différentes architectures pour identifier les tâches qui demandent un temps de calcul important, mais ceci sans tenir en compte les paramètres de dépendances. Dans ce chapitre, nous analysons les temps d'exécution de chaque bloc fonctionnel pour déterminer précisément son temps de calcul en fonction de tous les paramètres de dépendance qui déterminent le nombre d'occurrences moyen lors d'une expérimentation. Les blocs fonctionnels nécessitant des ressources importantes sont ensuite optimisés pour réduire le temps de traitement global. Ces optimisations sont réalisées en adéquation avec l'architecture sur laquelle est implanté l'algorithme. En pratique, un algorithme SLAM est beaucoup plus complexe à étudier plus particulièrement quand il s'agit de fonctionner dans des environnements réels. Cela, nécessite parfois des modifications algorithmiques pour s'adapter au mieux aux caractéristiques de l'environnement tout en respectant les contraintes de l'architecture cible. Nous allons étudier chaque étape de l'algorithme FastSLAM2.0 et définir les dépendances aux différents paramètres. Certains paramètres pourront être fixés pour rendre constant ou borner le temps de traitement de l'étape.

Rappelons que la structure du FastSLAM2.0 est donnée par (4.1). L'ensemble S_t contient la position de chaque particule s_t^m, les paramètres de l'amer X_n^m, C_n^m et N^m est le nombre d'amer dans la carte des particules. Notons que ce nombre est constant pour chaque particule, vu que la phase

d'appariement est réalisée en tenant en compte la particule ayant le poids le plus grand. Contrairement au cas ou l'appariement est fait pour chaque particule qui engendre un nombre différent d'amer par particule. Par contre, ce type d'implémentation n'est pas envisageable vu la complexité de calcule.

$$S_t = \{s_t^m, N^m, X_1^m, C_1^m, \ldots, N^m, X_n^m, C_n^m\} \tag{4.1}$$

L'algorithme 1 résume l'algorithme FastSLAM2.0 globale. L'algorithme décrit le cas d'intégration d'une seule observation z_t par contrôle u_t. Le choix est fait pour une raison de simplification, dans le cas de plusieurs observations, le traitement est fait de manière séquentielle. L'algorithme 2 décrit la partie pré-traitement d'image. Les amers sont détectés par FAST puis appariés en calculant la distance ZMSSD.

Algorithme 1 : Algorithme FastSLAM 2.0 globale

```
while 1 do
    Pré-traitement des données extéroceptif                ▷ voir algorithmes 2;
    u_t ← (δ_s, δ_θ);
    Prédiction;
    for m = 1 to M do
        s_t^m = f(s_{t-1}^m, u_t) ;
    end
    Mise à jour de la position des particules;
    for m = 1 to M do
        for n = 1 to N do
            if X_n est apparié then
                mise à jours de la nouvelle distribution (μ_n^m, Σ_n^m);
            end
        end
        s_t^m ~ N(μ_n^m, Σ_n^m);
    end
    Estimation;
    for m = 1 to M do
        for n = 1 to N do
            if X_n est apparié then
                (X, C)_n^m ← Kalman(X_n^m, C_n^m, ẑ_n, z_n);
                calcule du poids ω^m;
            end
        end
    end
    Initialisation;
    for m = 1 to M do
        Calculer les paramétres initials de l'amer
    end
    Ré-échantillonnage;
    Ré-échantillonnage d'importance;
end
```

Algorithme 2 : Traitement d'image

Traitement d'image;
if $z_t \leftarrow$ *Image* **then**
  $(u_k, v_k) \leftarrow$ Détecteur FAST;
  sélectionner la particule s_t^x ayant le poids le plus grand;
  $\hat{z_n}\,(\hat{u_n}, \hat{v_n}) \leftarrow h_c\,(s_t^x, X_n^x)$;
  if $(\hat{u_n}, \hat{v_n}) \in$ *Camera Frame* **then**
    Sélectionner une observation z_n^k dont la distance $ZMSSD$ est minimale;
  end
end

4.3 Analyses des dépendances de l'algorithme

Bloc de prédiction (BF1) L'étape de prédiction met à jour la position du robot mobile (s_t) en fonction des données proprioceptives acquises à partir des odomètres (n_l, n_r). Le temps de traitement de ce processus n'est pas constant. Il met à jour le vecteur 3D contenant la position du robot et sa matrice de covariance. En effet, le temps d'exécution de ce bloc varie en fonction de plusieurs paramètres. Tout d'abord la nature de l'environnement dans lequel l'algorithme est censé opérer. Ceci détermine le nombre des particules qu'il faut utiliser en fonction de la complexité de l'environnement. Le temps d'exécution dépend du nombre des données odométriques acquises dans chaque itération de l'algorithme. Le nombre d'occurrence moyen par itération (MOI) de ce bloc est déterminé par rapport au nombre d'acquisition des données odométriques. (plus de détail est donné en Annexe-7.4, section 7.4.1).

Bloc de traitement d'image (BF2) Cette étape calcule la position d'un amer dans l'image à l'aide d'une mesure de corrélation. Chaque amer, présent dans la carte de la particule la plus probable, est projeté sur l'image en utilisant le modèle Pinhole. Puis, l'algorithme lui cherche une correspondance en utilisant la mesure de corrélation ZMSSD. Le temps de calcul des projections ne dépend que du nombre d'amers présents dans la carte. Cependant, le temps de recherche de correspondance ne peut être borné, il dépend du nombre de pixels candidats qui est défini par l'incertitude de localisation du robot et des amers.
Le temps de traitement de ce bloc dépend de plusieurs paramètres :

- Le nombre d'amers présents dans la carte
- Le nombre d'observation détectées dans l'image
- La taille du descripteur utilisé pour identifier l'amer
- L'incertitude de localisation des particules et de chaque amer

Le nombre d'amers n'est pas limité : plus le robot parcourt un trajet long, plus le nombre d'amers augmente. De plus, il n'est pas possible de borner facilement les incertitudes de localisation du robot ou des amers. Cependant, dans certains cas et particulièrement avec un grand nombre de particule, nous limitons le nombre d'amer potentiel pour respecter les contraintes de l'architecture embarquée cible.

Bloc de mise à jour de la position des particules (BF3) L'étape de mise à jour de la position des particules met à jour la localisation de chaque particule en utilisant les observations issues de l'étape de traitement d'image. Elle utilise des équations semblables aux équations de Kalman pour

mettre la position des particules ainsi que leurs incertitudes. L'intérêt majeur de ce bloc est d'éviter le problème de l'appauvrissement des particules et garde une richesses en nombre de particules pour des résultats plus consistants. Le temps de traitement de ce bloc est important et dépend des paramètres suivants :

— Le nombre d'amers appariés
— Le nombre de particules

Le nombre d'occurrences moyen de ce bloc est déterminé en fonction des amers appariés. La procédure de construction de la nouvelle distribution proposée est décrite dans l'annexe 7.4, section 7.4.2.

Bloc d'estimation (BF4) L'étape d'estimation met à jour la localisation de chaque amer pour chaque particule en utilisant le résultat des observations issues de l'étape de mise en correspondance. Elle utilise les équations de Kalman pour mettre à jour la position des amers ainsi que leurs incertitudes. L'intérêt majeur du FastSLAM2.0, par rapport à l'EKF-SLAM, est que le temps de calcul de cette étape dépend du nombre d'amers dans la carte. En effet, l'étape d'estimation consiste à réaliser uniquement des opérations matricielles de taille 3×3 alors que pour l'EKF-SLAM, la taille des matrices augmente en fonction du nombre d'amers présents dans la carte. Le temps de traitement de cette tâche est significatif et dépend des paramètres suivants :

— Le nombre d'amers dans la carte
— Le nombre d'amers observés
— Le nombre de particules

Ces différents paramètres pourront être bornés en limitant le nombre d'observations à chaque étape ainsi que le nombre d'amers présents dans la carte. La gestion de la carte d'amers est détaillée au paragraphe 3.3.6 dans le chapitre 3. Le nombre d'occurrences moyen de ce bloc est déterminé en fonction des amers de la carte et des observations. Le formalisme mathématique de l'estimation est détaillé dans l'annexe 7.4, section 7.4.3.

Bloc d'initialisation (BF5) L'étape d'initialisation ajoute des amers dans la carte en calculant leurs paramètres initiauxpar la méthode de l'inverse de profondeur. Le temps de traitement de l'étape d'initialisation de nouveaux amers dépend du :

— Type de l'environnement
— Nombre d'amers à initialiser
— Nombre d'amers dans la carte
— Nombre de particules

Le nombre d'occurrence moyen de cette étape est calculé en fonction du nombre des nouveaux amers à initialiser dans la carte. (voir 7.3, section 7.3.4.2

Bloc de rééchantillonnage (BF6) Cette étape met à jour le poids de chaque particule en fonction des résultats de l'étape d'estimation. Cette mise à jour dépend du :

— Type de l'environnement
— Nombre d'estimations réalisées
— Nombre de particules

Après avoir mis à jour le poids de chaque particule, cette étape vérifie qu'une particule n'a pas un poids trop faible. Si une particule représente une probabilité trop faible, elle est remplacée par une copie de la particule ayant le poids le plus élevé. Ceci est réalisé en utilisant une méthode de rééchantillonnage systématique décrite au paragraphe 3.3.7 dans le chapitre 3. Le temps de cette étape dépend du nombre de particules à dupliquer. Le nombre d'occurrences moyen de ce bloc est déterminé en fonction

du nombre des estimations réalisées. La théorie de rééchantillonnage systématique est détaillée dans l'annexe 7.4 section 7.4.4.

4.4 Implémentation sur une architecture multi-cœurs homogène

4.4.1 Temps d'exécution global

Pour effectuer une première évaluation du temps de calcul de l'algorithme, ce dernier est implanté séquentiellement sur le processeur OMAP4460 cadencé à 1.2 GHz (aucune optimisation n'est réalisée). Pour exploiter ce processeur nous avons utilisé la carte Pandaboard ES décrite dans le chapitre 3, tableau 3.1. Pour étudier les temps d'exécution de l'algorithme, deux types d'environnements ont été utilisés. Nous avons défini pour chaque environnement des seuils pour chaque paramètre de dépendance. Nous avons évalué l'algorithme sur le 3éme environnement de simulation définis dans la Figure 2.2, Chapitre 2. Ensuite nous avons évalué l'algorithme sur une partie de la trajectoire de l'environnement réel Figure.2.3, Chapitre 2

Environnement de simulation

- Données odométriques : une seule acquisition par itération
- L'amer est identifié, la correspondance entre l'observation et les amers de la carte est connue
- Le nombre maximum d'amers dans la carte de chaque particule est 150.
- Le nombre maximum d'observations est 90.
- Le nombre de particules est 100.

Environnement réel

- La taille de l'image est 320×240 pixels.
- La taille du descripteur est 16×16 pixels.
- Le nombre maximum d'amers dans la carte de chaque particule est 300.
- Le nombre maximum d'observations est 800.
- Le nombre de particules est 100.

4.4.1.1 Évaluation avec un jeu de données simulé

Le tableau 4.1 résume les temps d'exécution sur un seul cœur de l'OMAP4 des différent blocs fonctionnels du FastSLAM2. Les temps d'exécution sont calculés avec les données d'un environnement simulé. Le bloc de prédiction (FB1) est exécuté en 4.6 μs pour une seule particule. Il est exécuté une seule fois sur le jeu de donné simulé dans chaque itération de l'algorithme. Donc le temps d'exécution moyen pour 100 particules est 0.46 ms. Les deux blocs fonctionnels FB3 et FB4 ne sont exécutés que s'il y a au moins un amer apparié par itération. Dans les évaluations en simulation, 51 est approximativement le nombre moyen des amers appariés par itération dans tous le processus. Donc le temps d'exécution moyen par itération est respectivement 61.87 ms pour FB3 et 107.2 ms pour FB4. Le bloc d'initialisation FB4 est exécuté en 0.126 ms pour un nombre d'occurrences égale à 0.6. Finalement, La phase de rééchantillonnage est exécutée une fois par itération dans 0.06 ms. Le temps d'exécution global sur un seul cœur de l'OMAP4 est 169.76 ms.

Table 4.1: *Temps d'exécution des BFs sur l'architecture mono-cœur pour un environnement simulé*

Blocs Fonctionnels	MOI	TPO $t_{\mathrm{BF_x}}(\mu s)$	TPO-100 $t_{\mathrm{BF}_x}(\mu s)$	TEMI $t'_{\mathrm{BF_x}}(ms)$
FB1 : Prédiction	1	4.6	460	0.460
FB3 : Mise à jour de la position des particules	51.56	12	1200	61.872
FB4 : Estimation	51.56	20.80	2080	107.244
FB5 : Initialisation	0.6	2.10	210	0.126
FB5 : Rééchantillonnage	1	60	60	0.060
Total (ms)	-	-	-	**169.762**

MOI : Moyenne d'Occurrence par itération.
TPO : Temps d'exécution par Particule par Occurrence.
TPO-100 : Temps d'exécution par 100 particules par Occurrence.
TEMI : Temps d'Exécution Moyen par Itération.

4.4.1.2 Évaluation avec un jeu de données réelles

Le tableau 4.2 résume les temps d'exécution sur un seul un cœur de l'OMAP4 des différents blocs fonctionnels du FastSLAM2. Les temps sont calculés avec des données d'un environnement réel sur une trajectoire courte. Le bloc de prédiction (FB1) est exécuté en 4.6 μs pour une seule particule. Il est exécuté une seule fois sur le jeu dans chaque itération de l'algorithme. Donc le temps d'exécution moyen pour 100 particules est 0.46 ms. Dans le bloc de traitement d'image (FB2), le temps d'exécution du détecteur FAST dépend de la taille de l'image (le jeu de donnée réel fournit des images de taille $320x240$ pixels). Il est exécuté en 3.35 ms une seul fois à chaque itération. La phase de l'appariement est exécutée dans 3.14 ms avec un MOI égale à 785.6. Les deux blocs fonctionnels FB3 et FB4 ne sont exécutés que s'il y a au moins un amer apparié par itération. Dans les évaluations en environnement réel, 36.25 est approximativement le nombre moyen des amers appariés par itération dans tous le processus. Donc le temps d'exécution moyen par itération est respectivement 43.5 ms pour FB3 et 75.4 ms pour FB4. Le bloc d'initialisation FB4 est exécuté en 4.35 ms pour un nombre d'occurrence égale à 20.73. Finalement, La phase de rééchantillonnage est exécutée une fois par itération dans 0.06 ms. Le temps d'exécution global sur un seul cœur de l'OMAP4 est 130.76 ms. La 4.4.4 analyse et discute la différence entre le temps obtenu avec un jeu simulé et le temps obtenu avec un jeu réel.

Table 4.2: *Temps d'exécution des BFs sur l'architecture mono-cœur pour un environnement réel*

Blocs Fonctionnels		MOI	TPO $t_{\mathrm{BF_x}}(\mu s)$	TPO-100 $t_{\mathrm{BF}_x}(\mu s)$	TEMI $t'_{\mathrm{BF_x}}(ms)$
FB1 :Prédiction		1	4.6	460	0.460
FB2 : Traitement d'image	Détection FAST	1	3356	3356	3.356
	Appariement	785.6	4.83	4.83	3.1424
FB3 : Mise à jour de la position des particules		36.25	12	1200	43.5
FB4 : Estimation		36.25	20.80	2080	75.4
FB5 : Initialisation		20.73	2.10	210	4.35
FB6 : Rééchantillonnage		1	60	60	0.060
Total (ms)		-	-	-	**130.26**

MOI : Moyenne d'Occurrence par itération.
TPO : Temps d'exécution par Particule par Occurrence.
TPO-100 : Temps d'exécution par 100 particules par Occurrence.
TEMI : Temps d'Exécution Moyen par Itération.

4.4.2 Optimisation parallèle

La spécificité du processeur OMAP4460 est la disposition de deux cœurs Cortex A9 possédants chacun sa propre unité de calcul vectoriel. Dans cette section, on propose des optimisations parallèles de l'algorithme sur les deux cœurs de l'OMAP pour accélérer d'avantage le temps d'exécution global. Pour cela, des adaptations de chaque partie de l'algorithme sont proposées pour exploiter les deux processeurs disponibles. Expérimentalement, ces optimisations ont été programmées à l'aide de OpenMP.

Bloc de prédiction (FB1) La prédiction de la position des particules est parallélisée au niveau des deux cœurs. Chaque cœur réalise la prédiction d'un ensemble de particules. Le traitement de la position des particules par le modèle d'évolution est réalisé séquentiellement sur chaque cœur du processeur.

Bloc de traitement d'image (FB2) La phase de détection par le détecteur FAST a été parallélisée aussi sur les deux cœurs du processeur OMAP. Pour cela, l'image provenant de la caméra est séparée en deux parties. Les deux imagettes sont traitées par les deux cœurs. Ces calculs sont indépendants les uns des autres. On ne réalise pas de traitement spécifique pour regrouper les résultats. En effet, le détecteur ajoutera peu de points d'intérêt le long de la ligne de coupe. Pour optimiser efficacement la phase de mise en correspondance sur l'architecture Dual-Core de l'OMAP4460, nous avons choisi de paralléliser la tâche toute entière. L'exécution de la boucle d'appariement est parallélisée sur l'ensemble des observations détectées, chaque cœur du processeur traitera l'appariement d'un amer de la carte avec l'observation simultanément.

Bloc de la mise à jour de position des particules (FB3) La mise à jour de la position des particules est parallélisée au niveau des deux cœurs. Chaque cœur met à jours la position d'un ensemble de particules. Le traitement au niveau du calcul de la nouvelle distribution probabiliste est réalisé séquentiellement sur chaque cœur du processeur. Le calcul de la nouvelle distribution probabiliste pour chaque particule peut être parallélisé au niveau des amers. Malheureusement, cela n'est pas possible vu les contraintes de l'architecture étudiée. Le processeur OMAP4460 ne dispose pas d'assez de cœurs suffisant qui lui permettent de réaliser ce type de calcul et de paralléliser le calcul au niveau des boucles imbriquées.

Bloc d'estimation (FB4) Les amers appariés doivent être mis à jour pour chaque particule dans le filtre. Le calcul est parallélisé de sorte à ce que chaque cœur réalise la mise à jour des amers pour un ensemble de particules. La mise à jour des amers par les équations de Kalman est réalisée séquentiellement sur chaque cœur du processeur. L'architecture Dual-Core du processeur ne permet pas de paralléliser le calcul au niveau de la mise à jour des amers.

Bloc d'initialisation (FB5) Les amers non appariés doivent être ajoutés dans la carte de chaque particule dans le filtre. Chaque cœur réalise l'initialisation de l'amer pour un ensemble de particules. La procédure d'initialisation par inverse de profondeur est réalisée séquentiellement sur chaque cœur du processeur. De nouveau, le calcul parallèle au niveau des bloc imbriqués d'initialisation n'est pas possible sur l'architecture de l'OMAP4460.

Bloc de rééchantillonnage (FB6) La parallélisation du bloc de rééchantillonnage sur les deux cœurs du processeur n'est pas évidente. Des modifications algorithmiques sont nécessaire afin d'adapter

le calcul dans ce bloc à l'architecture étudiée. Le bloc FB6 peut être découpé en sous-blocs fonctionnels réalisant le calcul suivant :

— Mise à jour du poids total (BF6.1)
— Le calcul de la somme des poids des particules (BF6.2)
— La normalisation des poids des particules (BF6.3)
— Le calcul du nombre de particules efficaces avant rééchantillonnage (BF6.4)
— La génération systématique de l'intervalle de sélection (BF6.5)
— Le calcul de la somme cumulative (BF6.6)
— La conservation/suppression des particules et des amers (BF6.7)

Algorithme 3 : Bloc Fontionnels BF6

1. Mise à jour du poids total BF6.1;
   $w_t^m = w_t^m * \prod_{i=1}^{N_m} \omega_i$;
2. Somme des poids BF6.2;
   $w_s = \sum_{m=1}^{M} w_t^m$;
3. Normalisation des Poids BF6.3;
   $w_{t_n^m} = \frac{w_t^m}{w_s}$;
4. Nombre optimal des particules BF6.4;
   $N_{eff} = \frac{1}{\sum_{m=1}^{M}(w_t^m)^2}$;
5. Génération de l'intervalle de sélection BF6.5;
   $det[0] = k$;
   $det[i] = det[i-1] + k$;
   $sel[i] = det[i] + Rn$;
6. Calcul de la somme cumulative BF6.6;
   $w_c^m = w_{t_n^m} + w_c^{m-1}$;
7. Conservation ou suppression des particules et des amers BF6.7;

Le calcul dans les blocs BF6.1, BF6.2, BF6.3 et BF6.4 ne présente pas des dépendances de données. Par conséquent, le calcul est parallélisé au niveau des particules. Le calcul du poids total dans BF6.1 est parallélisé sur les deux cœurs du processeur. Chaque cœur calcul le poids total pour chaque particule en se basant sur le nombre des amers appariés. Pour BF6.2, chaque processeur réalise une sommation partielle d'un ensemble de poids des particules, ensuite un des deux cœurs du processeur calcul le poids total. Pour BF6.3, chaque cœur calcule le poids normalisé d'un ensemble de particules. Il en est de même pour BF6.4, le calcul est parallélisé au niveau des particules et chaque cœur calcul la somme carrée des poids des particules, la valeur efficace est calculée enfin sur un seul cœur du processeur.

Contrairement aux deux bloc BF6.5 et BF6.6, le calcul présente des dépendances de données. Autrement dit, le calcul de l'élément i dépend de celui de $i-1$, la parallélisation devient donc compliquée. Pour cela, nous proposons des modifications algorithmiques afin de porter les deux blocs sur les deux cœurs de l'architecture de l'OMAP.

Une description de l'implantation parallèle du bloc BF6.4 est illustrée dans la Figure 4.1. La génération de l'intervalle de sélection est faite à partir d'un intervalle déterministe. La taille de l'intervalle sélectif est M, M est le nombre de particules. Tout d'abord, on génère l'intervalle déterministe en initialisant la case 0 par la valeur $\frac{k}{2}$ (avec $k = \frac{1}{M}$) et la case numéro $\frac{M}{2}$ par la valeur $\left(\frac{k(1+M)}{2}\right)$. Ensuite les deux cœurs du processeur commence à générer en parallèle les autres valeurs de l'intervalle. Les deux cœurs continuent le traitement pour générer ensuite l'intervalle sélectif en parallèle.

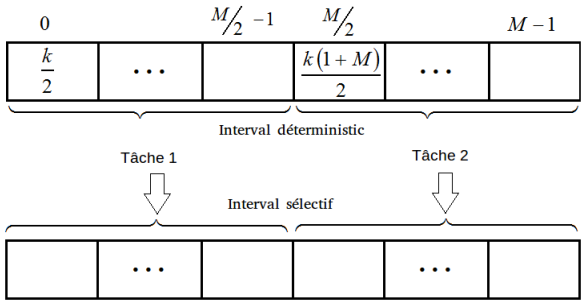


Figure 4.1: *Implémentation Parallèle de la génération de l'intervalle de sélection*

Pour paralléliser le calcul de la somme cumulative sur les deux cœurs du processeur, nous proposons d'utiliser un arbre sommateur [66]. Cette approche est couramment utilisée pour rendre parallèle un calcul séquentiel par nature. Il est basée principalement sur le principe de disperser ensuite recueillir. La figure 4.2 illustre l'utilisation de l'arbre ascendant et descendant pour le calcul de la somme cumulative des valeurs $\{5, 3, -6, 2, 7, 10, -2, 8\}$.

L'arbre ascendant calcule tout d'abord la somme de toutes le feuilles des sous-arbres internes de manière ascendante pour calculer enfin la somme 27. L'arbre descendant calcule de manière descendante la somme cumulative finale $\{5, 8, 2, 4, 11, 21, 19, 27\}$ en se basant sur les résultats de la somme partielle de l'arbre ascendant.

Le calcul dans le dernier sous-bloc BF6.7 ne présente aucune dépendance de données. Il est parallélisé au niveau des particules.

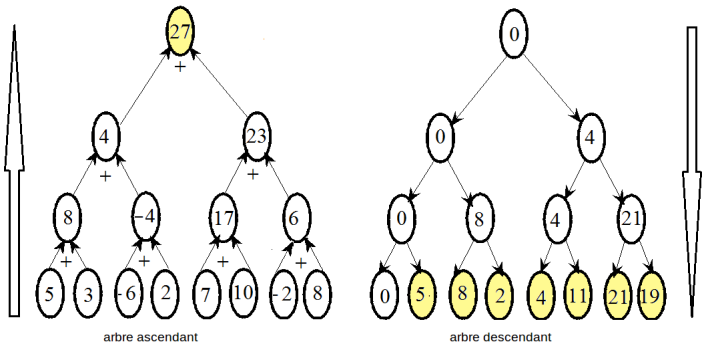


Figure 4.2: *Implémentation parallèle de la somme cumulative par l'arbre ascendant et descendant*

4.4.3 Évaluation des résultats de l'optimisation parallèle

Le tableau 4.3 résume les résultats d'exécution après parallélisation des blocs fonctionnels de l'algorithme sur l'architecture de l'OMAP pour un environnement simulé. Le temps d'exécution global de l'algorithme est de 169.76 *ms* sur un seul cœur de l'OMAP, tandis que le temps d'exécution global sur les deux cœurs de l'OMAP est 101.97. L'accélération globale obtenue par rapport à l'implémentation monocœur est de 1.67.

Table 4.3: *Temps d'exécution des BFs sur l'architecture multi-cœurs, environnement simulé*

	TEMI	
Blocs Fonctionnels	mono-cœur	multi-cœurs
FB1 : Prédiction	0.460	0.294
FB3 : Mise à jour de la position des particules	61.872	32.150
FB4 : Estimation	107.244	68.615
FB5 : Initialisation	0.126	0.092
FB5 : Rééchantillonnage	0.060	0.043
Total (ms)	**169.762**	**101.197**

TEMI : Temps d'Exécution Moyen par Itération.

Le tableau 4.4 résume les résultats d'exécution après parallélisation des blocs fonctionnels de l'algorithme sur l'architecture de l'OMAP pour un environnement réel. Le temps d'exécution global de l'algorithme est de 130.26 ms sur un seul cœur de l'OMAP, tandis que le temps d'exécution global sur les deux cœurs est 77.35. L'accélération globale obtenue par rapport à l'implémentation monocœur est 1.68.

Table 4.4: *Temps d'exécution des BFs sur l'architecture multi-cœurs, environnement réel*

		TEMI	
Blocs Fonctionnels		mono-cœur	multi-cœurs
FB1 : Prédiction		0.46	0.26
FB2 : Traitement d'image	Détection FAST	3.35	1.55
	Appariement	3.14	1.45
FB3 : Mise à jour de la position des particules		43.5	22.53
FB4 : Estimation		75.4	48.33
FB5 : Initialisation		4.35	3.19
FB6 : Rééchantillonnage		0.06	0.043
Total (ms)		**130.26**	**77.35**

TEMI : Temps d'Exécution Moyen par Itération.

4.4.4 Analyses des résultats

Nous avons évalué en deux situations l'algorithme FastSLAM2.0 sur l'architecture OMAP. Une première situation s'agit de l'environnement simulé et suppose que l'association entre l'amer et l'observation est connue. Une deuxième situation s'agit de l'environnement réel et suppose que l'association de donnée est inconnue. Cette évaluation nous a permis de mettre en évidence les paramètres de dépendance de chaque bloc fonctionnel de l'algorithme. Dans l'environnement simulé, l'association des donnée est supposée connue et l'amer est observé et acquit avec son identification dans la scène.

Dans ce cas, l'algorithme n'as pas besoin du bloc de traitement d'image. Par contre, en environnement réel l'association n'est pas connue et la phase de traitement d'image est nécessaire. La phase de traitement d'image contient deux parties indépendante, la détection et la mise en correspondance. La détection par FAST est exécutée une seule fois dans chaque itération dans environ 3 ms. La phase d'appariement est exécutée en fonction du nombre d'observations et des amers dans la carte. Environ 4 μs est le temps d'exécution nécessaire pour calculer la distance $ZMSSD$ entre une observation z_t et un amer de la carte m_t. Nous avons calculé le nombre d'itération du calcul de ZMSSD (MOI = 765.6) qui correspond au nombre moyen des observations détectées dans chaque itération. Le temps d'exécution de la phase de détection est environ 3.14 ms. Pour les deux environnements, réel et si-

mulé, le bloc de prédiction est exécuté dans environ 0.46 ms pour une seule occurrence par itération. Toutefois, ceci est valable juste pour des environnements réel de courte trajectoire où la position de la particule est prédite en utilisant juste la dernière donnée odométrique acquise entre deux points de référence de la réalité terrain. Pour des environnements large, des modifications algorithmiques, concernant la méthode de prédiction des particules, seront nécessaires. Nous verrons ses modifications dans les sections suivantes.

Dans l'environnement simulé, l'association est connue ce qui augmente le nombre des amers appariés (MOI) et augmente donc le temps de traitement. Contrairement à l'environnement réel, l'association n'est pas connue à l'avance, ceci augmente le taux d'erreur pour identifier l'amer. Dans ce cas le nombre des amers appariés (MOI) sera faible. Dans l'environnement simulé, 51.56 est le nombre d'itérations des deux blocs FB3 et FB4 ce qui donne un temps d'exécution respectivement égale à 61.872 ms pour FB3 et 107.24 ms pour FB4. Dans l'environnement réel, 36.25 est le nombre d'itérations des deux blocs FB3 et FB4 ce qui donne un temps d'exécution respectivement égale à 43.5 ms pour FB3 et 75.4 ms pour FB4.

Dans l'environnement simulé, peu d'amers sont initialisés par itération vu que l'association est connue et le taux de trouver un amer apparié est élevé. Contrairement à l'environnement réel, beaucoup d'amer sont initialisés par itération vu que l'association n'est pas connue et le taux de trouver un amer apparié est minimal. 0.126 ms est le temps d'exécution nécessaire pour initialiser un amer et calculer sa position initiale pour une seule particule dans un environnement simulé, ceci pour un MOI égale à 0.6. Dans l'environnement réel, 4.35 ms est le temps d'exécution pour initialiser un amer pour un MOI égale à 20.73. Le temps d'exécution de la phase de rééchantillonnage est 0.06 ms pour les deux environnements réel et simulé puisque ce bloc n'est exécuté qu'une seule fois. Cependant, des modifications algorithmiques de ce bloc seront aussi nécessaires pour adapter l'algorithme à un environnement large et complexe.

Dans les deux cas, on obtient à peu près le même facteur d'accélération de l'implémentation parallèle par rapport à celle réalisée sur un seul cœur de l'OMAP.

4.4.5 Évaluation des résultats de cartographie et de localisation

Le FastSLAM2.0 a été évalué sur un simulateur d'environnement défini dans le chapitre 2. Le FastSLAM2.0 a été testé dans un environnement interne qui constitue une petite salle de $49m^2$ à 4 mures et plusieurs points d'intérêts visibles. Les deux figures 4.3 et 4.4 représentent les résultats de simulation du FastSLAM2.0 monoculaire avec 100 particules. La trajectoire prédite construite par les données odométriques (ligne rouge) diverge par rapport à la référence (ligne bleue). Les amers vus pour la première fois sont initialisés par une incertitude considérable (cercle bleu). Les amers ayant été déjà vus ont une incertitude qui diminue et leur position est corrigée par le filtre de Kalman. Le FastSLAM2.0 utilise ensuite ces amers pour corriger la trajectoire du robot (ligne verte).

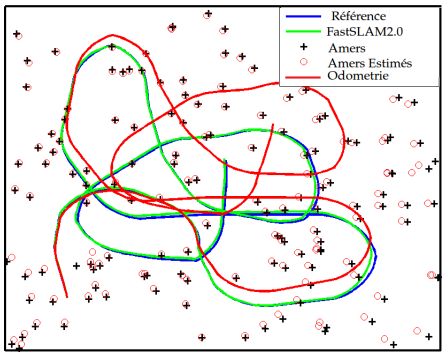


Figure 4.3: *Résultats de cartographie et localisation du FastSLAM2.0 monoculaire en simulation/ environnement 1*

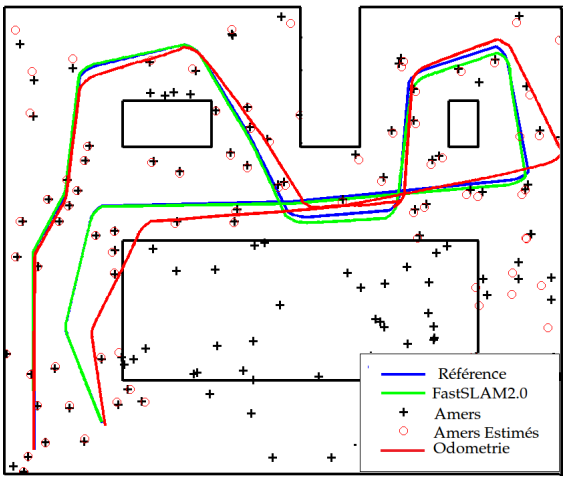


Figure 4.4: *Résultats de cartographie et localisation du FastSLAM2.0 monoculaire en simulation/ environnement 2*

Dans la figure 4.5 nous testant l'algorithme avec 100 particules mais cette fois-ci dans un environnement réel. Au début de l'expérimentation, la trajectoire fournie par le FastSLAM2.0 (verte) tend vers la réalité terrain (bleue) avec une erreur acceptable. Au fur et à mesure que le robot avance dans l'environnement, les résultats du FastSLAM2.0 deviennent de plus en plus imprécis. La trajectoire diverge et résulte d'une erreur de localisation plus grande que celle fournie par l'odométrie. L'algorithme tel qu'il a été développé peut opérer en simulation avec un nombre minimum de particules. Contrairement aux environnements larges, cela n'est pas suffisant pour avoir des résultats consistants de cartographie et de localisation. L'utilisation du FastSLAM2.0 dans des environnements larges nécessite une réécriture algorithmique. Nous proposons dans les sections suivantes les modifications que nous avons adoptées pour adapter l'algorithme aux environnements réels à grande échelle.

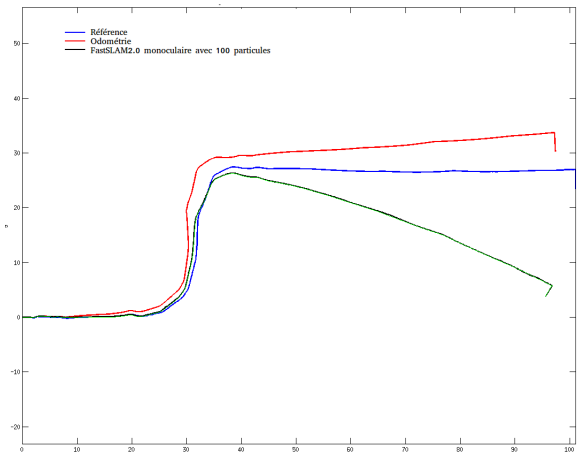


Figure 4.5: *Algorithme FastSLAM2.0 Monoculaire avec 100 particules en environnement réel (Rawseeds)*

4.5 Le FastSLAM2.0 monoculaire à grande échelle

4.5.1 Convergence du FastSLAM2.0

Le FastSLAM2.0 est une version développée pour surpasser l'ancienne version et résoudre le problème de l'appauvrissement des particules décrit dans le chapitre 3, section 3.3.4. Cette version utilise une distribution probabiliste modifiée qui incorpore les toutes dernières observations afin de mettre la position des particules à jour. Ces améliorations faites par [41] résultent d'un algorithme qui peut converger avec nombre minimal des particules, voir une seule dans des cas particuliers. [41] ont mené des expérimentations systématiques du FastSLAM2.0 sur un environnement réel externe (le parc de Vectoria [131]). Ils ont démontré la convergence du FastSLAM2.0 avec une seule particule sur l'environnement du test. Cependant, la version qu'ils ont testé est basée sur un capteur Laser. Ce type de capteur est capable de donner une information précise sur la profondeur des amers dans la scène. Les amers sont donc facilement initialisés dès leur première observation. Leur position initiale est calculée par le modèle d'observation inverse utilisant l'information de la profondeur. Ceci permet la correction de la position du robot de manière fiable et robuste et fournit donc des résultats consistants même avec une seule particule. Toutefois, ce n'est pas toujours le cas avec la version monoculaire du FastSLAM2.0.

Le nombre de particules dans le FastSLAM2.0 monoculaire est important pour garder une estimation raisonnable de la position du robot et de l'incertitude des amers [127]. La caméra ne peut donner aucune information sur la profondeur des amers. Pour cela, on fait appel à des méthodes pour estimer la profondeur des amers détectés dans l'image. Une de ces méthode utilisée dans notre étude est la paramétrisation par inverse de profondeur. Cette méthode initialise les amers dès leur première observation dans la carte mais avec une grande incertitude sur la profondeur. La position est corrigés avec les équations de Kalman à chaque fois que l'amer est observé. Le nombre de particules utilisées doit être augmenté pour garder une estimation cohérente de l'incertitude du robot.

[127] a mené des expérimentations du FastSLAM2.0 monoculaire avec 50, 250 et 1000 particules afin d'évaluer l'effet du nombre de particules sur l'estimation de la position du robot et des amers. Il a démontré que 50 particules sont suffisantes mais sur une séquence très courte avec peu d'images. Cependant, une analyse plus détaillée et rigoureuse est nécessaire pour des grandes séquences d'images et des environnements à grande échelle. Il reste encore des défis importants et des contraintes à tenir en compte pour permettre au FastSLAM2.0 monoculaire d'opérer dans des environnements larges. Le

nombre exacte des particules nécessaires pour la convergence du FastSLAM2.0 monoculaire n'est pas encore défini et peut augmenter avec la complexité de l'environnement. Durant nos évaluation, nous avons utilisé un environnement large avec 5000 images pour tester l'algorithme. Pour cela, nous avons augmenté le nombre de particules pour avoir des estimations consistantes des différentes incertitudes.

4.5.2 Convergence de l'incertitude des particules

Dans le FastSLAM2.0, après la prédiction des particules dans FB1, ces dernières sont mises à jour dans le bloc FB3 à partir d'une distribution probabiliste construite de manière incrémentale à partir des dernières observations. Le problème qui s'impose, c'est que cette nouvelle distribution est construite à partir d'une certaine estimation de l'incertitude des particules notée P_m. Une initialisation aléatoire de cette incertitude peut conduire à la divergence du filtre et donc affecter les résultats de localisation.

Les auteurs dans [41] et [127] n'ont donné aucune information sur comment cette incertitude initiale est calculée. L'initialisation empirique de cette matrice de covariance n'est pas une solution sage pour le FastSLAM2.0 monoculaire qui doit opérer dans un environnement large. Autrement dit, comme on a vu dans la section précédente, le FastSLAM2.0 monoculaire utilise une méthode d'initialisation partielle des amers de la carte. L'initialisation partielle repose principalement sur l'incertitude des particules. Une mauvaise estimation de cette incertitude affecte la décision de l'initialisation complète des amers. La position des particules ne sera donc pas corrigée de manière fiable et robuste, ce qui entraîne la divergence du filtre. Ceci est démontré dans notre travail [7]. Avec une utilisation d'un nombre minimal de particules avec une initialisation empirique de la matrice P_m, l'algorithme ne peut cartographier qu'une petite portion de la trajectoire et n'est pas capable de fermer la boucle sur une grande trajectoire.

Pour bien maîtriser l'incertitude des particules, nous proposons de calculer mathématiquement la matrice de covariance P_m à chaque fois que la position des particules est prédite pour tenir en compte l'accumulation de l'incertitude entre deux acquisitions d'image. Ceci reflète mieux l'incertitude du robot pour une meilleur initialisation partielle. La matrice P_m est réinitialisée à chaque acquisition d'image. La procédure de prédiction et du calcul de P_m est donnée dans l'Algorithme 4.

Algorithme 4 : Bloc FB1 : prédiction des particules et leur incertitude

$P_m = 0$;
for *each odometric aqcuisition* **do**
  for *each particle* m **do**
    $s_t^m = f(s_{t-1}^m, u_t)$;
    $G_u = \frac{\partial f}{\partial(s_x, s_y, s_\theta)}$;
    $G_p = \frac{\partial f}{\partial(\delta_s, \delta_\theta)}$;
    $P_m = G_p P_m G_p^T + G_u Q G_u^T$;
  end
end

La méthode proposée est coûteuse en termes de temps de calcul, puisque il faut calculer P_m à chaque acquisition des données odométriques. Comme notre travail concerne le calcul embarqué des algorithmes SLAM, cette méthode assure une estimation consistante des incertitudes mais au détriment des performances temps réel. Des optimisations seront donc nécessaires pour permettre l'exécution temps réel sans affecter la consistance des résultats.

4.5.3 Gestion de rééchantillonnage

Le FastSLAM2.0 monoculaire utilise une initialisation partielle des amers. Les amers sont rajoutés dans la carte dès la première observation, mais ils sont partiellement initialisés avec une grande incertitude sur la profondeur. Leur profondeur est corrigée au fur et à mesure qu'ils sont ré-observés jusqu'à l'obtention d'une profondeur raisonnable. Ils seront ensuite initialisés et leur position dans le repère du monde est calculée. Pour ne pas affecter les résultats de localisation, la position du robot n'est corrigée qu'à partir les amers qui sont complètement initialisés. Pour cela, dans la phase de rééchantillonnage et durant la mise à jour du poids total, les probabilités calculées par rapport aux amers mis en correspondance, partiellement initialisés ne sont pas tenues en compte et sont écartés. Ces probabilités présentent des valeurs aberrantes qui peuvent entraîner à un mauvais rééchantillonnage des particules.

4.5.4 Évaluation des Résultats de cartographie et de localisation

Les résultats expérimentales du FastSLAM2.0 monoculaire après modifications sont montrés dans la figure 4.6. La trajectoire odométrique (ligne rouge) diverge au fur et à mesure que le véhicule avance vu que les données des encodeurs sont fortement bruités. L'environnement est riche en objets et contient de longs couloirs. Ceci permet la détection des amers qui ont été déjà vus plusieurs fois de manière fiable et robuste et permet à l'algorithme de corriger la trajectoire du robot. La qualité de localisation du FastSLAM2.0 est bonne comparée à l'intégration des données odométriques. La trajectoire définie par le FastSLAM2.0 (ligne verte) suit correctement le trajet de référence (ligne bleue) tout au long des couloirs explorés.

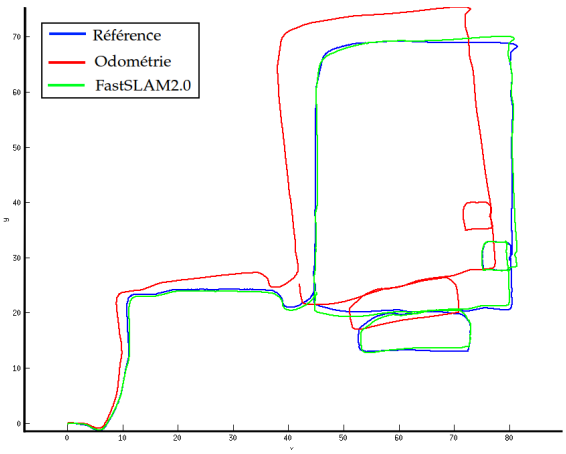


Figure 4.6: *Résultats de cartographie et de localisation du FastSLAM2.0 monoculaire après réécriture algorithmique*

La figure 4.7 montre l'erreur euclidienne de localisation du FastSLAM2.0 monoculaire par rapport à la réalité terrain. L'erreur euclidienne des résultats expérimentaux confirme la qualité et la consistance de localisation du FastSLAM2.0 monoculaire comparée à l'intégration odométrique. En effet, l'intégration odométrique souffre d'une erreur cumulative qui augmente au fur et à mesure de l'expérimentation alors que le FastSLAM2.0 a une erreur qui demeure stable et qui ne dépasse pas 40 cm. Le nombre de particules dans le FastSLAM2.0 affecte fortement les résultats de localisation. Plus le nombre de particules augmente plus les résultats sont consistants surtout dans environnements réels de grande taille. Ceci est confirmé dans la figure 4.8, qui montre une erreur euclidienne moyenne qui diminue en augmentant le nombre de particules.

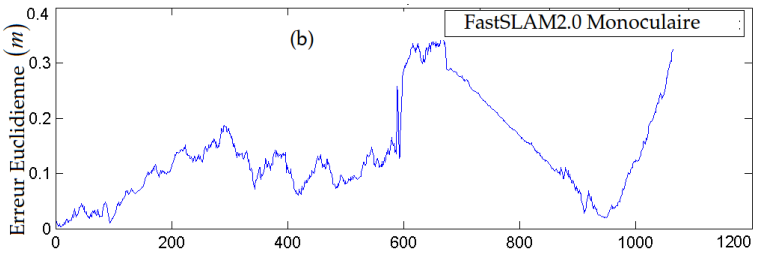


Figure 4.7: *Erreur euclidienne de localisation en fonction du nombre des étapes*

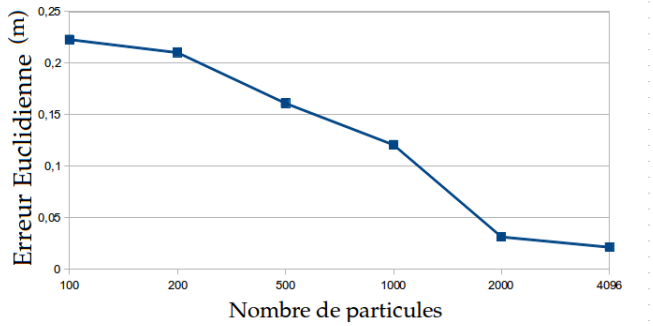


Figure 4.8: *Erreur euclidienne moyenne de localisation en fonction du nombre des particules*

L'utilisation du FastSLAM2.0 dans des environnements avec les modifications apportées peuvent augmenter d'avantage les temps de calcul surtout quand le nombre de particules augmente. En conséquence, une architecture avec un nombre de cœurs réduit n'est pas réellement adaptées au FastSLAM2.0 à grande échelle. Nous proposons dans les sections suivantes des optimisations sur des architectures massivement parallèles adéquates au FastSLAM2.0 monoculaire pour opérer en environnements larges.

4.6 Implémentation sur une architecture massivement parallèle

Le FastSLAM2.0 monoculaire qu'on souhaite implémenter est censé opérer dans un environnent large avec plusieurs particules. Il réalise plusieurs boucles de calculs indépendantes pour chaque particule : ces calculs peuvent être réalisés de manière parallèle. Le processeurs OMAP4460 ne dispose que de deux cœurs et ne sont donc pas adaptés au calcul des blocs fonctionnels avec un grand nombre de particules. Pour cela, des architectures massivement parallèles sont nécessaire pour permettre une exécution temps réel de l'algorithme.

4.6.1 Choix d'une architecture adaptée

Pour sélectionner une architecture massivement parallèle, adéquate à l'algorithme FastSLAM2.0, nous avons opté pour plusieurs architectures à système hétérogène intégrant un CPU et un GPU ensemble sur la même puce. L'évaluation sur plusieurs architectures nous permettra de prendre en considération les contraintes imposées par chaque architecture afin d'aboutir à un choix adéquat d'une architecture dédiée au FastSLAM2.0 monoculaire. Le tableau 4.5 résume les caractéristique des GPUs embarqués utilisés dans notre évaluation.

La Pandaboard ES intègre un GPU graphique de type SGX 544 pour le traitement d'image cadencé à 200 MHz. La iMX6 Sabre intègre un GPU de type Vuvante GC2000 contenant 4 cœurs SIMD cadencés à 500MHz. L'ODROID XU4 intègre un GPU de type Mali-T628-MP6 contenant 8 SIMD cœurs cadencés à 600 MHz. La Tegra X1 intègre un GPU performant de type NVIDIA Maxwell de 256 NVIDIA CUDA cœurs cadencé à 950 MHz. La TK1 intègre aussi un GPU performant de type NVIDIA Kepler avec 192 CUDA cœurs cadencés à 850 MHz. Le tableau 4.6 résume les caractéristique des deux GPU de haute-gamme utilisées dans notre étude pour des raison de comparaison. La machine de bureau intègre un GPU de type NVIDIA GTX 540M contenant 336 CUDA cœurs cadencés à 950 MHz. le PC portable intègre un processeur graphique de type intel cadencé à 533 MHz contenant 8 SIMD cœurs.

Table 4.5: *Spécifications des GPUs embarqués*

	Systèmes Embarqués				
	Tegra X1	Tegra K1	ODROID XU4	i.MX6 Sabre Lite	Pandaboard ES
GPU	NVIDIA Maxwell	NVIDIA Kepler	ARM MALI-T628-MP6	Vivante™ GC2000	PowerVR SGX540
GPU Mhz	900	850	600	500	200
Nombre de cœurs	256	192	8	4	4
Languages	CUDA OpenGL	CUDA OpenGL	OpenCL OpenGL	OpenCL OpenGL	OpenGL ES

Table 4.6: *Spécifications des GPUs de haute-gamme*

	Station de travail	PC Portable
GPU	Nvidia GTX 540M	Intel Graphics Accelerator
GPU Mhz	950	533
Nombre de cœurs	336	8
Languages	CUDA/OpenCL OpenGL	OpenCL/OpenGL

Programmation des GPUs L'accès aux ressources des GPUs par un programmeur peut se faire à travers plusieurs langages : CUDA, OpenGL, OpenCL et C pour le graphique. Le langage CUDA présente aux utilisateurs une interface basée sur le langage C pour des applications de développement direct sur les GPUs de NVIDIA. Ceci, limite la portabilité du code spécifiquement pour les GPUs de NVIDIA.

Le développement avec OpenCL sur les GPUs reste souvent très limité [132]. Les pilotes "Drivers" de l'OpenCL pour certains GPUs embarqués, tel que la TK1 et TX1, ne sont pas disponibles pour le grand public. Les GPUs embarqués qui supportent OpenCL pour le développement GPGPU contiennent un nombre de cœurs très limités et ne sont pas vraiment destinés aux calculs haute-performances (*HPC : High-performances computing*). Les GPUs embarqués disponibles qui supportent OpenCL sont mentionnés dans le tableau 4.5. Le Vivante GC2000, Mali-T628 MP6 et le MALI-T604-MP4 contiennent respectivement 4, 8 et 4 SIMD cœurs.

L'OpenGL pour le calcul générique sur GPU est couramment utilisé [66]. La majorité des GPU, embarqués et de haute-gammes, supportent OpenGL qui reste à son tour une plateforme indépendante

d'une architecture spécifique. L'utilisation de l'OpenGL nécessite parfois une connaissance et des prés-requis dans la programmation graphique. Pourtant, il fournit une portabilité du code entre différents GPUs. Ceci rend l'implémentation résultante indépendante et permet de mener des études comparative entre les architectures à base des GPUs.

Notre étude vise l'évaluation du FastSLAM2.0 sur plusieurs architectures embarquées (CPU, GPUs et FPGA). Pour rester indépendant de la technologie du constructeur, nous adoptons le développement par OpenCL et OpenGL pour réaliser des optimisations parallèles. Ces deux langages sont les plus répandus et les plus supportés par les différentes architectures de calcul parallèle.

4.6.2 Adéquation algorithme architecture

4.6.2.1 Définition du graphe d'algorithme et étude des contraintes

Bloc de prédiction (FB1) La figure 4.9 montre le graphe orienté acyclique du bloc de prédiction FB1. La figure 4.9-a montre le graphe composé simplifié du bloc. Ce bloc prend en entrée les données des encodeurs $u_t\,(n_r, n_l)$ et l'état des particules à l'instant précédent pour calculer la position future des particules. L'opération de prédiction est indépendante pour chaque particule. Chaque unité de calcul peut donc traiter une seule particule de façon indépendante, cela est montré dans le graphe de la figure 4.9-b. Chaque sous-graphe fb_1 calcule la position d'une seule particule.

La densité des particules doit être distribuée de manière consistante de telle sorte à ce qu'elle représente la position réelle du robot. Autrement dit, les particules générées après l'intégration des données odométriques doivent couvrir la vraie position du robot dans l'espace. En conséquence, le modèle d'évolution doit être appliqué avec une répartition aléatoire pour tenir en compte les différents bruits dans le système du mouvement du SLAM (localisation de Monte Carlo [35]). Ceci est modélisé par le sous-bloc *cd* qui signifie le calcul de déplacement. Ce petit bloc récupère les données des encodeurs de deux dimension issues du capteur pour générer en sortie un ensemble hétérogène distribué des déplacements pour satisfaire une distribution cohérente des particules. Nous avons visé une implémentation parallèle de ce bloc sauf que la génération des nombres aléatoires n'est pas évidente à implémenter en parallèle. La génération des nombres aléatoires avec un grande degré de précision est souvent efficace quand elle est implémenté sur une seule unité de calcul [133]. Il existe des travaux qui ont essayé de proposer des méthodes pour la génération parallèle des nombres aléatoires tels que dans [134] [135], mais cela reste incompatible avec la localisation de Monte Carlo et demeure un sujet de recherche en cours [136, 133, 137].

La figure 4.9-c montre le graphe factorisé à l'aide des sommets frontières de factorisation pour simplifier la définition du bloc. Ce graphe fait apparaître 3 sommets frontières $S = \{F, F, J\}$. Le premier sommet "F" distribue les données odométriques bruités au différentes opérations modélisées par fb_1. Le deuxième sommet "F" distribue l'état précédent de chaque particule au différents unités de calcul. Le dernier sommet "J" regroupe la position de chaque particule après l'opération de prédiction pour donner en sortie l'ensemble S_t des particules à l'état future.

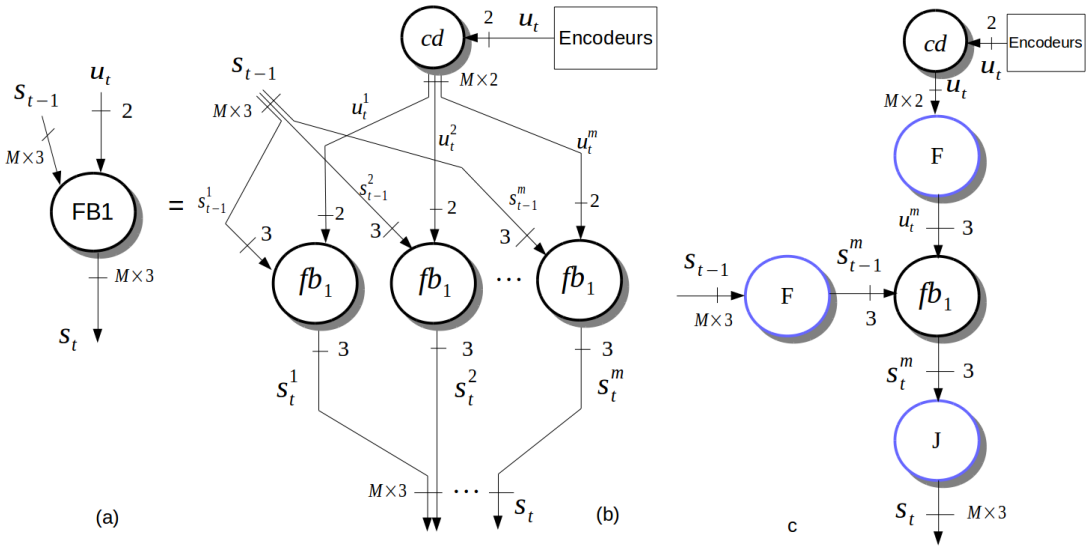


FIGURE 4.9: *Graphe du bloc du prédiction FB1 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontières de factorisation*

La figure 4.10 détaille le sous-graphe de l'opération fb_1. Il détaille le calcul des trois composantes $\left(s^m_{t_x}, s^m_{t_y}, s^m_{t_\theta}\right)$ de la m-ième particule. La figure 4.10-b correspond aux équations du modèle probabiliste d'évolution. La figure 4.10-c correspond au sous-graphe factorisé qui fait apparaître six frontières de factorisation à savoir $\{D, F, D, F, I, J\}$. Le sommet "I" signifie que le sous-graphe est répété de manière itérative pour calculer les trois composantes de la position. Les sorties de toutes les itérations sont ensuite regroupées par le sommet "J" pour constituer le vecteur de position. Les sommets D et F distribuent les entrés aux différentes unités de calcul.

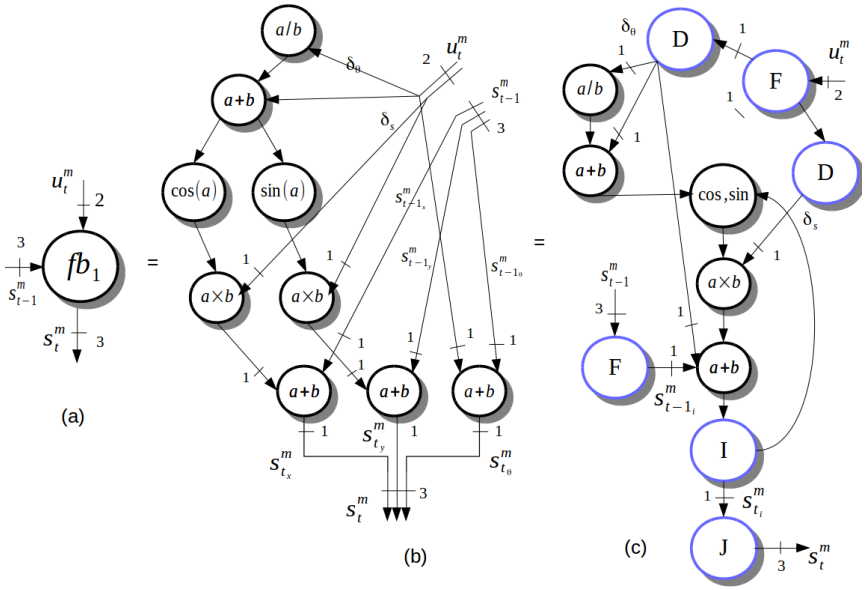


FIGURE 4.10: *Graphe du sous-bloc du prédiction FB1 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontières de factorisation*

La figure 4.11 résume le graphe global du bloc de prédiction. Le graphe fait apparaître trois zones frontières telle que $\{F_1, F_2, F_3\}$. Le graphe interagit avec le milieu extérieure (encodeurs) par le bais de F_1 qui se répète pour toutes les particules dans le système. La zone F_2 correspond à une seule unité

de calcul et F_3 calcule les composantes vecteurs de la position.

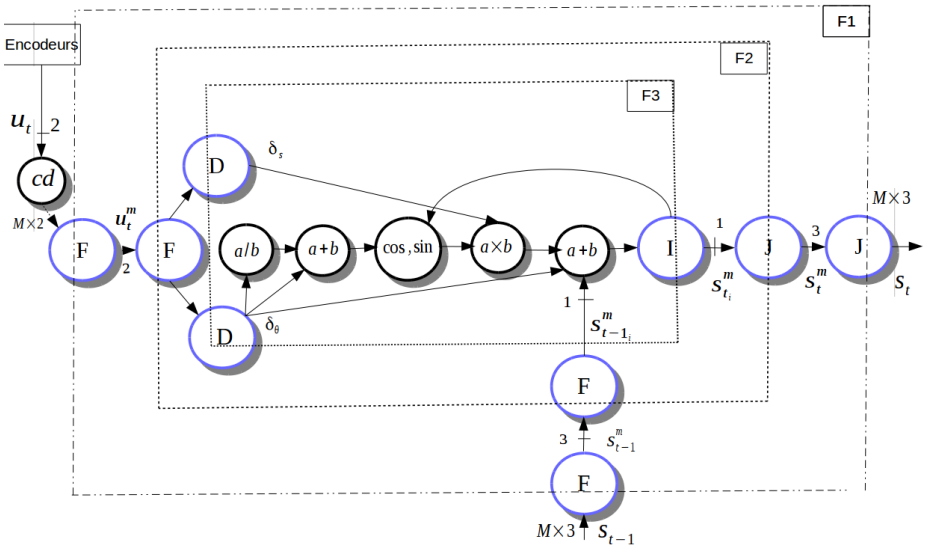


Figure 4.11: *Graphe finale du FB1 factorisé à laide des sommets frontières de factorisation*

Bloc de traitement d'images (FB2) La figure 4.12 montre le graphe orienté acyclique du bloc de traitement d'image FB1. Ce bloc prend en entrée les données images directement de la caméra et détecte les observations z_i en utilisant FAST. Les amers de la carte $\grave{X}_t^{m_w}$ sont projetés sur l'image acquise par la l'opération "Proj" en utilisant la position de particule qui a le plus grand poids $\bar{s}_t^{m_w}$ déterminée dans le bloc FB6. L'opération "ZMSSD" calcule la similarité entre les amers projetés $\widehat{z}_i$ et ceux détectés z_i. L'opération de calcul de ZMSSD entre une observation et les amers de la carte est indépendante. Chaque unité de calcul peut traiter une seule observation de façon indépendante. La sortie de cette fonction est un ensemble des amers appariés $(\hat{z}, z)$.

Nous avons vu dans le chapitre 3 que le FastSLAM2.0 a une complexité en $O(N.M.K)$. La complexité en M provient du traitement des particules dans le filtre à chaque itération de l'algorithme. La complexité en N provient de l'étape de ré-échantillonnage vu qu'il faut copier entièrement la carte de l'environnement si les particules sont dupliquées. Pour résoudre ce problème, nous avons procédé à une représentation par arbre binaire de la carte de chaque particule. En conséquence, l'opération "tree" parcours l'arbre binaire pour récupérer les huit paramètres des amers appariés pour toutes les particules dans le système $\hat{X}_t$. Ce vecteur est de taille $M\left(\hat{N}\times 8\right)$, M est le nombre des particules, $\hat{N}$ est le nombre des amers appariés.

L'implémentation de l'arbre binaire nécessite une allocation dynamique de la mémoire vu qu'il faut à chaque initialisation d'un nouvel amer rajouter une feuille qui lui correspond. L'allocation dynamique n'est pas supportée par les langages de programmation que nous avons utilisés pour la programmation du GPU. Nous proposons une solution lors de la présentation du graphe d'implémentation.

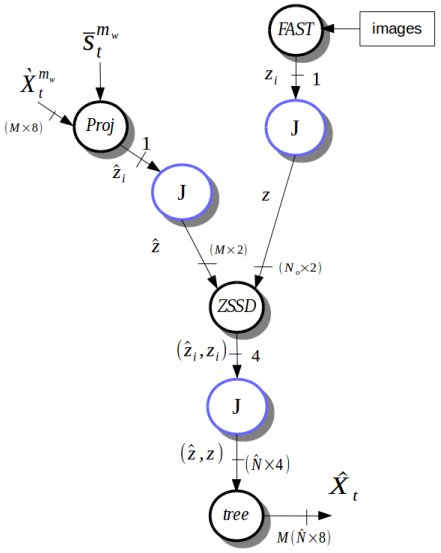


Figure 4.12: *Graphe final du bloc de traitement d'images : FB2 factorisé à l'aide des sommets frontières de factorisation*

Bloc de mise à jour des particules (FB3) La figure 4.13 montre le graphe orienté acyclique du bloc de mise à jour FB3. La figure 4.13-a montre le graphe composé simplifié du bloc. Ce bloc prend en entrée la position des particules S_t prédite, leur matrice de covariance P_t, l'ensemble des amers appariés $\hat{X}_t$, la sortie et la position mise à jour $\hat{S}_t$. La procédure de mise à jour de la position des particules est indépendante sur chaque particule. Chaque unité de calcul peut traiter une seule particule de façon indépendante, ceci est montré dans la figure 4.13-b. Chaque opération fb_3 traite une seule particule. La figure 4.13-c montre le graphe factorisé à l'aide des sommets frontières. La matrice de covariance P_t et la position prédite S_t des particules sont distribuées aux différentes opération $fb_3 \cdots fb_3^M$ par le sommet F. Tandis que les $\hat{N}$ paramètres $\hat{X_t} = (u, v, x_0, y_0, \rho, \theta, \phi, C_t)$ des amers appariés calculés dans (FB2) sont diffusés aux différentes opération par le sommet D. Le sommet "J" regroupe la position des particules pour donner en sortie l'ensemble $\hat{S}_t$mis à jour par rapport aux amers appariés.

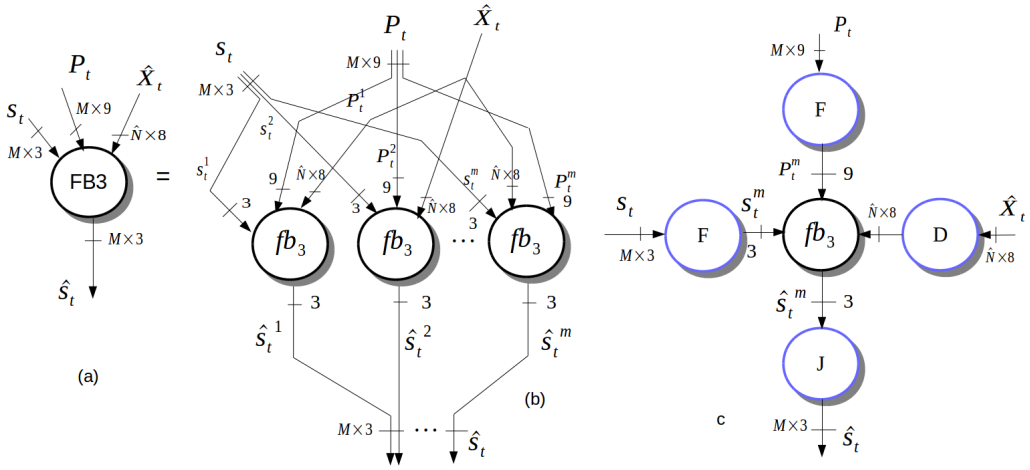


Figure 4.13: *Graphe du bloc de mise à jour des particules FB3 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontières de factorisation*

La figure 4.14 montre le graphe décomposé 4.14-b et factorisé 4.14-c de l'opération fb_3. Le graphe décomposé correspond à la procédure de mise à jour itérative de la position. Chaque graphe calcule la distribution probabiliste par rapport à un seul amer. Le calcul présente une dépendance de donnée et chaque graphe dépend de l'ancienne valeur calculée par le graphe précédent. Cette procédure itérative est répétée jusqu'à la construction finale de la position $\hat{s}_t^m$. Pour simplifier ce graphe, la figure 4.14-c le graphe factorisé à l'aide des sommets frontières. Les deux opérations Σ et μ calculent la moyenne et la matrice de covariance de la distribution probabiliste. La répétition itérative est définie par le sommet "I". Les sommets "F" et "D" respectivement distribuent et diffusent les données d'entrée à chaque opération.

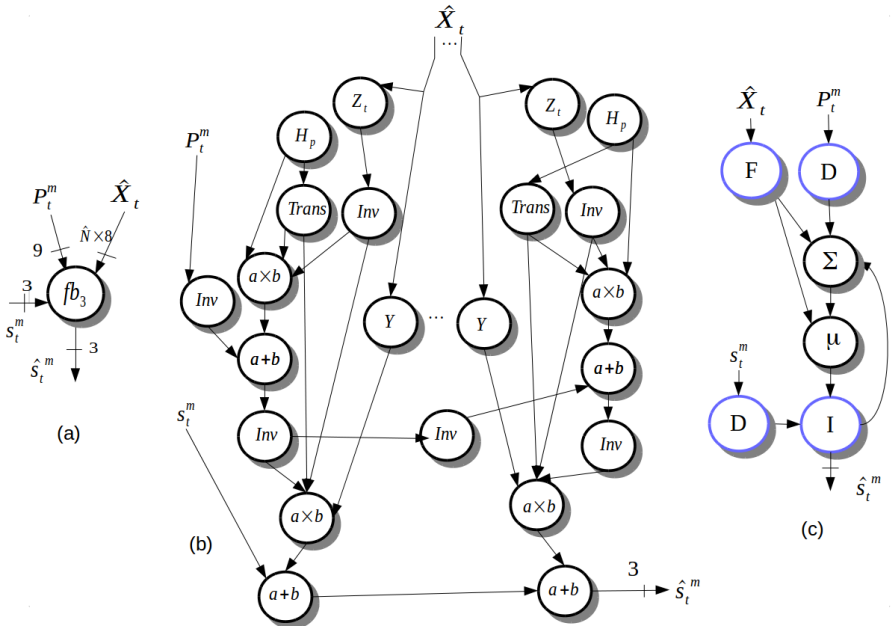


Figure 4.14: *Graphe du sous-bloc de mise à jour des particules FB3 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontières de factorisation*

La figure 4.15 montre le graphe total factorisé du bloc FB3. Le graphe fait apparaître deux zones frontières F1 et F2. La zone F1 interagit avec le milieux extérieur (bloc FB1) pour récupérer l'état des particules. Elle correspond en même temps aux différentes opérations fb_3. La zone F2 correspond au sous-graphe interne de chaque opération fb_3.

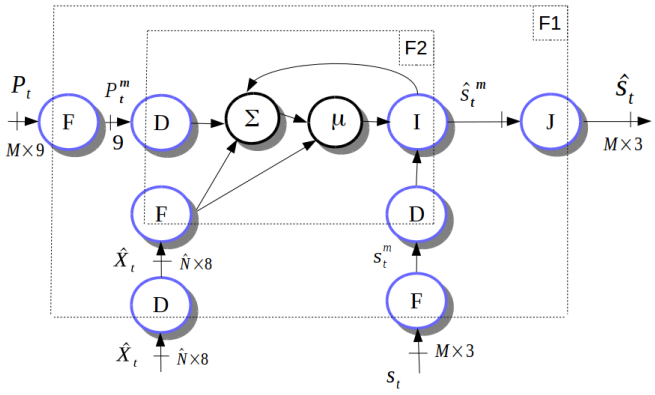


Figure 4.15: *Graphe final de FB3 factorisé à l'aide des sommets frontière de factorisation*

Bloc d'estimation (FB4) La figure 4.16 représente le graphe orienté acyclique du bloc d'estimation FB4. Le graphe de la figure 4.16-a montre le graphe simplifié du bloc. Ce bloc prend en entrée l'état des particules issue du bloc FB3 (S_t, P_t) et l'ensemble des amers appariés déterminé dans (FB2) $\hat{X}_t$. La sortie de ce bloc donne l'ensemble des amers corrigés $\grave{X}_t$et l'ensemble des poids Ω_t. La figure 4.16-b montre le graphe décomposé en plusieurs opérations nommé fb_4. Chaque opération met à jour les paramètres $(\rho, \theta, \phi, C_t)$ des amers appariés par les équations de Kalman pour la particule correspondante. Elle calcule aussi le poids de chaque particule Ω_t^m en se basant sur les paramètres des amers appariés, la position et la matrice de covariance (s_t^m, P_t^m) mises à jour des particules. L'opération de mise à jour des amers est indépendante et donc chaque unité de calcul peut mettre à jour les paramètres d'un amer de façon indépendante. La figure 4.16-c correspond au graphe factorisé de l'opération fb_4.

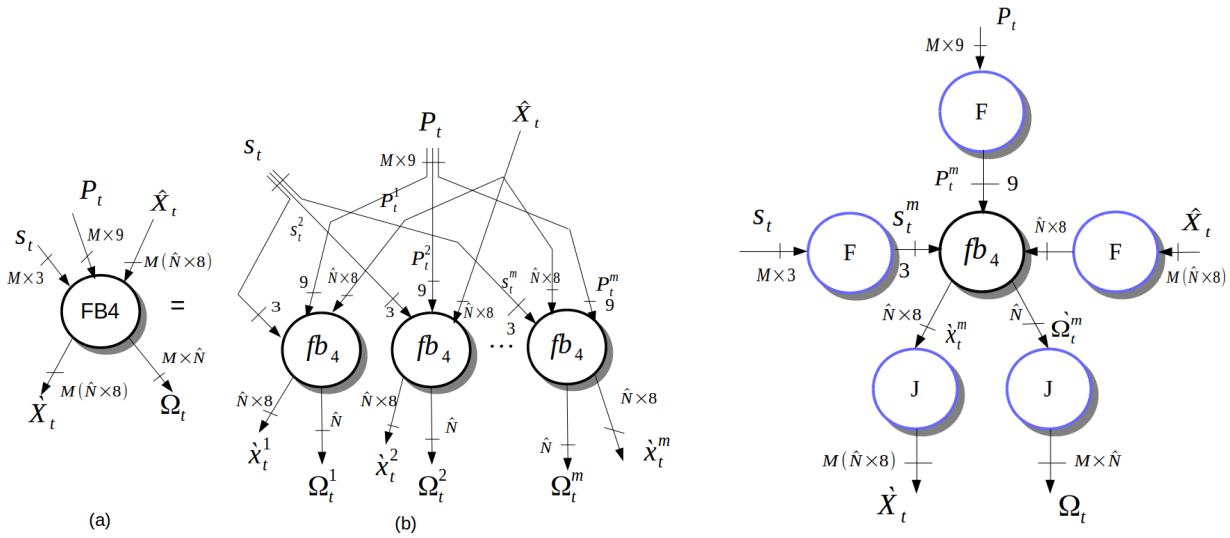


Figure 4.16: *Graphe du bloc d'estimation FB4 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontières de factorisation*

La figure 4.17 correspond au graphe décomposé de l'opération fb_4. Le graphe présente des sous-graphes qui corrigent les paramètres d'un seul amer par les équations ordinaire du filtre de Kalman et calculent le poids correspondant.

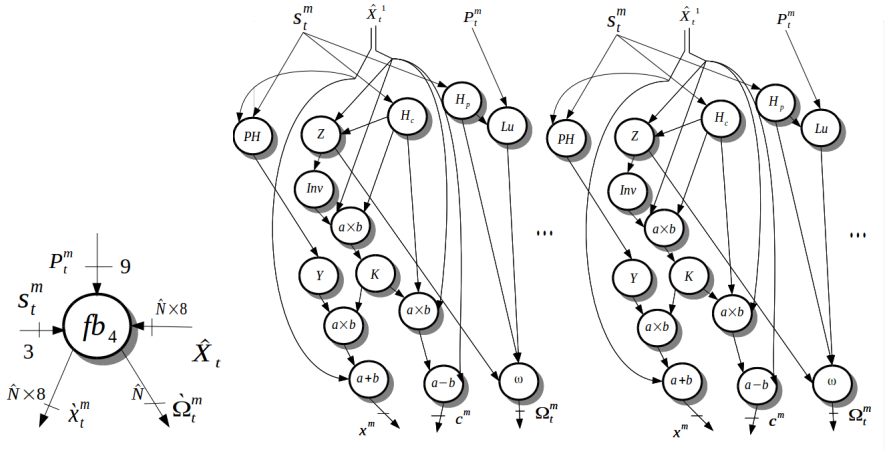


Figure 4.17: *Graphe du sous-bloc d'estimation FB4 : (a) graphe composé, (b) graphe décomposé*

La figure 4.18 résume le graphe global factorisé du bloc d'estimation. Il fait apparaître deux zone frontières F1 et F2 et plusieurs sommets de factorisation. Le bloc FB peut être résumé dans les deux opération K_x et w qui calculent respectivement l'ensemble des paramètres à jour et leur poids correspondants. Les sommets frontières distribuent/diffusent les données aux différentes opérations du graphe.

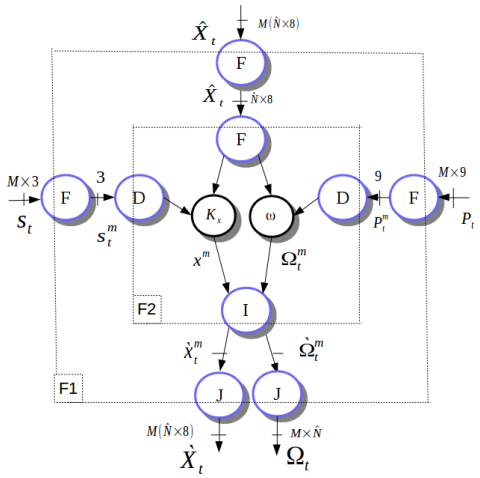


FIGURE 4.18: *Graphe final du FB4 factorisé à l'aide des sommets frontières de factorisation*

Bloc d'initialisation (FB5) La figure 4.19 montre le graphe orienté acyclique du bloc d'initialisation FB5. Le graphe de la figure 4.19-a montre le graphe simplifié du bloc FB4. Ce bloc reçoit en entrée l'état des particules à jour S_t et les nouveaux amers déterminés dans FB2. La sortie de ce bloc est l'ensemble des amers initialisés par la méthode de l'inverse de profondeur. La figure 4.19-c montre le graphe décomposé du bloc en sous opérations de calcul dénommé fb_5. Chaque opération initialise les nouveaux amers pour une seule particule. Elle calcul les paramètres de l'inverse de profondeur en se basant sur la position des particules à jour. L'opération d'initialisation est indépendante et chaque unité de calcul peut calculer les paramètres initiaux de façon indépendante. La figure 4.19-c représente le graphe factorisé de l'opération fb_5.

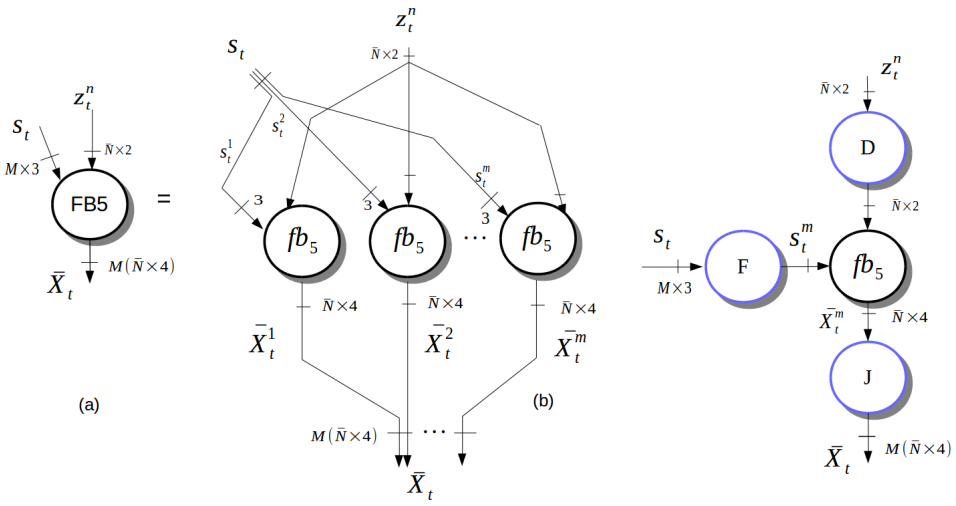


FIGURE 4.19: *Graphe du bloc d'initialisation FB5 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontières de factorisation*

La figure 4.20 représente le graphe décomposé de l'opération fb_5. Ce graphe est décomposé en sous-graphes, chacun calcule les paramètres initiaux des amers pour une seule particule. Ce graphe peut être résumé et factorisé en seulement quatre opérations qui calculent l'orientation θ, l'élévation ϕ , la position de la camera x_c, y_c et la position 3D des amers dans le repère du monde (x_c, y_c, z_c). Le graphe est factorisé par trois sommets frontières à savoir : F qui distribue l'observation z_t aux différents sous-graphes, D qui diffuse l'état des particules au différents graphes et J qui regroupe les paramètres calculés pour reconstruire le vecteur $\overline{X}_t^m$ qui représente l'ensemble des amers initialisés pour la m-ième particule. La figure 4.21 correspond au graphe total factorisé du bloc FB4.

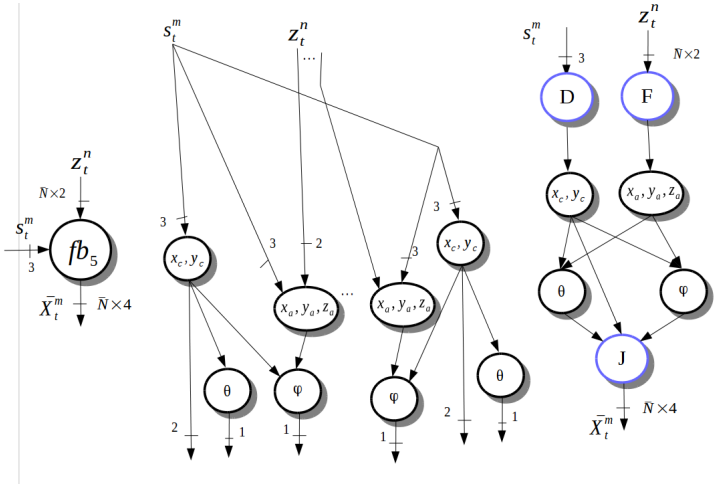


Figure 4.20: *Graphe du sous-bloc d'initialisation FB5 : (a) graphe composé, (b) graphe décomposé, (c) graphe factorisé à l'aide des sommets frontière de factorisation*

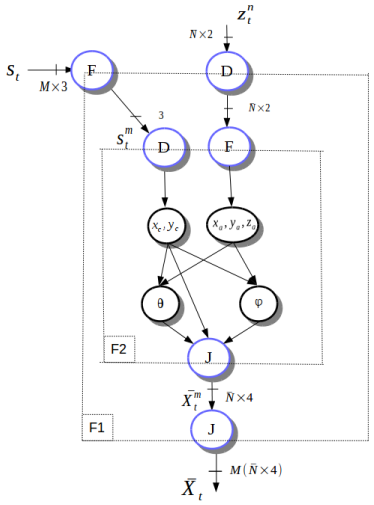


Figure 4.21: *Graphe final du FB5 factorisé à l'aide des sommets frontières de factorisation*

Bloc de rééchantillonnage La figure 4.22 représente le graphe global factorisé du bloc de rééchantillonnage FB6. Ce bloc échantillonne les particules pour tenir en compte les observations. Pour ce faire, il prend en entrée l'ensemble des probabilités Ω_t calculées par rapport à chaque amer apparié dans le bloc FB4 et l'ensemble $\hat{S}_t$ des particules à jour calculées dans FB3. La sortie de ce bloc est l'ensemble des nouvelles particules $\overline{S}_t$ qui sera utilisé pour des prochaines itérations de l'algorithme. Ce bloc contient certaines dépendances de données internes qui exigent des modifications algorithmiques

pour une implémentation parallèle telle que la somme cumulative et la stratification aléatoire pour générer l'intervalle de sélection.

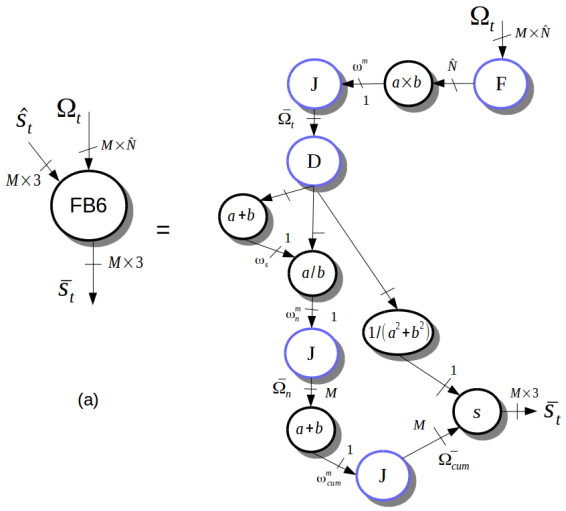


Figure 4.22: *Graphe final du bloc de rééchantillonnage FB6 factorisé à laide des sommets frontières de factorisation*

4.6.2.2 Définition du graphe d'architecture

La figure 4.23 représente le graphe de l'architecture que nous avons adoptée lors de l'implémentation. Le graphe simplifie la structure d'une architecture hétérogène intégrant un CPU multi-cœurs et d'un processeur GPU. Les cœurs du CPU sont modélisés par un opérateur "OPR", nous donnons l'exemple d'un Quad-Cœurs CPU. Chaque opérateur communique avec sa mémoire cache modélisée par le sommet "Rc" via un bus/mus/demux modélisé par "b". Chaque opérateur a un accès à la mémoire RAM via des communicateurs et des bus/mux/démux. L'architecture massive du GPU est modélisée par un ensemble des opérateurs qui communiquent avec une mémoire locale "R_{loc}" via un bus/mux/démux. La mémoire locale communique avec la mémoire globale du GPU modélisé par un R_D(mémoire périphérique/Device). Les deux processeurs, CPU et GPU communiquent entre eux via un bus PCIe expresse modélisé par un PCIe. Les opérateurs du CPU et la RAM ont accès au bus PCIe via une mémoire d'entrée/sortie modélisée par le sommet SAM. En réalité, l'architecture du GPU n'est pas évidente à modéliser par un simple graphe d'architecture. En effet, l'architecture interne du GPU varie en fonction du langage de programmation. Chaque langage utilisé a sa propre vue de l'architecture interne du GPU. Comme nous l'avons mentionné auparavant, nous utiliserons deux langages de programmation principaux : OpenGL et OpenCL. Pour cela, nous définissons le graphe d'architecture qui correspond chacun à un langage spécifique.

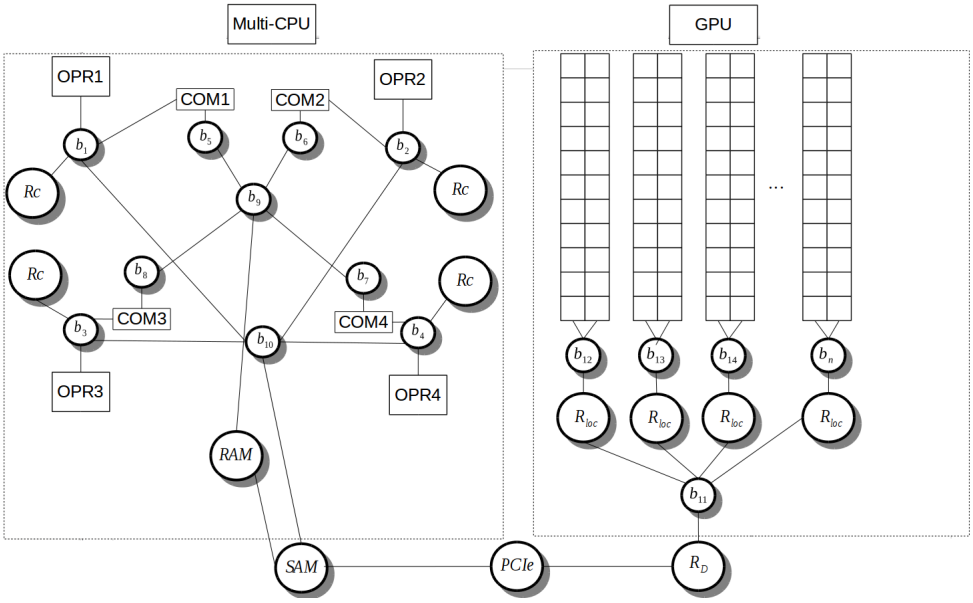


Figure 4.23: *Graphe de l'architecture hétérogène multi-CPU/GPU*

La figure 4.24 représente le graphe d'architecture du GPU tel qu'il est vu par OpenCL. Les opérateurs de calcul sont divisés en des groupes appelés groupes de travail. Chaque groupe de travail à sont tour, est divisé en des sous-groupes de travail. Chaque sous-groupe de travail a sa propre mémoire privée modélisée par le sommet R_{pr}. Cette mémoire privée n'est accessible que par ce sous-groupe de travail. Toutes les variables qui n'ont pas été déclarées avec un spécificateur d'adresse sont considérées privées est déclarées dans cette mémoire. Les sous-groupes de travail ont accès à une mémoire locale modélisée par R_{loc} via un bus/mux/démux. Cette mémoire n'est accessible que par les sous-groupes dans le même groupe de travail. Toutes les variables déclarées dans la fonction du kernel avec un spécificateur d'adress "$\_\_local$" sont considérées comme des variables locales et sauvegardée dans R_{loc}. Tous les opérateurs de calcul que ce soit dans un même groupe/sous groupe de travail ou non, ont accès à une mémoire globale modélisée par R_{dev}. Tout transfère de données entre CPU et GPU est fait à travers cette mémoire. Cette mémoire contient un espace d'adresse réservé aux constantes déclarées au sein d'un kernel, cette espace appelé : mémoire constante.

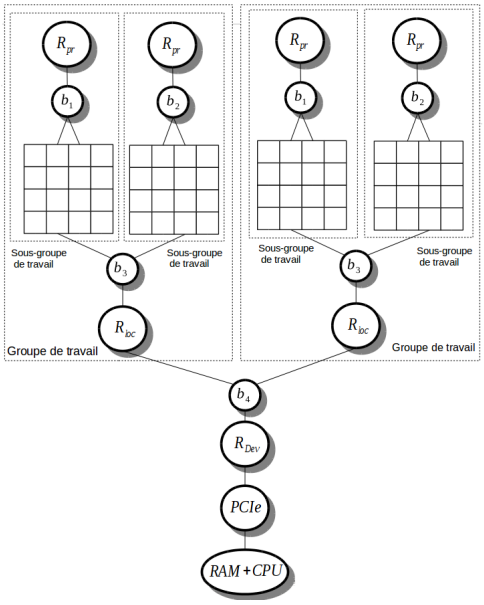


Figure 4.24: *Graphe de l'architecture GPU vue par OpenCL*

Le graphe de l'architecture tel qu'il est vu par OpenGL dépend en fait de l'architecture du GPU. En effet, il existe deux types d'architecture de GPU : unifiée et non-unifiée.

La figure 4.25 représente le graphe d'architecture non unifiée du GPU. Il existe deux opérateurs de calcul : les opérateurs des vertices et les opérateurs des fragments. Les opérateurs des vertices traitent les données issues de la mémoire des attributs modélisés par R_{vbuff}. Ces opérateurs sont dédiés au calcul géométriques et au traitement des vertices de la primitive géométrique. Les opérateurs des fragments traitent les données issues de la mémoire texture modélisée par le sommet R_{Tex}. Ces opérateurs sont dédiés au calcul pixelisé et au traitement des couleurs des fragments (pixels). Les deux ensembles d'opérateurs (vertices et fragments) sont séparés par une phase de pixellisation préalablement fixée et non accessible par le programmateur. Cette phase transforme la primitive définie par des vertices par les opérateurs de vertices en un ensemble des fragments qui seront traités par les opérateurs des fragments. Le graphe d'architecture fait apparaître trois types de mémoire principaux. La mémoire R_{vbuff} est une zone mémoire spécifique aux attributs des vertices de la primitive. Cette mémoire est accessible par le CPU pour charger et stocker les données des attributs des vertices et n'est accessible que par les opérateurs de vertices au sein du GPU. La limitation de cette mémoire est le transfèrt qui n'est pas réversible. Autrement dit le CPU ne peut pas lire les données de cette mémoire après une écriture (pas de lecture aléatoire). La mémoire R_{Tex} est une zone mémoire spécifique aux données textures et couleurs des pixels. Cette mémoire est de type RAM (mémoire à accès aléatoire). Le CPU peut librement avoir accès à cette mémoire que ce soit en écriture ou en lecture mais elle n'est accessible que par les opérateurs des fragments au sein du GPU. La mémoire R_{FB}, par défaut, est une mémoire à accès aléatoire en écriture seule par le CPU et seulement par les opérateurs de fragments au sein du GPU. Cette zone mémoire peut être accessible par lecture aussi en utilisant le frame buffer object FBO. Il est utilisée principalement pour sauvegarder les résultats de calcul et plus particulièrement la sortie des kernel des opérateurs des fragments et pour permettre la réutilisation des résultats en entrée du kernel.

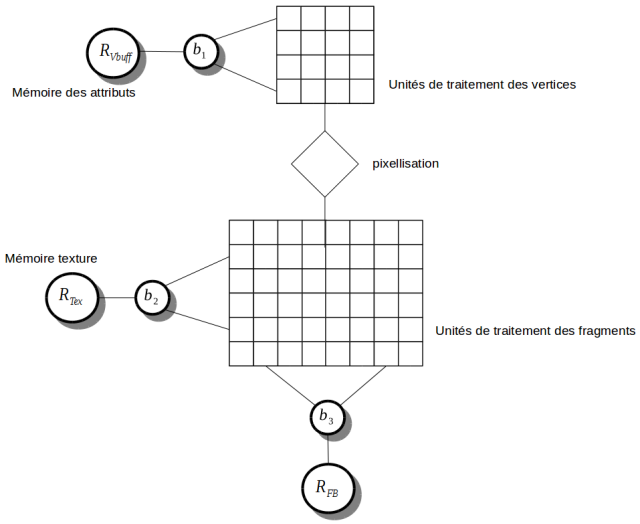


Figure 4.25: *Graphe de l'architecture du GPU vue par OpenGL*

La figure 4.26 représente le graphe de l'architecture correspondant à l'architecture unifiée du GPU. L'architecture unifiée représente une structure où tous les stages de shader appelés aussi pipelines, ont la même caractéristique. Au lieu d'avoir deux ensembles d'opérateurs séparés (opérateurs des vertices et des fragments) comme dans le modèle précédent, ces opérateurs sont unifiés en un seul groupe d'opérateurs. Ces opérateurs peuvent soit traiter les vertices d'une primitive soit ses pixels. Ils peuvent aussi avoir accès aux différentes mémoires disponibles, la mémoire des attributs R_{vbuff} ou bien la mémoire texture R_{Tex}. Une telle architecture permet l'utilisation flexible du GPU. Autrement dit, dans le cas où on a une quantité énorme des données des attributs à traiter, on peut allouer plus d'opérateurs pour cette tâche. Au contraire, dans le cas où on a moins de données des attributs et plus de données pixels, on peut allouer plus d'opérateurs pour traiter les données pixels. L'allocation est adaptée en fonction du type de calcul qu'on souhaite réaliser.

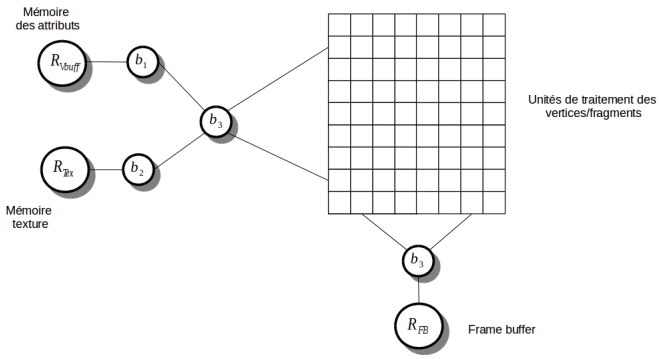


Figure 4.26: *Graphe de l'architecture du GPU vue par OpenGL*

4.6.2.3 Définition du graphe d'implémentation

Les systèmes à architectures hétérogènes (HSA), que nous étudions dans cette section, permettent l'utilisation du GPU et du CPU simultanément pour améliorer le temps d'exécution global. On se basant sur le partitionnement en blocs fonctionnels de l'algorithme, le graphe d'algorithme et de l'architecture, nous définissons le graphe d'implémentation du FastSLAM2.0. Les blocs fonctionnels qui nécessitent un temps d'exécution significatif sont parallélisés sur le GPU tandis que les traitements

séquentiels sont implémentés sur le CPU.

Pour simplifier les transformations du graphe d'algorithme en fonction du graphe d'architecture, nous donnons dans la figure 4.27 le graphe de l'algorithme simplifié. Le graphe montre les entrées/sorties principales ainsi que les blocs fonctionnels du FastSLAM2.0. L'algorithme interagit de manière itérative avec deux capteurs à savoir : les encodeurs et la caméra qui constituent les entrées du système. La sortie de l'algorithme est l'état des particules $\overline{S}_t$ à l'instant t et respectivement l'ensemble des amers corrigés $\grave{X}_t$ et initialisés $\bar{X}_t$.

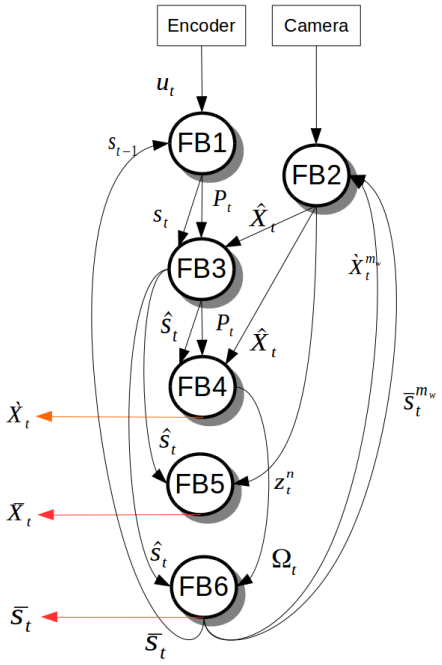


Figure 4.27: *Graphe d'algorithme global simplifié à l'aide des sous-graphes composés des bloc fonctionnels*

La figure 4.28 représente le mapping du graphe de l'algorithme sur le graphe de l'architecture. Les flèches en rouge représentent l'entrée du bloc ou de l'opération et celles en bleu représentent la sortie du bloc ou de l'opération. Les flèches en noir représentent l'ordonnancement des opérations et des blocs du systèmes. Les données des capteurs (données des encodeurs et images) sont étiquetées temporellement par l'opération "TS" implémentée dans le CPU.

Le graphe de l'algorithme du bloc FB1 (figure 4.9) fait apparaître l'opération intérimaire "*cd*" qui génère une distribution aléatoire des données odométriques. Une partie de cette opération qui concerne la génération du nombre aléatoire RDN est implémentée sur CPU vu que la génération séquentielle des nombres aléatoires est plus efficace pour assurer une localisation correcte du robot. Les nombres aléatoires sont stockés dans la mémoire du CPU R_{CPU} puis ils sont transférés à travers un bloc DMA vers la mémoire du GPU R_{Dev}. Les opérations indépendantes $fb_1^1 \cdots fb_1^M$ du blocs FB1 (figure 4.9) sont parallélisées sur le GPU. Chaque opérateur du GPU exécute les équations du modèle d'évolution (la zone frontière F3 dans la figure 4.11) pour calculer la position future des particules.

Le graphe orienté du bloc FB2 (figure 4.12) ne présente pas des opérations de traitement au niveau des particules. Ceci est dû au fait que la phase de mise en correspondance n'est réalisée que pour la particule qui a le plus grand poids $\bar{s}_t^{m_w}$. L'opération $FAST$ est séquentielle et déjà optimisée en utilisant un processus de "machine learning". Les deux opérations $Proj$ et $ZMSSD$ peuvent être parallélisées au niveau des amers de la carte. En effet, à chaque itération de l'algorithme, peu d'observations sont détectées à cause des contraintes sur l'incertitude des particules, donc peu

d'opérateurs sont suffisants pour traiter la parallélisation. Par conséquent, l'ensemble du bloc FB2 est implémenté sur le CPU parce qu'il est le plus adapté à ce type de traitement. L'implémentation de ce bloc sur le GPU va réduire certainement les performances d'exécution. En effet, plusieurs opérateurs ne seront pas utilisés car le calcul dans ce bloc n'est pas fortement parallèle. Dans ce cas, le temps de transfert sera plus important par rapport au temps de calcul, ce qui diminue les performance temps-réel. Nous savons que pour optimiser la phase de rééchantillonnage, un arbre binaire est utilisé pour organiser la carte de chaque particule. L'implémentation de l'arbre nécessite une allocation dynamique de la mémoire qui n'est pas supportée ni par OpenCL ni par OpenGL. Donc, l'implémentation de l'arbre binaire modélisée par l'opération CAB ne peut être que sur le CPU, ce qui impose un transfert de données du CPU vers le GPU à chaque itération de l'algorithme.

Le bloc FB3 est implémenté sur le GPU. Les opération de calcul $fb_3^1 \cdots fb_3^M$ sont parallélisées sur le GPU (figure 4.13). Chaque opérateur du GPU calcule la nouvelle distribution probabiliste (La zone F2 dans la figure 4.15) pour corriger la position des particules. La parallélisation du calcul au niveau des amers pour la construction de la distribution probabiliste n'est pas possible car le graphe défactorisé (figure 4.14-c) présente une dépendance de données. La distribution à l'instant Δ_t est calculée à partir de celle à l'instant Δ_{t-1}. Donc, la séquence des graphes (figure 4.14-c) est exécutée séquentiellement l'une après l'autre par les opérateurs.

Les opérations $fb_4^1 \cdots fb_4^M$ du bloc FB4 sont parallélisées sur le GPU (figure 4.16). Chaque opérateur du GPU corrige les paramètres des amers (la zone F2 dans la figure 4.18) pour une seule particule indépendamment des autres. Théoriquement parlant, la parallélisation du calcul au niveau des amers (séquence des sous-graphes dans la figure 4.17) est possible, car le calcul ne présente aucune dépendance de données. Chaque amer peut être corrigé indépendamment des autres amers (chaque sous-graphe corrige un amer). Malheureusement, en pratique cela n'est pas possible dû aux contraintes matérielles des GPUs. En effet, tous les GPUs modernes sont capables de lancer un seul niveau de parallélisation. Autrement dit, la parallélisation au niveau des boucles intermédiaires (le parallélisme imbriqué) n'est pas possible. En conséquence, nous choisissons de paralléliser le calcul au niveau du GPU qui présente un parallélisme massive (niveau des particules, opérations $fb_4^1 \cdots fb_4^M$). Le kernel de chaque opérateur implémente donc les deux opérations ("K_x","ω") pour mettre à jour les amers.

Les opérations $fb_5^1 \cdots fb_5^M$ du bloc FB5 sont parallélisées sur le GPU (figure 4.19). Chaque opérateur initialise un seul amer (la zone F2 de la figure 4.21) de manière indépendante. Pour la même raison que précédemment, la parallélisation au niveau des amers (séquence des sous-graphes dans la figure 4.20) n'est pas réalisable sur le GPU. Le calcul est donc parallélisé d'une manière massive (opérations $fb_5^1 \cdots fb_5^M$).

L'opération du bloc FB6 peut être divisée en deux sous-tâches intermédiaires à savoir la rééchantillonnage des particules et le rééchantillonnage de la carte. Le rééchantillonnage des particules est parallélisé sur le GPU (les opérations du graphe de la figure 4.22). Le rééchantillonnage de la carte est réalisé sur le CPU vu que l'arbre binaire est implémenté sur le CPU.

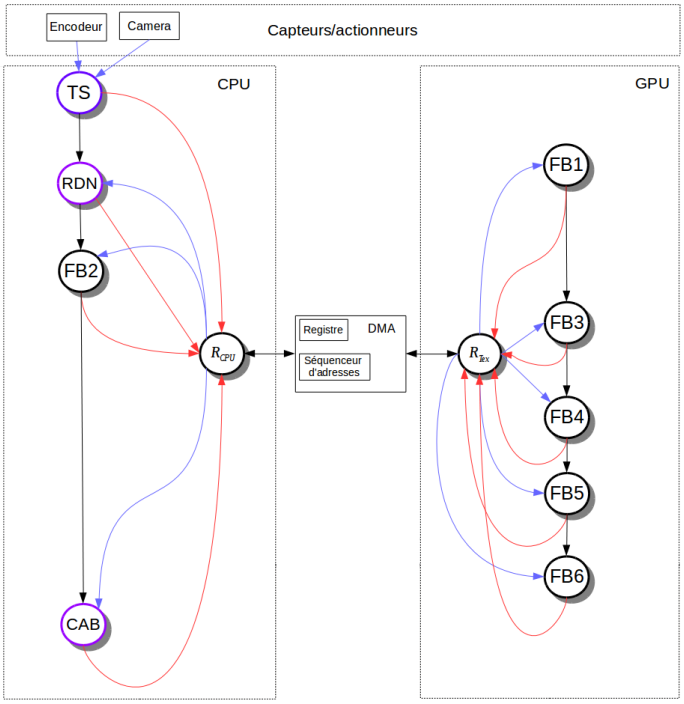


FIGURE 4.28: *Graphe d'implémentation simplifié*

Le goulot d'étranglement d'une implémentation hétérogène réside dans deux facteurs principaux à savoir : le transfert de données entre les différents opérateurs et les accès mémoires. Ceci peut réduire considérablement les performances qui peuvent être obtenues à partir d'un système à architecture homogène. Afin de résoudre ce problème, nous avons pris en considération ces deux facteurs dans l'élaboration du graphe d'implémentation.

Le graphe d'implémentation met en évidence un mécanisme important : tout transfert de données entre CPU et GPU est réalisé par un opérateur DMA. Dans un transfert DMA asynchrone, le CPU intervient juste pour charger les données dans une zone mémoire directement accessible par le GPU et non pas dans toute la période de transfert. En effet, le contrôleur mémoire s'occupe du transfert de données de la zone mémoire vers la mémoire du GPU en réalisant un transfert DMA asynchrone sans perte de cycles du CPU. Sans un tel transfert, le CPU sera occupé pendant toute la période de l'opération de transfert. Donc le CPU ne sera pas disponible pour exécuter d'autre tâches. Comme il est montré dans le graphe d'implémentation, le CPU est chargé de faire une synchronisation de données "TS", une génération des nombres aléatoires "RDN", exécution du bloc de traitement d'images FB2 et classification des amers dans l'arbre binaire CAB. En utilisant un DMA, le CPU initialise juste le transfert et se libère pour exécuter les tâches qui lui sont confiées. Ceci est un avantage permettant une exécution parallèle des deux processeurs CPU et GPU. La figure 4.29 montre l'amélioration de la bande passante du transfert mémoire par DMA en utilisant des données génériques. Le transfère par DMA est plus rapide et efficace par rapport à un transfert ordinaire (1.2 GB/s vs 640 MB/s).

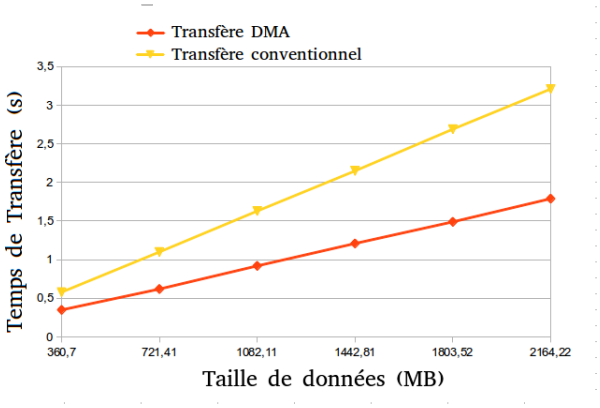


FIGURE 4.29: *Transfert de données asynchrone (DMA) Vs un transfert ordinaire*

L'accès mémoire au niveau du GPU est optimisé en utilisant la mémoire texture qui renforce d'avantage la puissance de calcul parallèle des GPU. Le modèle d'accès à la mémoire texture est de type "localité spatiale". Autrement dit, les données de la mémoire texture sont lues par des adresses qui sont proches l'une des autres, ceci est illustré dans la figure 4.30. Les tâches de 0 jusqu'à 3 représente le kernel du GPU et le tableau représente la mémoire texture du GPU. Toutes les tâches ont accès aux données localisées dans des adresses proches les unes aux autres. La mémoire texture a un système de cache optimisé pour ce type de modèle d'accès mémoire. Ceci augmente d'avantage la bande passante des accès mémoires.

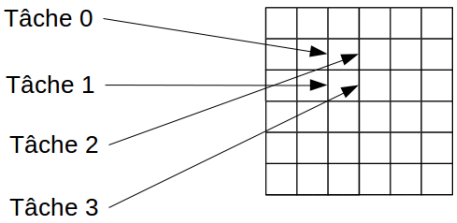


FIGURE 4.30: *Modèle de localité spatiale pour l'accès mémoire*

4.6.2.4 Transformation du graphe d'implémentation

Transformation par OpenGL

Bloc de prédiction FB1 La position des particules $s_t^m = (s_x, s_y, s_\phi)$ est transférée dans la mémoire globale du GPU et stockée dans une texture. Chaque texel de la mémoire texture contient la position d'une particule. Dans notre implémentation, on a préféré générer les nombres aléatoires sur le CPU pour chaque particule et ensuite les transférer dans une autre texture dans la mémoire du GPU. Cette méthode assure la génération précise d'un ensemble de particules et éviter le problème de la mauvaise dispersion des particules. Une solution éventuelle, c'est d'implémenter des techniques de génération parallèle des nombres aléatoires comme décrit dans [134], mais cela serait au détriment de la consistance des résultats de localisation. Une mauvaise distribution des particules influence les résultats de localisation plus particulièrement pour un environnement à grande échelle.

Plusieurs données odométriques sont acquises à chaque itération de l'algorithme. Par conséquent, la position des particules doit être prédite à chaque acquisition pour reconstruire proprement la trajectoire. Pour implémenter cette procédure, on utilise la technique de "multipass". Cette technique consiste à utiliser les textures pour l'écriture et la lecture à la fois. La texture est utilisée comme cible pour stocker les résultats pour une seule opération de prédiction, puis elle est réutilisée directement comme entrée pour l'opération suivante. Le problème qui s'impose ici est que les textures en OpenGL sont soit à lecture seule où à écriture seule. Pour dépasser cette limite, on utilise trois textures à la fois pour les opérations de prédiction. Une texture inchangée à lecture seule est utilisée pour stocker les données odométriques. Les deux autres textures sont attachées à un frame buffer (FBO) : une texture à lecture seule s^t_{old} est utilisée pour stocker la position des particules et une texture à écriture seule s^t_{new} est utilisée pour stocker les résultats de prédiction. Après chaque opération de prédiction, on échange l'option de ces deux textures, la texture à lecture-seule devient à écriture-seule et la texture à écriture-seule devient lecture-seule et ainsi de suite. (Voir Annexe 7.4, section 7.4.5.1 pour le détail de l'implémentation OpenGL du bloc FB1).

Bloc de mise à jour de la position des particules FB3 La procédure du calcul de la nouvelle distribution probabiliste et la mise à jour de la position des particules est réalisé en utilisant la technique de "multipass". Chaque "pass" calcule la moyenne de la distribution probabiliste $\mu^m_t = (x, y, \theta)$ et sa matrice de covariance Σ^m_t de dimension 3x3 en parallèle pour chaque particule. Une seule texture n'est pas suffisante pour stocker les résultats de calcul d'un pass. Pour dépasser cette limite, on utilise plusieurs textures dans un pass afin de stocker tous les paramètres de la distribution probabiliste. Pour cela, on a besoin de quatre textures pour stocker la moyenne et la matrice de covariance pour chaque particule. En conséquence, douze textures seront utilisées pour la procédure de mise à jour distribuées comme suite :

- Quatre textures à lecture seule inchangées utilisée pour stocker les paramètres des amers appariés $(u, v, x_0, y_0, \rho, \theta, \phi, C_t)$.
- Huit textures attachées au frame buffer FBO : dont quatre textures à lecture-seule sont utilisées pour stocker les valeurs initiales de la distribution $\left(\mu^{m,t-1}_{old}, \Sigma^{m,t-1}_{old}\right)$, et quatre textures à écriture seule sont utilisées pour stocker les résultats d'un pass $\left(\mu^{m,t-1}_{new}, \Sigma^{m,t-1}_{new}\right)$.

Il est important de noter qu'on pourrait allouer un nombre de textures suffisant pour stocker tous les amers, ce qui permettrait la construction de la distribution probabiliste en un seul pass. Ceci va certainement réduire le taux de transfert de données et améliore les performances de la parallélisation. Pourtant, une allocation inutile de la mémoire doit être évitée. Ces allocations, peuvent augmenter le besoin en termes de mémoire au détriment d'autre ressources qui en a besoin. (Voir Annexe 7.4, section 7.4.5.2 pour le détail de l'implémentation OpenGL du bloc FB3).

Bloc d'estimation FB4 Chaque pass d'exécution corrige les paramètres d'un seul amer apparié pour toutes les particules en parallèle. Pour cela, on a besoin de douze textures pour paralléliser le bloc FB4 sur le GPU. Huit textures à lecture seule sont utilisées pour stocker les paramètres de l'inverse de profondeur d'un amer $(u, v, x_0, y_0, \rho, \theta, \phi, C_t)$ et l'état courant des particules (s^m_t, P^m_t). Les paramètres de l'amer corrigés par le filtre de Kalman sont stockés dans quatre textures à écriture seule attachées au FBO. Le shader du fragment exécute les équations du fitlre de Kalman pour mettre à jour les paramètres de l'amer et calculer la probabilité correspondante. (Voir Annexe 7.4, section 7.4.5.3 pour le détail de l'implémentation OpenGL du bloc FB4).

Bloc d'initialisation FB5 Pour une initialisation par inverse de profondeur, cinq paramètres doivent être calculés par rapport à chaque particule $(x_i, y_i, \theta_i, \phi_i, \rho)$. Le paramètre de l'inverse de profondeur ρ peut être initialisé par une valeur constante lors de la première observation $\rho = 0.25$, comme mentionné dans [47], en conséquence, il est initialisé dans le CPU. Les paramètre restant $(x_i, y_i, \theta_i, \phi_i)$ sont calculés par le GPU. Pour cela, on transfère la position des particules et la position du nouvel amer apparié vers la mémoire texture du GPU. Deux textures seront nécessaires : une texture à lecture-seule de stockage interne type RGBA utilisée pour stocker la position des particules (une seule particule par texel : $R = s_x^m$,$G = s_y^m$,$B = s_\theta^m$,$A = 0$), et une texture à écriture seule attachée au FBO pour stocker les paramètres initiaux de l'amer (un seul amer par texel $R = x_i$,$G = y_i$,$B = \theta_i$,$A = \phi_i$). La position de l'amer dans l'image (u_i, v_i) est chargée vers le shader du fragment utilisant une valeur uniforme. (Voir Annexe 7.4, section 7.4.5.4 pour le détail de l'implémentation OpenGL du bloc FB5).

Bloc de rééchantillonnage FB6 L'implémentation de la phase de calcul du poids total est similaire à celle du FB1. Trois textures sont utilisées : une texture à lecture-seule contenant les probabilités calculées dans le bloc FB4, et deux textures à écriture-seule attachées au FBO et contenant respectivement l'ancienne valeur du poids w_{old} et la nouvelle valeur du poids w_{new}. Pour calculer la nouvelle valeur du poids par rapport aux différentes probabilités calculées, on échange l'option des deux textures w_{old} et w_{new} de manière récursive jusqu'à la mise à jour complète du poids des particules.

Le poids normalisé est calculé dans un seul pass. Pour ce faire, deux textures sont utilisées, la première contient les valeurs mises à jour du poids et la deuxième stocke le poids normalisé de chaque particule. La somme des poids est chargée au shader du fragment à travers une valeur uniforme.

La somme des poids est une opération de réduction qui peut être implémentée en parallèle sur le GPU en réduisant récursivement une texture de taille MxM à une texture scalaire de taille 1x1. L'algorithme réduit de manière récursive la texture de sortie en calculant la somme locale de chaque région de taille 2x2 et en écrivant la valeur dans la zone de sortie correspondante. La figure 4.31 illustre la première opération de réduction pour calculer la somme des poids stockés dans une texture de taille 4x4. La texture à gauche montre la texture d'entrée contenant les valeurs des poids. La région grise regroupe les sous-valeurs qui seront utilisées pour calculer la première somme locale des poids. La texture à droite montre le résultat de la première opération de réduction. Chaque texel de sortie contient la somme locale de la région correspondante de taille 2x2 dans la texture d'entrée.

Le calcul du nombre effectif des particules avant rééchantillonnage est implémenté en parallèle de la même manière en utilisant des opérations de réduction. Pour ce faire, trois textures seront utilisées : une texture contenant les valeurs des poids des particules w et deux textures temporaires réalisant l'opération de réduction elle même.

4	5	63	10
2	6	7	3
3	22	32	2
36	1	4	12

17	83
62	50

Figure 4.31: *Implémentation de la somme des poids sur GPU. Ceci est répété de manière récursive jusqu'à ce que la texture de taille MxM est réduite à une texture scalaire 1 x 1*

Transformation par OpenCL

Bloc de prédiction (FB1) Au début de l'algorithme, l'état de chaque particule est initialisé dans un buffer dans la mémoire globale du GPU. Afin de permettre une implémentation efficace, nous avons modélisé le bruit provenant des données odométriques par une distribution Gaussienne qui a la forme suivante :

$$n_l = n_l + 4 * \left(\sqrt{(-2.0 * \log(\epsilon_1))} * cos(2 * \pi * \epsilon_2)\right) * \epsilon_g \tag{4.2}$$

$$n_r = n_r + 4 * \left(\sqrt{(-2.0 * \log(\epsilon_3))} * cos(2 * \pi * \epsilon_4)\right) * \epsilon_g \tag{4.3}$$

avec n_l et n_r sont respectivement les données des encodeurs gauche et droite, $(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4)$ sont des valeurs aléatoire comprises entre [0,1], ϵ_g est l'erreur de glissement. L'erreur de glissement reflète le glissement des roues qui génère un écart entre la mesure et la vraie distance parcourue. Cette erreur peut avoir une petite valeur si le robot se déplace avec une vitesse réduite. Afin d'implémenter la génération des nombres aléatoires sur le GPU en utilisant OpenCL, on note que chaque particule est traitée par une tâche, et que chacune des tâches a son propre identificateur. Par conséquent au lieu de générer les valeurs $(\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4)$ pour chaque particule, on ne génère qu'une petite quantité des nombres aléatoires dans le CPU (60 nombres dans notre implémentation) et on les transfère vers le GPU. On les combine après avec l'identificateur de chaque tâche pour générer les nombres aléatoires privés spécifique pour chaque particule. Une fois les nombres aléatoires sont générés au sein du kernel, la nouvelle position des particules est calculée et stockée dans le même buffer dans la mémoire du GPU. Nous avons vu dans l'implémentation de ce bloc par OpenGL qu'on génère tous les nombres aléatoires dans le CPU et on les transfère tous vers le GPU. Malheureusement, la méthode optimisée de génération de bruit utilisée par OpenCL n'est pas possible par OpenGL car les valeurs stockées dans les textures sont accessibles par les coordonnées des textures. Ces cordonnées ne sont pas contrôlées par le programmeur, elles sont générées automatiquement par la phase de pixellisation (rasterizer) par interpolation.

Bloc de mise à jour de la position des particules (FB3) Dans l'implémentation OpenCL, on transfère les paramètres de tous les amers appariés et l'état des particules en un seul coup. Ce qui permet la construction de la nouvelle distribution probabiliste au sein du kernel. Le nouvelle distribution est stockées dans la mémoire du GPU puisqu'il s'agit d'une mémoire à écriture/lecture en même temps. Nous avons vu que dans l'implémentation OpenGL, un seul pass calcule la distribution probabiliste par rapport à un amer apparié. Pour complètement construire cette distribution par rapport à tous les amers apparié, dans chaque itération il faut transférer les paramètres de l'amer mis en correspondance du CPU vers la mémoire texture du GPU. Contrairement à l'implémentation OpenGL, une seule exécution du kernel construit complètement cette distribution car tous les amers matchés ont été transférés dans la mémoire global du GPU avant le lancement du kernel. Cette implémentation n'est pas permise sous OpenGL, car les paramètres d'un seul amer mis en correspondance occupe 4 textures. Plusieurs amers nécessitent plusieurs textures, ce qui est impossible pour toutes les implémentations matérielles du OpenGL existantes. Le nombre maximum des textures qui peuvent être allouées même pour les GPUs NVIDIA de haute-gamme est 32. Donc l'implémentation OpenCL permet de réduire considérablement le taux de transfert par rapport à OpenGL.

Bloc d'estimation (FB4) L'implémentation de ce bloc en OpenCL augmente les performances d'exécution par rapport à la version OpenGL. On rappelle que dans le bloc FB3 (mise à jour de la position des particules), tous les paramètres des amers appariés sont transférés dans la mémoire globale du GPU. Par conséquent, le bloc FB4 est exécuté sans aucun transfert au préalable. On lance directement le kernel pour mettre à jour la position de tous les amers appariés par les équations de Kalman en parallèle pour toutes les particules. Contrairement à l'implémentation OpenGL, le transfert de données avant chaque exécution du shader du fragment est inévitable. Puisque la mémoire de texture est limitée en termes de ressource et ne permet pas de stocker les paramètres de tous les amers appariés, on ne peut pas allouer plus que 32 textures pour stocker ces valeurs. Ce qui signifie que l'implémentation OpenCL de ce bloc réduit considérablement le taux de transfert.

Bloc d'initialisation FB5 L'implémentation de ce bloc en OpenCL est exécutée sans aucun transfert de données au préalable puisque les paramètres des amers non appariés et leurs observations correspondantes sont déjà stockées dans la mémoire globale du GPU. L'exécution du kernel calcule les paramètres de la paramétrisation par inverse de profondeur de chaque nouvel amer pour toutes les particules en parallèle. Nous avons vu précédemment que l'arbre binaire est utilisé pour la gestion de la carte et que cet arbre est implémenté sur le CPU. En conséquence, tous les amers appariés qui ont été mis à jour dans le bloc FB4 et tous les nouveaux amers qui ont été initialisés dans le bloc FB5 sont transférés vers le CPU afin de permettre leur classification et arrangement dans l'arbre binaire.

Bloc de rééchantillonnage FB6 Pour implémenter la sommation des poids en parallèle en utilisant OpenCL, la somme est divisé sur des tâches intermédiaires. Chaque sous-tâche est exécutée par un kernel et calcule la somme locale d'un ensemble de poids des particules. Le poids total est la somme des résultats des sommes locales de chaque sous tâche. Les sous-tâches sont exécutées en parallèle. Le calcul de la somme totale par OpenCL est illustré dans la Figure 4.32. Le calcul du nombre effectif des particules est implémenté avec la même manière. Les sous-tâches calculent la somme carré des poids et la valeur finale est calculée par le CPU.

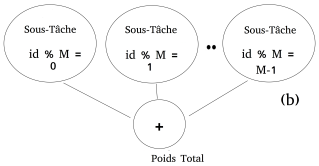


Figure 4.32: *Calcul de la somme des poids*

4.6.3 Résultats expérimentaux

Dans cette section nous allons analyser les temps d'exécution de l'implémentation hétérogène du FastSLAM2.0 sur les différentes architectures embarquées. Les résultats expérimentaux sont obtenus en se basant sur la méthodologie d'évaluation discuté auparavant. Tous d'abord, les temps d'exécutions sont obtenus en évaluant l'algorithme avec le jeu de données réelles de Rawseeds. Ces temps sont calculés en se basant sur les équations définis dans le chapitre 2, section 2.3.5. Chaque bloc fonctionnel de l'algorithme est évalué séparément afin de synthétiser le temps d'exécution global de l'algorithme. Les évaluations sont faites en se basant sur un certains nombres de seuils définis à l'avance afin de prendre en compte les paramètres de dépendance de chaque bloc fonctionnel. Nous définissons alors les seuils suivants :

— Données odométriques : le nombre de données acquises à chaque itération est détermine en fonction de la fréquence des encodeurs.
— La taille de l'image : 320×240 pixels.
— La taille du descripteur : 16×16 pixels.
— Le nombre maximum d'amers dans la carte de chaque particule : 500
— Le nombre maximum d'observation : 60.
— Le nombre maximum des amers appariés : 40.
— Le nombre maximum des amers en cours d'initialisation : 40
— Plusieurs nombre de particules seront utilisées.

Nous évaluons dans cette section l'algorithme FatSLAM2.0 à grande échelle.,Pour cette raison les seuils définis ne sont pas les mêmes que ceux définis lors de l'évaluation de l'algorithme sur une courte trajectoire. Dans cette version, nous constituons la trajectoire en exploitant toutes les données odométriques acquises durant le test. On ne peut pas définir un seuil pour ce paramètre puisque toutes ces données seront nécessaires pour une reconstruction robuste de la trajectoire odométrique sur une longue trajectoire. Le nombre des données odométriques est lié donc à la fréquence des encodeurs utilisés dans le jeu Rawseeds. La taille de l'image et du descripteur restent inchangé. Le nombre maximum des amers dans la carte de chaque particule est fixé pour 500. Ce nombre est diminué quand un grand énorme de particules est utilisé pour ne pas épuiser les ressources mémoire qui sont par nature très limitées dans une architecture embarquée. Le nombre de particules est variable pendant l'expérimentation afin de tester différentes possibilités d'implémentation en fonction des ressources de calcul de l'architecture cible. Le nombre maximum d'observations, des amers appariés et des amers en cours d'initialisation sont fixé respectivement à 60, 40 et40.

Durant l'expérimentation, nous évaluons l'implémentation mono-cœur, multi-cœurs et GPGPU du FastSLAM2.0. Les temps d'exécutions sont calculés après une exécution de l'algorithme sur 500 itérations en utilisant le jeu de données réelles. Les performances du calcul sur une architecture mono-cœur et multi-cœurs CPU constitueront une référence pour évaluer les performances de l'implémentation hétérogène CPU-GPGPU.

4.6.3.1 Évaluation de l'implémentation OpenGL sur la Tegra K1

Nous utiliserons la Tegra K1 pour évaluer l'aspect temporel de l'algorithme et analyser les dépendances. Les langages de programmation utilisés sont : OpenGL pour exploiter les ressources de calcul des 192 cœurs CUDA type GPGPU, C++ pour programmer la version mono-cœur et OpenMP pour aboutir à une implémentation parallèle sur le CPU multi-cœurs.

Temps d'exécution global Pour une première évaluation, nous donnons le temps d'exécution global de l'algorithme et celui de chaque bloc fonctionnel. La figure 4.33 présente la charge "Workload" de chaque bloc fonctionnel de l'algorithme exécuté avec 4096 particules sur un mono-cœur ARM de la tegra K1. Le bloc FB1 est le plus coûteux en temps de calcul. Ce bloc est exécuté plusieurs fois dans une seule itération de l'algorithme en fonction des données odométriques afin de construire la trajectoire du robot. En outre, le calcul de la matrice de covariance P_m est gourmand en temps de calcul. Cette matrice doit être calculée à chaque acquisition des données odométrique ce qui augmente considérablement le temps d'exécution de ce bloc. Les deux bloc FB3 et FB4 ne sont exécutés que s'il y a au moins un amer apparié, donc leur temps d'exécution dépend du nombre d'amers appariés à chaque itération. Le bloc FB5 n'est exécuté que s'il y a au moins un nouveau amer détecté, son temps d'exécution dépend du nombre d'amers à initialiser. Les particules ne sont rééchantillonnées dans le bloc FB6 que s'il y a au moins une probabilité calculée par rapport à un amer apparié et

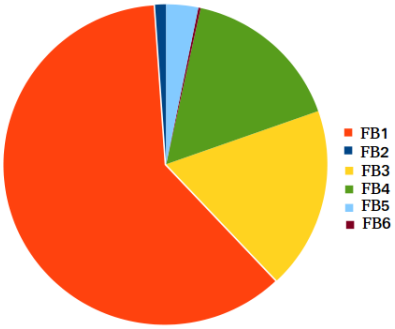


Figure 4.33: *Workload des différents blocs fonctionnels du FastSLAM2.0 sur un mono-cœur ARM de la Tegra K1*

complètement initialisé. Donc le temps d'exécution du bloc FB6 dépend du nombre de probabilités calculées qui ne présentent pas de valeurs aberrantes.

Implémentation hétérogène L'implémentation hétérogène discutée auparavant est illustrée dans la figure 4.34. L'architecture interne du bloc massivement parallèle est détaillée dans la figure 4.35. Elle consiste d'un ensemble des éléments de traitement (PE) qui fonctionne en parallèle et communique avec la mémoire du GPU. La figure 4.36 illustre le cas d'un élément de traitement PE_i correspondant aux équations de mise à jours par filtre de Kalman dans le bloc d'estimation FB4

Le tableau 4.7 synthétise une comparaison des temps d'exécution de chaque bloc fonctionnel et les facteurs d'accélération obtenus après la parallélisation, le nombre de particule utilisées est 4096. Pour une moyenne d'occurrences par itération MOI=15.1, FB1 est exécuté dans 2267.9 ms sur le CPU multi-cœurs . FB3 n'est exécuté que s'il y a au moins un amer apparié. Pour un MOI= 14.1, FB3 est exécuté dans 166.09 ms sur le CPU mono-cœur et 80.71 ms sur le CPU multi-cœurs. Après l'accélération des deux bloc FB1 et FB3 sur le GPU, le temps d'exécution de FB1 diminue à 51.66 ms. Ce qui représente un facteur d'accélération de 44x. Pour FB3, le temps d'exécution diminue à 32.17 ms, ce qui représente un facteur d'accélération de 2.5x. Le bloc FB4 est exécuté dans 100.5 ms sur le CPU mono-cœur, 71.09 ms sur le le CPU multi-cœurs et 19.45 ms sur le GPU, ceci pour un MOI=14.1. Le bloc FB5 est exécuté dans 55.45 ms sur le CPU mono-cœur, 29.58 ms sur le CPU multi-cœurs et 9.65 ms sur le GPU, ceci pour un MOI=10. Le bloc FB6 est exécuté dans 24.87 ms sur le CPU mono-cœur, 9.8 ms sur le CPU multi-cœurs et 2.7 ms sur le GPU, ceci pour un MOI=5.2. Le temps d'exécution global obtenu par l'implémentation hétérogène CPU-GPU est diminué d'un facteur de 20 par rapport à l'implémentation sur le CPU multi-cœurs.

Le facteur d'accélération varie d'un bloc à l'autre en fonction du type d'exécution. Le bloc de prédiction a été parallélisé avec une accélération de 43x par rapport à l'implémentation multi-cœurs . Ceci est dû au faite que la parallélisation du bloc FB1 est faite avec moins de transfert de données entre le CPU et le GPU. Pour un nombre de particules donné M, on ne transfère vers le GPU que M nombres aléatoires générés dans le CPU (les valeurs aléatoires générées sont de type virgule flottante simple précision). Après l'exécution du kernel du bloc FB1 il n y a pas de transfert de retour vers le CPU. Les particules prédites sont conservées dans la mémoire du GPU. Au contraire, les autres blocs fonctionnels (FB3, FB4, FB5 et FB6), nécessitent dans leur exécution un transfert de données du CPU vers le GPU avant l'exécution et un transfert du GPU vers le CPU après exécution. Pour le bloc FB3, on transfère à chaque itération la carte de chaque particule (les paramètres de l'amer apparié) afin de mettre à jour la position des particules. Pour FB4, on transfère à chaque itération les paramètres de l'amer apparié avant l'exécution Après l'exécution, on transfère les paramètres corrigés

de l'amer vers le CPU afin de les classifier dans l'arbre binaire implémenté sur le CPU (figure 4.28). Avant l'exécution du bloc FB5, on transfère la position image des nouvelles observations du CPU vers le GPU. Après l'exécution, on transfère les nouveaux paramètres d'initialisation afin de les classifier dans l'arbre binaire implanté dans le CPU. Avant l'exécution du bloc FB6, on transfère du CPU vers le GPU les probabilités calculées par rapport à chaque amers, après l'exécution on transfère la particules rééchantillonnées du GPU vers le CPU. Ce transfert inévitable entre CPU et GPU réduit le gain qui peut être obtenu après la parallélisation.

Blocs Fonc-tionnels	Temps d'exécution d'une seule occurrence t_{FB_x} (ms)				MOI	Temps d'exécution moyen t'_{FB_x} (ms)				Accélération
	One-Core	Quad-Core	CPU-GPU			One-Core	Quad-Core	CPU-GPU		-
	-	-	Quad-CPU	GPU	-	-	-	Quad-CPU	GPU	-
FB2	6.38	4.83	4.83	-	1	6.38	4.83	4.83	-	1.32
FB1	272.03	150.19	-	3.42	15.1	4107.8	2267.9	-	51.66	43.9
FB3	11.78	5.72	-	2.28	14.1	166.09	80.71	-	32.17	2.50
FB4	7.12	5.04	-	1.37	14.1	100.50	71.09	-	19.45	3.65
FB5	5.45	2.95	-	0.96	10	55.45	29.58	-	9.65	3.06
FB6	4.7	1.88	-	0.51	5.2	24.87	9.8	-	2.7	3.6
Total (ms)	307.46	170.61	13.37		-	4461.09	2463.91	120.46		20.38

Table 4.7: *Temps d'exécution moyens des blocs fonctionnels sur la Tegra K1*

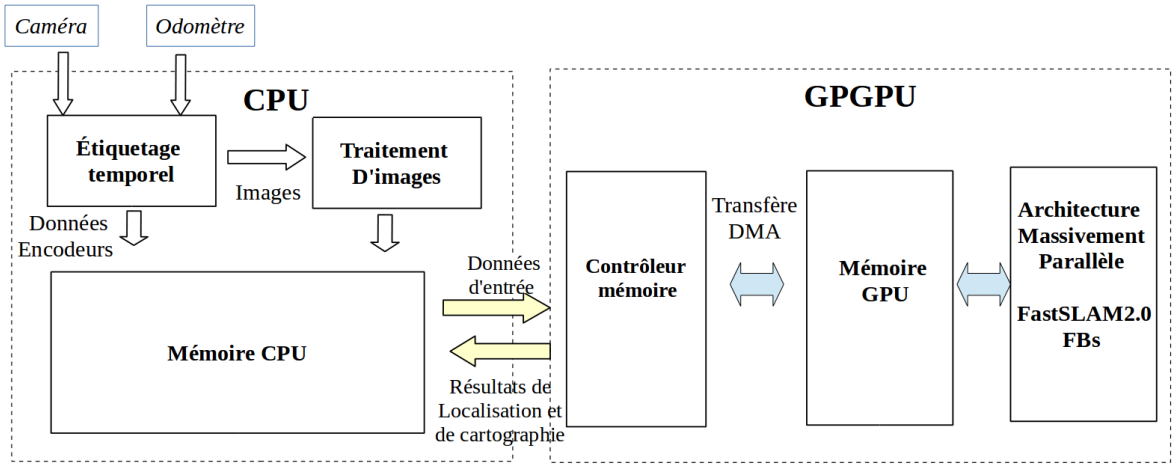


Figure 4.34: *Implantation CPU-GPGPU hétérogène distribuée du FastSLAM2.0*

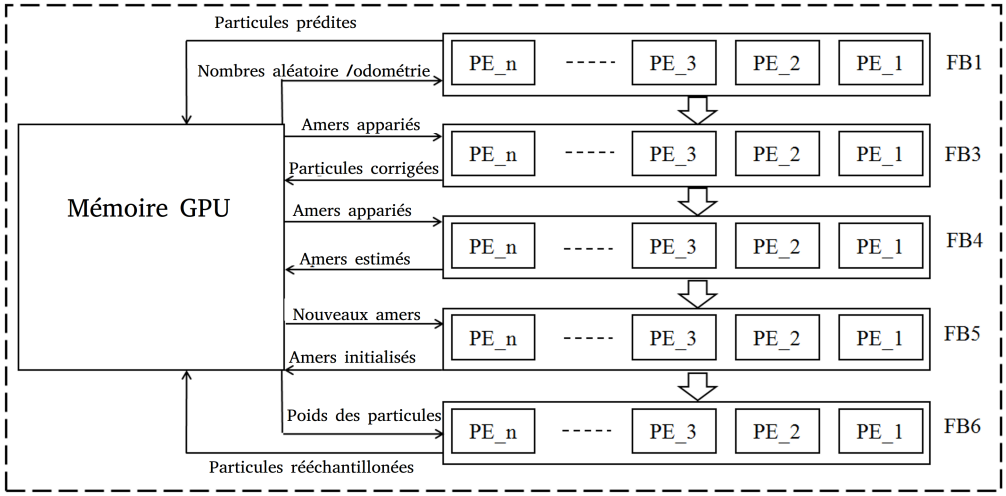


Figure 4.35: *Blocs interne de l'architecture massivement parallèle*

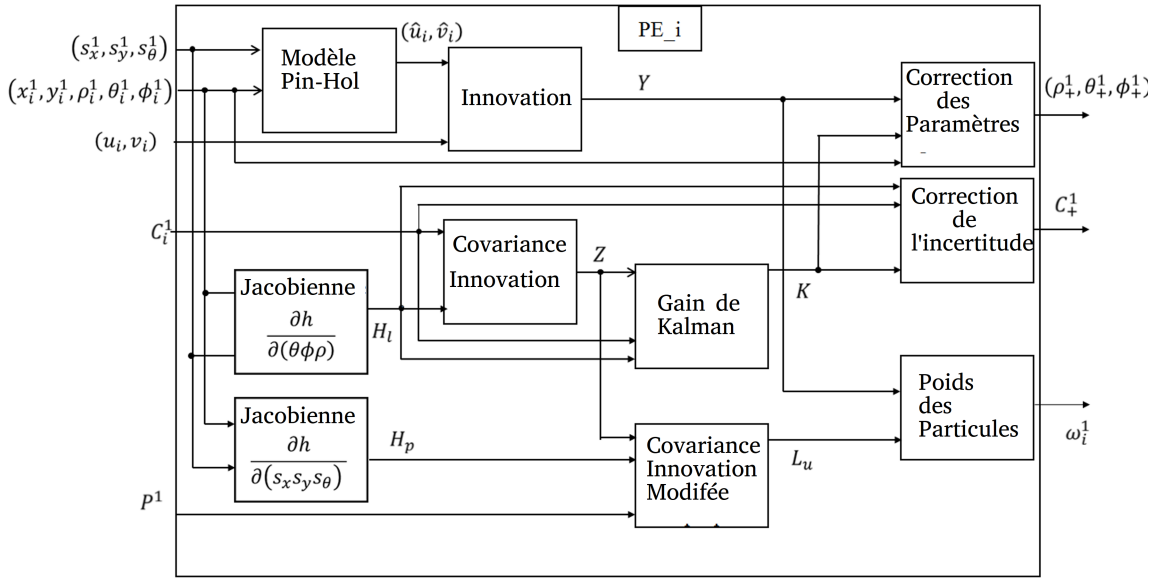


Figure 4.36: *Blocs interne de l'architecture massivement parallèle*

Analyse des dépendances Pour une analyse plus approfondie des dépendances de chaque bloc fonctionnel de l'algorithme, nous avons effectué une implémentation CPU mono-cœur, multi-cœurs et CPU-GPU de l'algorithme pour 500 itérations tout en variant le nombre de particules utilisées (entre 2^4 et 2^{16}). On note que des nombres de particules comme 2^4 ou 2^{16} ne sont pas suffisants pour résoudre le FastSLAM2.0 à grande échelle. La variation de ce nombre permet une bonne analyse en termes de complexité et de dépendances. La figure 4.37 montre les temps d'exécution des bloc fonctionnels en fonction du nombre de particules en échelle logarithmique.

Comme mentionné auparavant, chaque bloc fonctionnel dépend d'un certain nombre de paramètres. La complexité des deux bloc FB3 et FB4 augmente linéairement en fonction du nombre d'amers appariés. La décision de considérer qu'un amer est apparié avec une observation est fortement liée à l'incertitude du robot [40]. Cette incertitude varie avec que la variation du nombre de particules à cause de l'aspect aléatoire de la localisation du Monte Carlo. La variation d'incertitude engendre une variation dans le nombre des amers appariés pour chaque implémentation. Autrement dit, le nombre

des amers appariés n'est pas nécessairement le même lors de l'exécution des différentes implémentation (mono-cœurs, multi-cœurs ou CPU-GPGPU) ni lors de l'utilisation de différents nombres de particules. Ceci est approuvé dans la figure 4.37. Pour l'implémentation GPGPU des deux bloc FB3 et FB4, le nombre des amers appariés lors de l'utilisation de 2^{12} particules est supérieur à celui obtenu lors de l'utilisation de 2^{14}. Par conséquence, l'exécution GPGPU des deux bloc FB3 et FB4 est plus rapide avec 2^{14} qu'avec 2^{12} particules. Ceci est dû au fait que les deux blocs FB4 et FB3 sont exécutés séquentiellement au sein du kernel du GPU (séquence des sous-graphes dans la figure 4.14-b pour FB3 et dans la figure 4.17 pour le bloc FB4). Il en est de même pour l'implémentation CPU mono-cœur et multi-cœurs, l'exécution des deux blocs FB3 et FB4 est légèrement rapide avec 2^{12} qu'avec 2^{10} pour la même raison.

La complexité du bloc FB5 augmente avec le nombre des amers à initialiser et à ajouter dans la carte. La décision de considérer un amer comme nouveau est aussi liée à l'incertitude du robot et au nombre de particules utilisées. Donc, le nombre des nouveaux amers à initialiser est aussi variable d'une implémentation à l'autre. Ceci est prouvé dans la figure 4.37. Les deux implémentations GPU et multi-cœurs CPU peuvent traiter rapidement le bloc FB5 avec 2^{12} particules qu'avec 2^{10}. En effet, le nombre des nouveaux amers à initialiser diminue pour l'implémentation avec 2^{12} particules. Ceci est dû au faite que l'exécution du bloc FB5 est séquentielle au sein du kernel du GPU (séquence des sous-graphes dans la figure 4.17-b) ce qui augmente la complexité d'exécution. Quand le nombre des nouveaux amers augmente régulièrement, la complexité du bloc FB5 augmente de façon linéaire (le cas de l'implémentation mono-cœur du bloc FB6).

La variation du nombre des amers lors de l'exécution d'une implémentation engendre une variation du nombre des probabilités calculées par le bloc FB4. En outre, nous avons implémenté la procédure de gestion de rééchantillonnage discuté dans la section 4.5.3. Cette procédure peut diminuer le nombre des probabilités calculées et donc le temps d'exécution du bloc FB6. La figure 4.37 montre cette dépendance. Le temps d'exécution de l'implémentation GPGPU du bloc FB6 diminue avec 2^8 et 2^{12} particules. Il en est de même pour l'implémentation multi-cœurs du bloc FB6, le temps d'exécution diminue avec 2^{12} particules.

Contrairement, le bloc FB1 ne dépend que du nombre des données odométriques à traiter dans chaque itération. Le nombre des données odométriques fournies par l'encodeur reste inchangé et ne varie pas en fonction de l'implémentation. Par conséquence le complexité du bloc FB1 augmente avec le nombre de particules utilisées (figure.4.37).

Ces dépendances aux paramètres de l'environnement et du système sont inévitables, plus particulièrement dans le cas d'un algorithme SLAM basé sur une approche probabiliste tel que le FastSLAM2.0 monoculaire. La solution proposée est de borner le temps d'exécution en fixant des seuils pour chaque paramètre.

Évaluation du temps d'exécution global Afin d'explorer la puissance de calcul du GPGPU embarqué, on donne le temps d'exécution global pour trois implémentations : mono-cœur, multi-cœurs et CPU-GPGPU (correspondant au modèle d'architecture illustrée dans la figure 4.34, le temps global est donc le temps de traitement d'images sur CPU plus le temps d'exécution des blocs fonctionnel du FastSLAM2.0 sur GPU.) de l'algorithme en utilisant différents nombre de particules. La figure 4.38 montre aussi le facteur d'accélération de l'implémentation GPGPU calculé par rapport à l'implémentation multi-cœurs CPU. Les résultats sont donnés en utilisant une échelle logarithmique.

Pour un petit nombre de particules (2^4), l'implémentation CPU mono-cœur est plus performante que celle utilisant le CPU multi-cœurs et le GPGPU. L'implémentation multi-cœurs n'est pas performante avec peu de particules utilisées (2^4 et 2^8). Ceci est dû au fait que la plus part des données

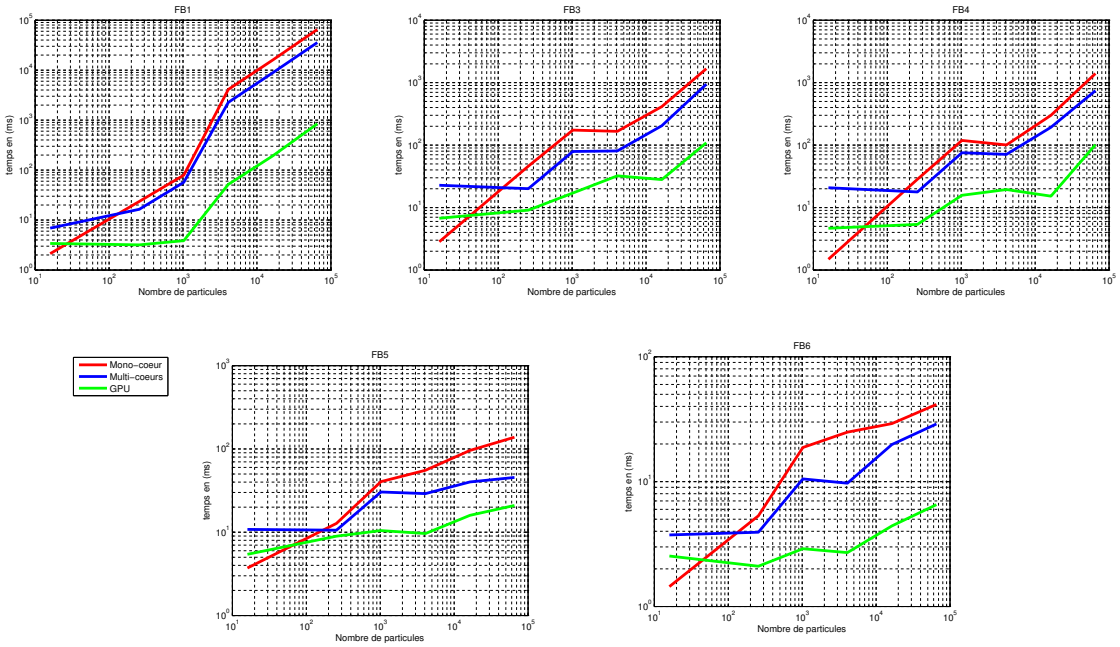


Figure 4.37: *Temps d'exécution des différents blocs fonctionnels en fonction du nombre de particules*

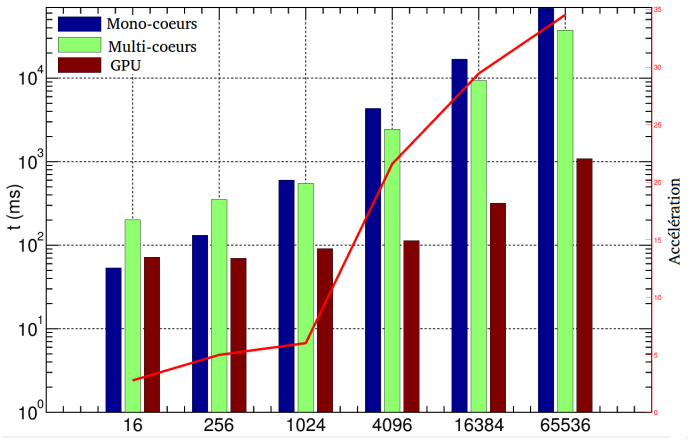


Figure 4.38: *Temps d'exécution et facteur d'accélération global*

sont partagées entre les différents threads (opérateurs) de traitement et que les accès mémoires sont faits de manière séquentielle entre les threads. D'autre part, les barrières de synchronisation entre les différents threads affecte l'effet du parallélisme. Ce problème est relativement souvent imposé dans l'implémentation OpenMP, ce qui engendre une dégradation de la performance obtenue par le CPU multi-cœurs. Dans ce cas, l'implémentation mono-cœur fournit les meilleurs résultats. La parallélisation avec plusieurs threads sur le CPU multi-cœurs ne vaut le coups qu'avec un grand nombre de particules. Dans ce cas, l'implémentation multi-cœurs fournit des meilleurs résultats par rapport à l'implémentation mono-cœur. Avec 2^4 particules, l'implémentation GPGPU du l'algorithme n'est pas performante. Ceci revient au fait que le transfert de données est beaucoup plus important que le temps de traitement avec peu de particules. Par contre, pour un plus grand nombre de particules, le degré de parallélisation devient important et donc l'implémentation GPGPU est plus performante que celle du CPU mono et multi-cœurs .

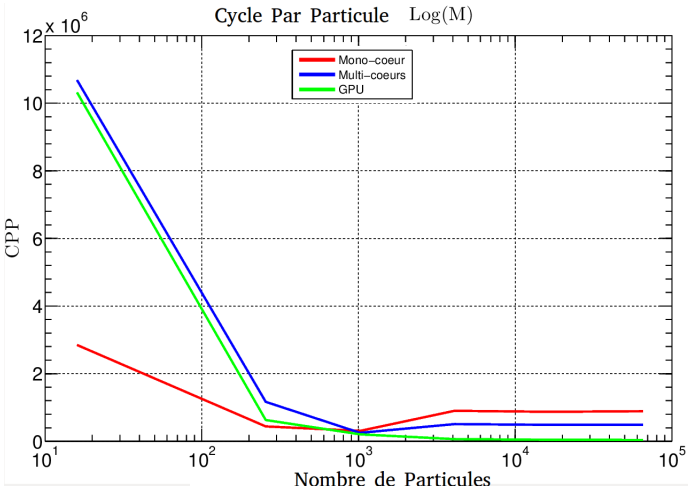


Figure 4.39: *CPP en fonction du nombre de particules*

Évaluation du CPP La figure 4.39 montre le cycle par point des différentes implémentation (mono-cœur, multi-cœurs et GPGPU) du FastSLAM2.0 sur la Tegra K1. Le CPP est calculé par la relation suivante :

$$CPP = \frac{f \times t}{M} \tag{4.4}$$

M est le nombre de particules, f est la fréquence d'horloge en Hz, t est le temps d'exécution en seconde. Le CPP est une métrique utilisée pour comparer les performances des implémentations sur différentes unités de traitement [138]. Pour les deux implémentations mono-cœur et multi-cœurs, la valeur du CPP est minimale pour 2^{10} particules. Ceci est dû au temps d'exécution t des deux implémentations qui augmente de façon significative avec 2^{12} particules (4.38). Dans ce cas le CPU consomme beaucoup de cycle pour traiter l'algorithme avec 2^{12}particules. Cette variation brusque en temps de calcul est causé par les copies mémoire dans la phase de rééchantillonnage. Les copies mémoire sont des opérations coûteuses en temps de calcul en particulier quand il y a beaucoup d'amers dans la carte. Ceci augmente considérablement le temps d'exécution en particulier avec un grand nombre de particules (2^{12}). Dans le cas d'un grand nombre de particules, et pour étudier la complexité, on diminue le seuil maximal des amers dans la carte de chaque particule pour éviter à épuiser les ressources mémoire de l'architecture. Ceci diminue la copie mémoire dans la phase de rééchantillonnage et donc le temps d'exécution global augmente en fonction du nombre de particules dans le filtre. Contrairement à l'implémentation GPGPU, le temps d'exécution global s'incrémente linéairement avec 2^{12}particules. En général, le CPP correspondant à l'implémentation GPGPU est maintenu à une valeur inférieure par rapport aux implémentations CPU mono-coeur et multi-coeurs .

4.6.3.2 Évaluation de l'implémentation OpenCL sur ODROID XU4.

Nous évalué une implémentation OpenCL du FastSLAM2.0 sur une architecture pour l'embarquée ODROID XU4. Le GPU Mali-T628 MP6 de cette architecture est l'un des rares GPU embarqués qui supportent OpenCL et qui, en même temps, contiennent un nombre moyen de cœurs GPGPU (paragraphe 4.6.1, tableau 4.5). Les langages de programmation utilisés sont : OpenCL pour exploiter les ressources de calcul des 8 cœurs SIMD type GPGPU, C++ pour réaliser l'implémentation mono-cœur et OpenMP pour aboutir à une implémentation parallèle sur le CPU octa-cœurs.

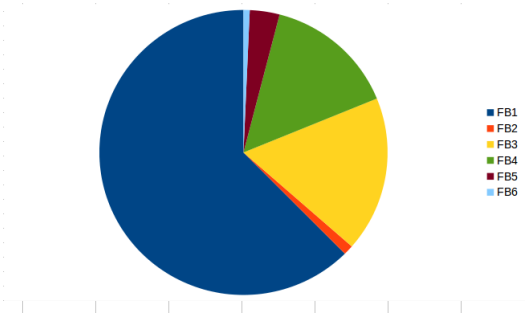


Figure 4.40: *Workload des différents blocs fonctionnels du FastSLAM2.0 sur un seul cœur de l'ODROID XU4*

Temps d'exécution global La figure 4.40 représente la répartition des temps d'exécution de l'implémentation de l'algorithme exécuté avec 4096 particules sur le ARM mono-cœur de l'ODROID XU4. La première constatation qu'on peut faire est que le tâche de prédiction FB1 consomme toujours énormément de ressources sur les deux architectures (ARM Cortex-A15 de la TK1 et l'Exynos 5422 de l'XU4). Deux blocs présentent aussi une consommation importante : le bloc de calcul de la position des particules FB3 et le bloc d'estimation FB4. La répartition des temps est très similaire sur les deux architectures (TK1/XU4). La seule différence majeure est au niveau de la fréquence d'exécution. En comparaison avec les temps de traitement du Cortex A15 sur la TK1, à fréquence inégale, l'architecture du cortex A15 a des résultats meilleurs. En effet, l'Exynos 5422 est pénalisé en fréquence au faveur du nombre de cœurs disponible.

Le tableau 4.8 synthétise une comparaison des temps d'exécution de chaque bloc fonctionnel et les facteurs d'accélération de l'implémentation mono-coeur, octa-cœur et CPU-GPGPU pour 4096 particules. Dans ce tableau, nous donnons directement les temps d'exécutions t'_{FB_x} calculés par rapport au MOI qui reflète la réalité. Une première constatation est que l'implémentation mono-coeur n'est toujours pas efficace, un temps d'exécution global de 5 s par itération ne respecte pas les contraintes temps réel. 1.2 s est le temps d'exécution global sur les 8 cœurs de l'Exynos. L'implémentation multi-coeurs CPU résulte d'un facteur d'accélération de 4.59x par rapport à l'implémentation naïve mono-coeur. Ceci grâce à l'architecture octa-cœurs de l'Exynos 5422. Le temps d'exécution global de l'implémentation hétérogène sur le CPU-GPGPU est 206.19 s. Ce qui implique un facteur d'accélération de 5.95 par rapport à l'implémentation octa-cœur. On remarque que même si le GPU de l'XU4 (Mali-T628 MP6) et le samsung Exynos 5422 contiennent tous les deux un nombre égale de cœurs (8 cœurs SIMD pour le GPU et 8 cœurs pour l'Exynos), l'implémentation GPGPU donne des résultats significativement meilleure. En effet, ceci est dû au même raisons discutées dans la section 2.3.5.Les données sont partagées entre les différents threads, les accès mémoires sont faits de manière séquentielle, les barrières de synchronisation affectent l'effet du parallélisme. Au contraire, l'exécution des kernels au sein du GPU est fortement parallèle, chaque kernel peut accéder à la mémoire indépendamment des autres et on peut donc tirer profit totalement du parallélisme des 8 cœurs SIMD du GPU.

Blocs Fonc-tionnels	MOI	Temps d'exécution moyen t'_{FB_x} (ms)				Accélération
		One-Core	Octa-Core	CPU-GPU		-
	-	-	-	Octa-Core	GPU	-
FB2	1	10.01	4.01	4.01	-	
FB1	15.1	5122.29	1098.7	-	90.06	12.19
FB3	14.1	234.18	60.96	-	56.08	1.087
FB4	14.1	161.80	37.59	-	33.90	1.10
FB5	10	73.194	19.25	-	17.30	1.11
FB6	5.2	37.05	6.53	-	4.84	1.34
Total (ms)	-	5638.52	1227.04	206.19		5.95

Table 4.8: *Temps d'exécution moyen des blocs fonctionnels sur l'XU4*

4.6.3.3 Comparaison de l'implémentation OpenCL et OpenGL

Le langage de programmation du GPU peut être un facteur limitant pour tirer profit au maximum des performances du GPU. Nous avons réalisé une comparaison entre les deux implémentations hétérogène CPU-GPGPU en utilisant OpenCL et OpenGL pour comparer ces deux langages. Afin de mener cette expérimentation, nous avons réalisé les évaluations sur la même architecture qui supportent ces deux langages. Pour rappel, le tableau 4.6 donne les caractéristiques des différentes architectures. Nous avons opté donc pour la station de travail qui supporte OpenCL et OpenGL et intègre le plus grand nombre de cœurs.

Les résultats des deux implémentations OpenCL et OpenGL sont obtenus en utilisant les mêmes stratégies d'optimisation. Autrement dit, les deux implémentations ont été achevées par la transformation du même graphe d'implémentation montré dans la figure 4.28. Les transferts DMA entre CPU et GPU sont réalisés par l'opérateur PBO (pixel buffer object) accessible par les API OpenGL, et par la mémoire (épinglé) accessible par les API OpenCL. Le PBO est un opérateur qui permet un transfert DMA des données. Il est utilisé principalement, pour la lecture/écriture des textures. L'OpenCL dispose des API de bas-niveau qui permettent l'exploitation de la mémoire pour exécuter un transfert DMA. La mémoire épinglé augmente d'avantage la bande passante, par contre c'est une ressource qui doit être exploitée efficacement. Autrement, la sur-utilisation de cette mémoire peut conduire à une dégradation de performances car l'opération d'allocation/désallocation de cette mémoire est coûteuse en termes de temps.

La comparaison confirme que l'utilisation de l'OpenCL permet d'obtenir le maximum de performance en termes d'accélération de calcul. OpenCL permet une bonne gestion de la mémoire qu'OpenGL. L'accès à la mémoire locale, globale et constante est libre au sein du kernel OpenCL. Ceci permet une gestion efficace du transfère et la structure de données. OpenGL offre trois types de mémoire à savoir : la mémoire buffer, texture et le frame buffer (figures 4.25, 4.26). Pourtant, la mémoire texture peut être utilisée pour le calcul générique GPGPU. L'architecture restreinte de cette mémoire affecte les performances de l'implémentation OpenGL. On a besoin de transférer à chaque itération, du et vers le CPU, les données d'entrées sorties. En outre, le frame buffer est une zone mémoire à écriture seule. Autrement dit, le fragment shader n'a pas d'accès à cette mémoire en lecture et donc cette mémoire ne peut pas être utilisée pour le calcul générique. Par conséquence,

l'utilisation du frame buffer object (FBO) est nécessaire pour le calcul générique. Comme mentionné auparavant, le FBO permet l'utilisation des textures pour stocker les résultats de calcul du shader du fragment et aussi les réutiliser comme des entrées pour ce shader (le calcul multi pass et la technique ping-pong). En conséquence, toute opération de lecture/écriture consomme la mémoire texture, vu que cette dernière est limitée en termes de ressources. Les performances de calcul seront aussi limitées. Le lecteur peut se référer à l'annexe-7.4, section 7.4.6 pour plus de détails sur cette comparaison.

4.7 Accélération matérielle sur une architecture programmable

Nous avons vu dans les sections précédentes que l'algorithme FastSLAM2.0 monoculaire a été optimisée sur une architecture hétérogène intégrant un CPU et un GPU. Cette implémentation est faite en accélérant les blocs fonctionnels qui demandent un temps de calcul significatif, sur l'architecture massive des GPU. Souvent, les GPUs utilisé pour cet objectif toutefois imposent des limitations strictes sur l'algorithme accéléré.

Les architectures programmables présentent une solution alternative remarquable pour l'accélération des algorithmes. Les FPGAs disposent d'un avantage par rapport aux coprocesseurs parallèles ordinaire. Les blocs d'éléments logiques sont placés en parallèle ou bien pipelinés pour aboutir à une architecture finale qui exploite toute la capacité du FPGA tout en restant bien évidemment dépendant du flot de données. Un FPGA a toujours été considéré comme la génération des circuits intégrés pouvant d'être entièrement configurés par les utilisateurs. Un FPGA est communément revendiqué pour de meilleures performances et fiabilité. Bien que, lors de l'utilisation des techniques traditionnelles de conception logique, les efforts nécessaires pour exploiter les capacités FPGA afin d'accélérer un algorithmes complexes, rendent souvent le FPGA un choix peu attrayant. La synthèse de haut niveau est une solution révolutionnaire qui renforce l'utilisation des FPGA. IL s'agit d'une technique qui transforme des langages de haut niveau en des éléments logiques visant à exploiter l'architecture reconfigurable de façon efficace pour accélérer les opérations de calcul tout en économisant l'utilisation des ressources.

Nous avons vu que dans le chapitre 2, paragraphe 2.4.3 que la création de bloc de traitement matériel est souvent une tâche longue et compliquée lorsque l'utilisateur doit développer directement en langage de description matériel (VHDL). Cependant des outils existent pour permettre à l'utilisateur de créer des fonctions hardware à partir de programmes écrits en langage de haut niveau. En particulier, le compilateur OpenCL de Altera (AOC : Altera offline Compiler) est un outil utilisé pour convertir des kernels en briques matérielles pour un FPGA. Cette transformation permet d'améliorer de manière significative les performances des fonctions. Cette conversion nécessite quelques changements du code OpenCL pour la gestion des bus de données, de la mémoire et des opérateurs de calculs.

4.7.1 Modèle d'implémentation

Le graphe d'implémentation présenté dans la figure 4.28, section 4.6.2.3, définit un modèle standard de l'implémentation du FastSLAM2.0 sur une architecture hétérogène intégrant un processeur hôte et un coprocesseur massivement parallèle. Autrement dit, les blocs fonctionnels de l'algorithme qui nécessitent un temps d'exécution important sont traités par le coprocesseur parallèle. Le processeur hôte traite les blocs séquentiels qui ne demandent pas de ressources importantes en termes d'opérateurs de calcul. Les architectures hétérogènes étudiées auparavant intègrent un CPU hôte et un processeur graphique pour le traitement parallèle des blocs. Pour aboutir à une implémentation matérielle sur

un FPGA, nous utiliserons OpenCL comme un langage de haut niveau et le compilateur AOC de Altera. OpenCL définit un standard et un modèle de programmation unifié pour des architectures hétérogènes suivant le modèle du processeur hôte pour le traitement séquentiel et un coprocesseur pour le traitement parallèle. Le compilateur offline de Altera suit exactement le même modèle. Autrement dit, il est basé sur un processeur hôte pour le traitement séquentiel des blocs et l'exécution des API OpenCL. Le FPGA est considéré comme un coprocesseur fortement parallèle pour l'accélération matérielle des fonction kernel de l'OpenCL. Par conséquence, nous reconsidérons le FPGA, d'un poni de vue haut-niveau, en suivant le même modèle d'implémentation défini dans la figure 4.28. Pour ce faire, nous utiliserons la même implémentation en OpenCL défini dans le paragraphe 4.6.2.4. Cependant, des modifications sont nécessaires pour adapter le kernel OpenCL pour une transformation optimisée en termes de ressources FPGA. Ces modifications seront discutées dans les sections suivantes.

4.7.2 Conception de haut-niveau par OpenCL

L'outil OpenCL de Altera (Figure 4.41) configure le FPGA par une application OpenCL dans deux étapes primordiales. Tous d'abord, le compilateur offline de l'Altera (AOC) compile les kernels de l'OpenCL. Deuxièmement, le compilateur ordinaire C/C++ côté hôte compile les APIs OpenCL de l'application hôte et crée ensuite une liaison entre le fichier objet généré avec le kernel compilé. Le kernel OpenCL d'une extension (.cl) contient les fonctions à paralléliser. Le compilateur AOC regroupe les différents kernels dans un seul fichier temporaire et ensuite les compile pour générer les fichiers suivants :

- Le fichier objet du compilateur offline d'Altera d'une extension (.aoco) : il s'agit d'un fichier objet intermédiaire qui contient des informations qui seront utilisées pour les prochaines étapes de compilation.
- Le fichier exécutable du compilateur offline d'Altera d'une extension (.aocx) : il s'agit d'un fichier de configuration matérielle qui contient les information nécessaires pour l'exécution.
- Des fichiers intermédiaires à la compilation sont aussi générés pour la création du fichier de configuration du FPGA.

Le fichier de configuration matérielle (.aocx) contient des informations utilisées par l'application hôte pour générer le programme objet qui cible le FPGA. L'application hôte charge ces programmes objets dans la mémoire. Par conséquence, lors de l'exécution de l'application hôte, elle fait appel à ces objets de la mémoire et programme le FPGA cible.

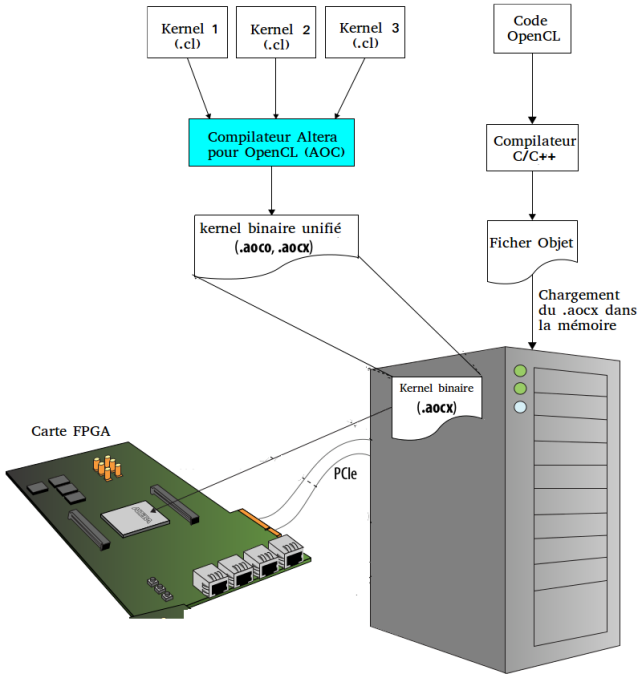


Figure 4.41: *Flot de programmation du FPGA*

Le compilateur Altera peut générer le fichier hardware (**.aocx**) dans une ou plusieurs étapes, tout dépend de la complexité du kernel. Pour des kernels simple qui ne contiennent pas des fonction lourdes, le compilateur Altera génère le fichier de configuration hardware en une seule étape. Ce type de compilation n'est possible que si le kernel cible nécessite des optimisations minimales. Dans notre cas, l'algorithme FastSLAM2.0 est plus complexe par nature et contient des fonctions kernels coûteuse en termes de calcul et d'espace mémoire. Pour cela, nous avons utilisé la compilation multi-processus. Ce type de compilation est utilisé pour optimiser la conception du design et améliorer les performances. La figure 4.42 illustre les différents stages de compilation. Ces étapes dans le flot de conception servent comme des points de passage pour identifier les erreurs de l'aspect fonctionnel et au niveau des performances. Ils nous permettent de modifier convenablement le kernel OpenCL sans avoir besoin de refaire quasiment une compilation complète. Le flot de conception comporte les les étapes suivantes :

— Compilation intermédiaire : cette phase de compilation vérifie les erreurs de syntaxe. Elle génère ensuite le fichier (.aocs) sans construire le fichier hardware de configuration. Elle génère aussi un rapport sur le pourcentage d'utilisation des ressources FPGA dans un fichier (.log). Ce fichier donne une idée sur le type d'optimisation qu'on peut faire.

— Émulation : simule le fonctionnement du kernel en l'exécutant sur une ou plusieurs périphérique sur une machine hôte.

— Le profiling : cette étape permet l'instrumentation des performance du kernel en utilisant le compteur de performance décrit en langage Verilog dans le fichier (.aocx). Lors de l'exécution, le compteur de performance collecte les informations de performance qu'on peut visualiser sur une interface GUI.

— Déploiement complet : si les performances du kernel optimisé obtenues sont satisfaisantes, une compilation complète est faite pour générer le ficher de configuration du FPGA.

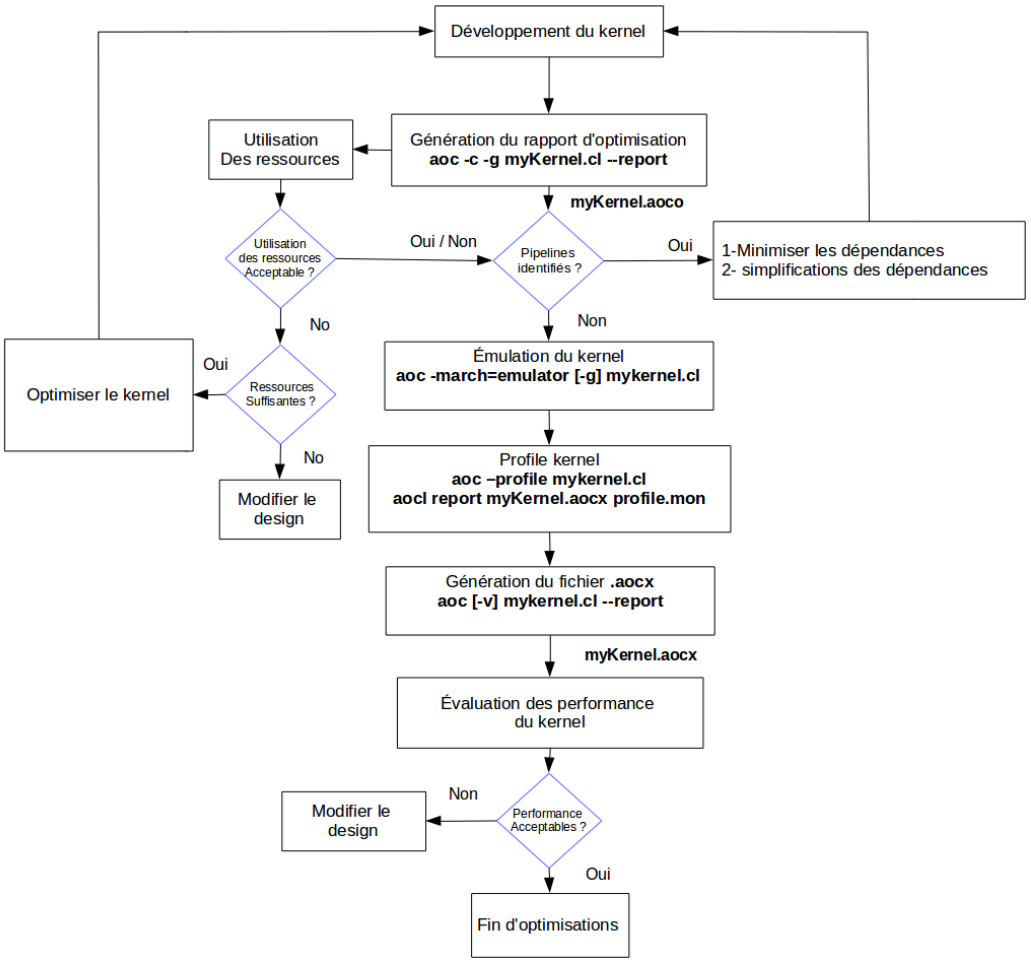


Figure 4.42: *Les différents stages d'optimisation et génération du fichier de configuration du FPGA*

4.7.3 Spécifications matérielle

4.7.3.1 Description du circuit reconfigurable Arria 10

Pour évaluer l'implémentation OpenCL, nous avons utilisé un FPGA récent afin de réaliser des évaluations par rapport aux technologies des architectures SoC récentes. Pour ce faire, nous avons opté pour l'architecture SoC Arria 10 comme cible pour notre implémentation OpenCL du FastSLAM2.0, figure 4.43. L'Arria 10 est l'un des derniers systèmes SoC en technologie 20 *nm* produit par Altera, capable de fournir de meilleur performances en termes de consommation d'énergie. Il permet jusqu'à 1500 GB/s d'opérations virgule-flottante avec les blocs DPS. Le système est cadencé avec une horloge de 100MHz. Le circuit comporte aussi un processeur ARM embarqué cadencé à 1.5 GHz. Cependant, dans notre implémentation, nous avons utilisé le processeur de la station de travail comme hôte puisque la carte intégrant le FPGA est monté sur cette station via le bus PCIe. Le tableau 4.9 montre les ressources disponibles de l'Arria 10 en termes d'éléments logiques, blocs DPS et blocs mémoires.

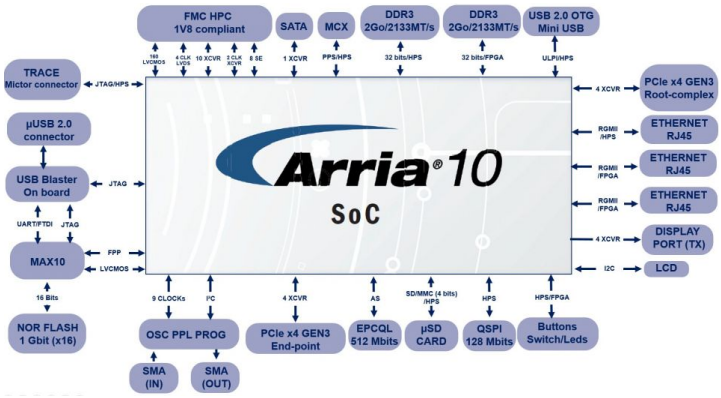


Figure 4.43: *Interfaces du circuit Arria 10*

Table 4.9: *Resources du FPGA du Arria 10*

Resources		Arria 10 SX 660
Éléments logiques (LE) (K)		660K
ALM		251,680
Registres		1,006,720
Mémoire	M20K	42,620
	MLAB	5,788
Blocs DSP		1,687
18 x 19 Multiplieurs		3,374
17.4 Gbps émetteur-récepteur		48
Blocs PCIe IP		2
Contrôleurs mémoires		16

4.7.3.2 Description du plateforme d'expérimentation

La figure 4.44 décrit la plateforme utilisée pour évaluer l'implémentation OpenCL du FastSLAM sur le SoC Arria 10. Nous avons utilisé une station de travail 32 bits. Le processeur de cette machine est cadencé à 2.5 Ghz avec 64GB de mémoire RAM avec un système d'exploitation Red Hat entreprise 7.2. Nous avons utilisé la carte Alaric intégrant l'Arria 10 SoC 660 KLEs (figure 4.45). Elle est montée sur le bus PCIe de la station de travail. Le développement a été fait en utilisant des outils spécifiques aux cartes Altera. Le code OpenCL est compilé sur la machine hôte pour générer le fichier exécutable "EXE". Le kernel OpenCL est compilé par AOC mais nécessite un support logiciel appelé BSP. Le BSP (l'abréviation du *"Board Support Package")* est un logiciel bas niveau de support , qui constitue une interface entre le kernel et le matériel. Le BSP fournit une interface standardisée entre le matériel (Arria 10) et le système d'exploitation. Un BSP n'a pas accès directement au matériel. Un BSP fournit une interface pour les pilotes de périphériques qui à son tour permet au noyau de communiquer avec les composants du matériel tels que les contrôleurs de périphériques, la mémoire interne et les bus externes. Pour générer le fichier de configuration hardware, AOC doit obligatoirement passer par l'outil Quartus en utilisant le langage Verilog. Nous avons utilisé Quartus Prime édition Pro pour la synthèse. La phase de synthèse est contrôlée en utilisant des commandes TCL.

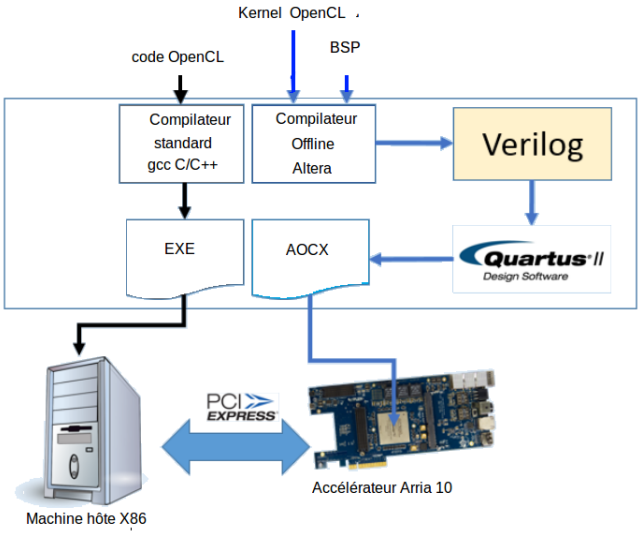


Figure 4.44: *Plateforme de développement sur l'Arria 10*

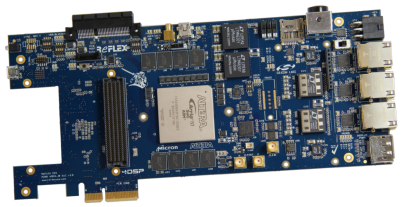


Figure 4.45: *Carte Alaric intégrant l'Arria 10*

4.7.4 Techniques et stratégies d'optimisation

Le kernel OpenCL doit être modifié et adapté au besoin de l'architecture reconfigurable. Pour cela, des optimisations s'avèrent indispensables pour améliorer d'avantage l'efficacité du traitement de données et optimiser les ressources.

Le déroulage de boucle (loop unrolling) Il s'agit d'une technique d'optimisation des boucles visant à augmenter la rapidité d'exécution. On duplique le corps de la boucle de manière à éviter de répéter l'instruction de saut. On utilise cette technique pour permettre aux compilateurs AOC d'optimiser l'exécution des boucles.

Spécification de la taille du groupe de travail La spécification de la taille du groupe de travail à utiliser au sein du kernel est important. Le compilateur AOC repose sur cette spécification pour optimiser d'avantage l'utilisation des ressources matérielles du kernel sans impliquer des éléments logiques en excès. Sans cette spécification, le compilateur assume une valeur par défaut, qui dépend du temps de compilation et des contraintes d'exécution. En spécifiant la taille du groupe de travail, AOC alloue exactement la quantité demandée des ressources matérielles pour bien gérer les sous-groupes de travail dans un groupe.

Spécification du nombre d'opérateurs La possibilité de spécifier le nombre d'opérateurs (unités de calcul) présente un avantage par rapport aux FPGAs. Ceci augmente d'avantage l'efficacité du traitement de données d'un kernel OpenCL. On peut demander au compilateur AOC de générer

des unités de calcul multi-kernel. Chaque unité de calcul est capable d'exécuter multiple groupes de travail simultanément. Cette spécification doit être utilisée avec une précaution, car la multiplication des unités de calcul accroître le débit de données mais cela au détriment de la bande passante de la mémoire globale (l'accès séquentielle à la mémoire globale entre les différentes unités de calcul diminue l'effet du parallélisme). En spécifiant le nombre des unités de calcul, AOC distribue les groupes de travail sur les différents unités de calcul.

Spécification du nombre des unités SIMD Pour augmenter l'efficacité du traitement d'un kernel OpenCL, on peut spécifier le nombre des sous-groupes de travail dans un groupe de travail que AOC exécute en opérations SIMD. En spécifiant le nombre des unités SIMD, AOC reproduit l'exécution des données par rapport à la valeur spécifiée à chaque fois que cela est possible.

Optimisation des accès mémoire Les accès mémoires sont à optimiser au sein d'un kernel pour améliorer l'effet du parallélisme. On peut minimiser l'espace occupé par les blocs de mémoire locale en spécifiant la taille des pointeurs. Il est souvent souhaitable aussi de spécifier le type de la mémoire globale où le processeur hôte peut allouer un buffer. Si non, le processeur hôte utilisera toujours la mémoire qui lui définie par défaut.

4.8 Résultat expérimentaux et comparaison de performance

Le tableau 4.10 résume le pourcentage d'utilisation des ressources de deux versions d'implémentation OpenCL. La colonne "Avant optimisation " donne les ressources utilisées de l'implémentation OpenCL telle qu'elle a été implanté sur GPU et sans aucune optimisation. Les blocs fonctionnels utilisent 120 des éléments logique, 65 des registres logiques, 87 des blocs mémoire et 48 des blocs DSP. Afin d'accélérer tous les blocs fonctionnels, nous avons implémenté les optimisations discutée dans la section 4.7.4 au sein de chaque kernel OpenCL. Le tableau 4.10 colonne "après optimisation" montre le nouveau pourcentage d'utilisation après avoir implanté des optimisation des traitement de données et de la gestion des accès mémoire. Par conséquence tous les blocs fonctionnel exploite au maximum des ressources et peuvent être accélérés sur le FPGA.

Table 4.10: *Pourcentage d'utilisation des ressources FPGA (%)*

	Avant Optimisation				Apres optimisation			
Blocs Fonc-tion-nel	Utilisation des Element logique	Registres Logiques	blocs mémoire	blocs DSP	Utilisation des Elements logique	Registres Logiques	blocs mémoire	blocs DSP
FB1	35	19	22	17	34	19	20	17
FB3	40	20	26	19	33	20	26	19
FB4	29	18	20	13	16	18	20	13
FB5	6	3	6	0	6	3	6	0
FB6	11	5	13	0	8	5	13	0
Total (ms)	121	65	87	49	97	65	85	49

Pour évaluer l'implémentation OpenCL sur FPGA, nous avons utilisé la même méthodologie d'évaluation que précédent. Les résultats sont obtenus un utilisant le jeu de donnée réel. Pour évaluer les performances de l'implémentation de l'agorithme sur FPGA, nous utiliserons différents types du GPU des deux tableau 4.5, 4.6. Quatre GPU embarqués sont utilisés à savoir : le GPU de la XU4, Tegra K1,

Tegra X1 et un GPU haut-de-gamme de la machine bureau. Le tableau 4.11 résume les gains obtenus par l'architecture proposée de l'FPGA comparés à une implantation logicielle sur les GPUs. Les résultats sont obtenus avec 5000 particule et pendant 500 itération de l'exécution de l'algorithme. Les gains obtenus de l'accélération matérielle sur FPGA sont considérables par rapport à l'implémentation sur les GPUs. L'implémentation OpenCL sur FPGA donne les meilleurs résultats par rapport aux GPUs. En effet, le gain maximum obtenu sur le FPGA est de 8x fois par rapport au GPU haut-de-gamme, 10x fois par rapport au 256 cœurs CUDA, 12x fois par rapport à 192 cœurs CUDA et 18x fois par rapport à 8 SIMD GPU. En majorité, tous les blocs fonctionnels ayant un degré de parallélisation fort ce qui implique qu'ils bénéficient fortement de leur architecture matérielle. Plus le degré de parallélisation est importante, plus le gain lors de l'utilisation d'une architecture matérielle est important. Cependant, le nombre d'utilisation des éléments logiques est est assez important pour l'ensemble des blocs de l'algorithme (97%, tableau 4.10).

FBs	ARM - GPU SoC XU4		ARM - GPU SoC K1		ARM - GPU SoC X1		Machine bureau		Hôte - FPGA	
	Octa-Coeurs ARM	GPU	Quad-Coeurs ARM	GPU	Quad-Coeurs ARM	GPU	Hôte	GPU Haut-de-gamme	Hôte	FPGA
FB2	4.01		4.83	-	4.62	-	2.33		2.01	
FB1		90.06	-	51.66		38.33		31.33	-	3.48
FB3		56.08	-	32.17		23.86		19.50	-	2.16
FB4		33.90	-	19.45		14.43		11.79	-	1.31
FB5		17.30	-	9.65		7.15		7.153	-	0.65
FB6		4.84	-	2.7		2.033		1.743	-	0.18
Total (ms)	172.28		120.46		90.42		73.84		9.79	

Table 4.11: *Comparaison des temps d'exécution global entre GPUs et FPGA*

Nous avons vu dans la section 4.5 que pour augmenter la consistence des résultats de localisation en environnement large, il faut augmenter le nombre de particules. Pour évaluer les performances de l'architecture proposée vis à vis aux contraintes de l'environnement réel, nous donnons dans la figure 4.46 les temps d'exécution global pour différentes architectures en fonction du nombre de particules. Pour un grand nombre de particules, l'accélération matérielle de l'algorithme fournit par le FPGA répond aux contraintes temps réel. Par conséquence, même si un grand nombre de particules est nécessaire pour un environnement plus complexe, l'implémentation FPGA sera toujours efficace et peut opérer en temps réel. En outre, les résultats obtenus sont prometteurs en terme de puissance de calcul, les temps obtenus sur le FPGA sont beaucoups plus améliorés par rapport à un GPU haut-de-gamme et par rapport à la TX1 le GPU embarqué le plus puissant à ces jours.

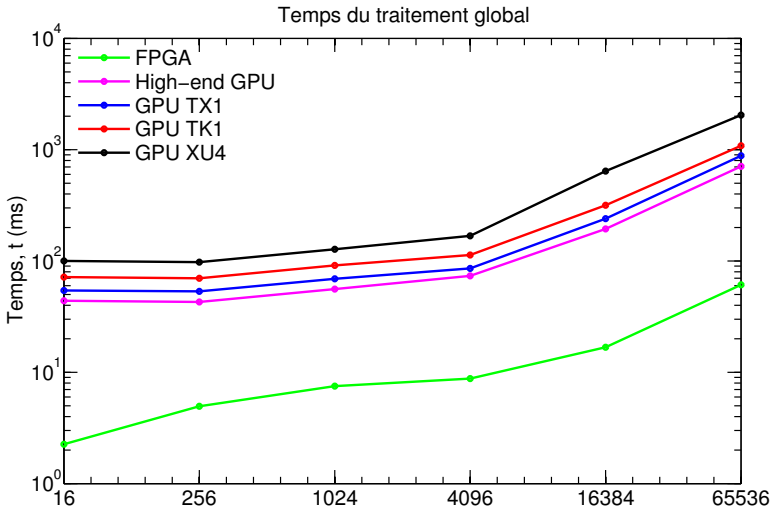


Figure 4.46: *Comparaison des temps d'exécution global entre GPUs et FPGA pour différents nombre de particules*

La figure 4.47 montre une comparaison du CPP entre les quatre architectures étudiées. Pour peu de particules, le CPP demandé est grand pour toutes les architectures parallèles. En augmentons le nombre de particules, le CPP du l'architecture dédiée demeure à une valeur minimal par rapport aux GPUs.

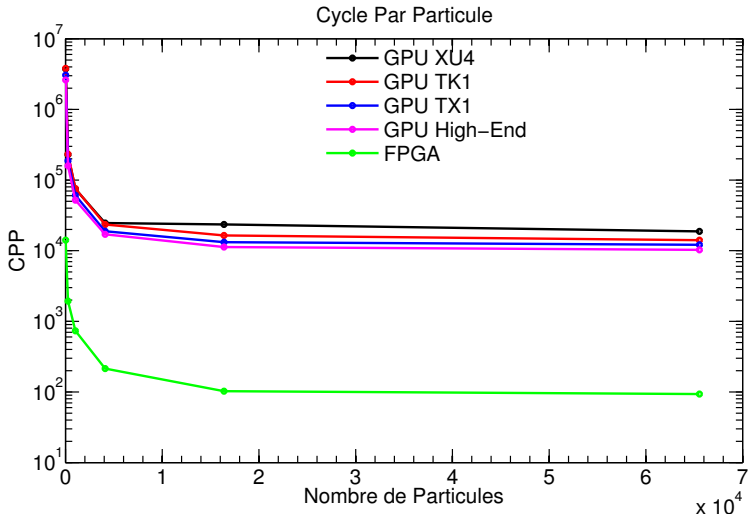


Figure 4.47: *Comparaison du cycle par particule entre les GPUs et le FPGA*

On étudie dans la figure 4.48, le facteur d'accélération global du FPGA par rapport. aux différents GPUs Le facteur d'accélération est donné en fonction du nombre de particule. Avec un nombre de particules minimal, l'implémentation du FPGA fournit les meilleurs résultats par rapport au GPUs et on obtient les meilleurs facteur d'accélération. Ceci est expliqué par le fait que le FPGA est largement rapide qu'il n'est pas affecté par le temps du transfère. Au contraire, les performances des GPUs sont dégradées avec peu de particules puisque le temps du transfère est suffisamment large par rapport aux temps du traitement au sein du kernel. Le tableau 4.12 synthétise une comparaison du transfère DMA entre les différentes architecture. Le transfère DMA entre le processeur hôte et FPGA via le bus PCIe est beaucoup plus rapide que celui dans les autres architectures ARM-GPU. Plus le nombre de particules augmente, le fatceur d'accélération du FPGA par rapport au GPU décroît. Ceci par-

ce-que le degré de parallélisation du GPU augmente et devient important (plus d'unité de calcul seront utilisées pour traiter les particules) et par conséquence le temps d'exécution global diminue. Toutefois, pour un plus grand nombre de particule, le degré de parallélisation devient limité dans le GPUs. Autrement dit, les GPUs ne dispose pas d'assez d'unité de calcul pour traiter une large quantité de particules. Au contraire, en augmentant le nombre de particule, on peut exploiter au maximum les performances du FPGA et donc les temps d'exécution sur le FPGA diminuent ce qui augmente le facteur d'accélération par rapport aux GPUs.

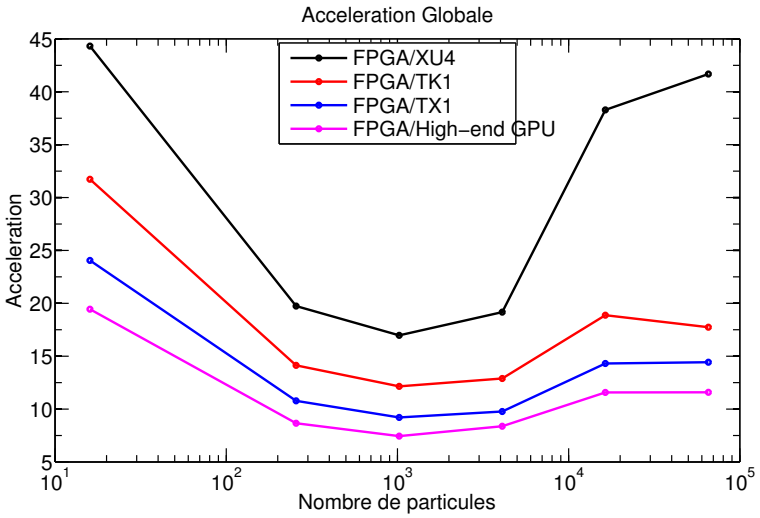


Figure 4.48: *Comparaison du facteur d'accélération entre les GPUs et le FPGA*

	Architectures	Temps de transfère GB/s
XU4	ARM-GPU	0.85
TK1	ARM-GPU	1.22
TX1	ARM-GPU	1.53
Machine Bureau	Intel-GPU	1.98
Arria 10	Intel-FPGA	2.923

Table 4.12: *Comparaison du temps de transfert entre un transfert DMA sur les architectures XU4, TK1, TX1 et un transfert via le bus PCIe sur la station de travail et l'Arria 10*

L'architecture proposée sur FPGA par l'OpenCL, dispose de très bonnes performances pour les blocs fonctionnels ayant une forte degré de parallélisation et disposant de plusieurs mémoires. Elle permet de réaliser en parallèle des traitements très importants. Cette architecture est très adaptée dans le cas du FastSLAM2.0 monoculaire à large échelle fonctionnant avec un large nombre de particule et qui nécessite de nombreux calculs indépendants.

4.9 Bilan

L'étude du FastSALM2.0 monoculaire nous a permis de définir un système complet permettant de résoudre la problématique du SLAM. Pour cela, nous avons optimisé son implantation tout d'abord sur une architecture homogène intégrant plusieurs cœurs CPU. Cette première version d'optimisation résulte d'un système de SLAM capable de cartographier une zone très restreinte dans l'environnement. Nous avons dû limiter le nombre de particules utilisées par le filtre pour respecter les contrainte de temps réel et la nature de l'environnement.

Pour pouvoir cartographier une plus large zone, nous avons choisi d'étudier, implanter et optimiser l'algorithme FastSLAM2.0 sur une deuxième variante d'architecture. Cela a nécessité tout d'abord des modifications et des réécritures algorithmiques du FastSLAM2.0. Ces modifications ont rendu l'algorithme consistant dans des environnements large à grande échelle. Cependant, cela était au détriment des performances temps réel. Pour résoudre ce problème, nous avons proposé un premier modèle conçu autour d'une architecture hétérogène intégrant un CPU et un GPGPU. Nous avons pu obtenir grâce à ce modèle un facteur d'accélération global de 23x avec le GPU embarqué le plus puissant à ce jour. Notre étude à démontré que l'adéquation algorithme architecture est indispensable pour obtenir des performances satisfaisantes. De plus, nous avons démontré via une comparaison entre OpenCL et OpenGL que la nature du langage utilisé pour le développement en GPU est important, son choix adéquat pourrait dans certain, permettre d'optimiser de plus l'algorithme et de bénéficier de mieux des spécificité de l'architecture du GPU.

Malgré ces optimisations importante sur l'architecture massivement parallèle, la taille de l'environnement reste relativement restreinte, ceci est dû que pour certain environnement plus complexe, l'incertitude des particules dépend très fortement sur le nombre de particule utilisée. Ceci nous amène donc à proposer un deuxième modèle conçu autour des architecture reconfigurable dédiées. Pour cela nous avons défini une architecture matérielle intégrant un CPU et un FPGA entièrement dédiée au FastSLAM2.0 monoculaire. L'implémentation a été aboutie en utilisant un langage de synthèse de haut-niveau au lieu des langages traditionnels afin de minimiser le temps développement. Nous avons comparé les résultat d'implémentation sur notre architecture matérielle par rapport à l'implémentation logicielle sur les différent GPUs embraqué grand public existant. Les gains obtenus sont conséquent même par rapport aux GPUs haut-de-gamme qui disposent actuellement d'un nombre énorme de processeurs de calcul. La multiplication des cœurs du calcul engendrant un fort parallélisme permettrait, dans notre cas, une meilleur répartition de calcul. Par conséquence, cela résulte d'un système de SLAM qui présente un compromis entre la consistance des résultats dans des environnement aussi complexe sans dégradation des performances temps réal.

Finalement, l'utilisation d'une architecture programmable associée à un processeur standard sembre être, à ce jour, la solution adéquate pour des applications de SLAM temps réel. Les parties qui ne demande pas d'un fort parallélisme ni de ressources importantes, peuvent être traités sur le processeur CPU standard. Tandis que ceux demandant un fort parallélisme peuvent être accéléré sur le FPGA. Cela est devenu possible, aujourd'hui, grâce aux développement des interfaçage avec un bus de données suffisamment rapide qui minimise le problème des accès mémoire.

Chapitre 5

Conclusion Générale

5.1 Conclusion et résumé des contributions

L'objectif principal de cette thèse était l'étude de la portabilité des algorithmes SLAM sur des architectures dédiées pour l'embarqué.

Le développement des algorithmes SLAM est un domaine très actif depuis plusieurs décennies dans lesquelles plusieurs solutions existaient pour résoudre le problème de localisation et de cartographie. En effet, de nombreux systèmes de SLAM développés sont principalement basés sur trois approches algorithmiques principales à savoir : l'approche probabiliste ou à base de graphes, l'odométrie visuelle et l'approche bio-inspirée. Ces algorithmes utilisent soit des capteurs haut de gamme comme les capteurs lasers, soit des capteurs bas-coût comme les caméras. Ils réalisent soit le full SLAM ou bien le SLAM en ligne. Nous avons donc étudié les différents algorithmes existant tout en étudiant la nature du traitement déployée ainsi que leur consistance en termes de résultats de localisation. Nous avons fait le choix de quatre algorithmes de différentes catégories à évaluer. L'évaluation temporelle sur des architectures homogènes embarquées nous a amené à proposer le FastSLAM2.0 comme étant un algorithme adapté aux architectures parallèles.

Nous avons donc procédé à des réécritures algorithmiques du FastSLAM2.0 monoculaire pour adapter son fonctionnement aux environnements larges. Pour ce faire, nous avons amélioré la façon dont les particules sont rééchantillonnées en proposant une méthode pour le calcul de l'incertitude au fil du temps. Nous avons aussi amélioré la phase de rééchantillonnage en proposant une méthode de suppression des amers aberrants afin de garder une localisation correcte et une dispersion des particules dans l'espace. Ces modifications ont révélé la nécessité d'une optimisation de l'algorithme pour garantir un compromis entre la consistance des résultats en environnements larges et le fonctionnement en temps réel. Pour pallier à ce problème, nous avons donc défini un premier modèle de système autour d'une architecture hétérogène à base de GPU. Nous nous sommes basés sur une méthodologie d'adéquation algorithme architecture pour un partitionnement efficace du FastSLAM2.0 monoculaire sur l'architecture cible. Cette méthodologie consiste à définir un graphe d'implémentation adéquat à partir des deux graphes de l'algorithme et de l'architecture. Ces graphes permettait la définition et la répartition des blocs fonctionnels sur les différents processeurs CPU et GPU de l'architecture cible. Le modèle de partitionnement qu'on a proposé a permis un gain de temps de traitement considérable par rapport à une implémentation homogène sur un CPU embarqué. Les résultats d'accélération obtenus permettaient d'avoir un système de SLAM qui répond aux contraintes du temps réel.

Le premier système a mis en avant les avantages offerts aujourd'hui par les nouvelles architectures hétérogènes CPU-GPU de faible consommation, en particulier pour la parallélisation multicoeurs et la portabilité du FastSLAM2.0. Cependant, pour des environnements aussi plus complexe, la complexité algorithmique du FastSLAM2.0 monoculaire augmente d'avantage, ce qui nécessite un fort parallélisme afin de produire des résultats consistants et en temps réel.

Nous avons proposé un deuxième modèle de système autour d'une architecture programmable à base de FPGA. Ce deuxième modèle a été étudié en suivant la même méthodologie que précédemment. En effet, le même graphe d'implémentation a été adopté et un langage de synthèse de haut niveau est utilisé pour réaliser cette implémentation. Ce deuxième modèle a été comparé au premier modèle CPU-GPU. Les résultats montrent un gain conséquent pour l'architecture programmable. Ce modèle a confirmé la possibilité de concevoir un système de SLAM capable de garantir un comportement en temps réel en environnement large et complexe tout en gardant des résultats consistants de localisation et cartographie simultanées.

5.2 Perspectives

L'implémentation embarquée des algorithmes SLAM est un domaine rarement étudié. Malgré tout, des optimisations sont fréquemment proposées pour améliorer les performances en temps réel des systèmes embarqués impliquant de la localisation et cartographie simultanées. A l'issu de cette thèse, nous envisageons les perspectives suivantes :

L'intégration du système SLAM résultant sur une plateforme expérimentale

Notre système SLAM conçu a été validé et évalué en utilisant une méthodologie HIL . Cependant, une utilisation d'une platforme expérimentale implémentant le système SLAM s'avère nécessaire pour confronter les contraintes imposées par la réalité du terrain. Ces travaux peuvent être envisagés en utilisant la le véhicule électrique automatisé du laboratoire.

Utilisation des outils avancés pour la génération automatique des exécutifs

La méthodologie d'adéquation algorithme architecture, connue par l'acronyme A3, est un domaine en cours de développement. Elle se base principalement sur des outils afin de trouver le meilleur couple distribution/ordonnancement du graphe de l'algorithme sur le graphe d'architecture cible afin de générer des exécutifs. Ces outils ne sont pas utilisés dans notre thèse vu qu'ils ne sont pas encore capable de prendre en compte des algorithmes hyper complexes comme le SLAM et ne tiennent pas en compte l'hétérogénéité des architectures récentes. Par conséquence, dans notre thèse le graphe d'implémentation est obtenu à la main en se basant sur des contraintes algorithmiques et matérielles. L'amélioration des outils existants pour la génération automatique du graphe d'implémentation et les exécutifs correspondants des algorithmes complexes pourrait permettre de définir plusieurs modèles d'implémentation afin d'en choisir le meilleur.

Mettre en place des algorithmes de contrôle commande afin d'augmenter l'autonomie des robots

Les algorithmes SLAM offrent des avantages indéniables quant à la localisation des robots et la cartographie de leurs environnements. Cependant, ils n'assurent pas une autonomie absolue des

robots. Des algorithmes de contrôle commande sont à mettre en place en complément aux algorithmes de SLAM afin de permettre aux robots de planifier leurs trajectoires. Une étude de portabilité de ces algorithmes sur des architectures embarquées est nécessaire pour garantir la prise de décision en temps réel.

Chapitre 6

Références

Bibliographie

[1] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard et Daniel Cremers : A benchmark for the evaluation of rgb-d slam systems. *In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 573–580. IEEE, 2012.

[2] Raul Mur-Artal, JMM Montiel et Juan D Tardos : Orb-slam : a versatile and accurate monocular slam system. *Robotics, IEEE Transactions on*, 31(5):1147–1163, 2015.

[3] David Ball, Scott Heath, Janet Wiles, Gordon Wyeth, Peter Corke et Michael Milford : Openratslam : an open source brain-based slam system. *Autonomous Robots*, 34(3):149–176, 2013.

[4] YVES SOREL : Adéquation algorithme architecture. *INRIA Rocquencourt, 78153, Le Chesnay Cedex*.

[5] Mohamed Abouzahir, Rachid Latif, Tajer Abdelouahed, Abdelhafid Elouardi et Samir Bouaziz : Localization and mapping algorithms implemented on a low-power embedded architectures : A case study. *In International Conference on Multimedia Computing and Systems ICMCS'16*, Sept 2016.

[6] M. Abouzahir, A. Elouardi, S. Bouaziz, R. LATIF et A. Tajer : Fastslam 2.0 running on a low-cost embedded architecture. *In IEEE. The 13th International Conference on Control, Automation, Robotics and Vision, ICARCV*, Marina bay Sands, Singapour, 2014.

[7] Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif et Tajer Abdelouahed : An improved rao-blackwellized particle filter based-slam running on an omap embedded architecture. *In Complex Systems (WCCS), 2014 Second World Conference on*, pages 716–721, Nov 2014.

[8] Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif et Abdelouahed Tajer : Implementation of fastslam2.0 on an embedded system and hil validation using different sensors data. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, 2015(91):91–119, 2015.

[9] Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif et Abdelouahed Tajer : Large-scale monocular fastslam2. 0 acceleration on an embedded heterogeneous architecture. *EURASIP Journal on Advances in Signal Processing*, 2016(1):88, 2016.

[10] Samir BOUAZIZ Omar HAMMAMI Mohamed ABOUZAHIR, Abdelhafid ELOUARDI et Ismail ALI : High-level synthesis for fpga design based-slam application. *In AICCA'16*, Nov-Dec 2016.

[11] Tim Bailey et Hugh Durrant-Whyte : Simultaneous localization and mapping (slam) : Part ii. *IEEE Robotics & Automation Magazine*, 13(3):108–117, 2006.

[12] Hugh Durrant-Whyte et Tim Bailey : Simultaneous localization and mapping : part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[13] Sebastian Thrun et John J Leonard : Simultaneous localization and mapping. *In Springer handbook of robotics*, pages 871–889. Springer, 2008.

[14] Marcus A Brubaker, Andreas Geiger et Raquel Urtasun : Lost ! leveraging the crowd for probabilistic visual self-localization. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3057–3064, 2013.

[15] Sebastian Thrun, Wolfram Burgard et Dieter Fox : *Probabilistic robotics*. MIT press, 2005.

[16] Gamini Dissanayake, Shoudong Huang, Zhan Wang et Ravindra Ranasinghe : A review of recent developments in simultaneous localization and mapping. *In 2011 6th International Conference on Industrial and Information Systems*, pages 477–482. IEEE, 2011.

[17] Josep Aulinas, Yvan R Petillot, Joaquim Salvi et Xavier Lladó : The slam problem : a survey. *In CCIA*, pages 363–371. Citeseer, 2008.

[18] Jos Neira, Andrew J Davison et John J Leonard : Guest editorial special issue on visual slam. *IEEE Transactions on Robotics*, 24(5):929–931, 2008.

[19] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss et Wolfram Burgard : A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[20] Friedrich Fraundorfer et Davide Scaramuzza : Visual odometry : Part ii : Matching, robustness, optimization, and applications. *IEEE Robotics & Automation Magazine*, 19(2):78–90, 2012.

[21] Davide Scaramuzza et Friedrich Fraundorfer : Visual odometry [tutorial]. *IEEE Robotics & Automation Magazine*, 18(4):80–92, 2011.

[22] Sajad Saeedi, Michael Trentini, Mae Seto et Howard Li : Multiple-robot simultaneous localization and mapping : A review. *Journal of Field Robotics*, 33(1):3–46, 2016.

[23] Stephanie Lowry, Niko Sünderhauf, Paul Newman, John J Leonard, David Cox, Peter Corke et Michael J Milford : Visual place recognition : A survey. *IEEE Transactions on Robotics*, 32(1):1–19, 2016.

[24] Alonzo Kelly : *Mobile Robotics : Mathematics, Models, and Methods*. Cambridge University Press, 2013.

[25] Paul Newman, John Leonard, Juan D Tardós et José Neira : Explore and return : Experimental validation of real-time concurrent mapping and localization. *In Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 2, pages 1802–1809. IEEE, 2002.

[26] Christian Forster, Luca Carlone, Frank Dellaert et Davide Scaramuzza : On-manifold preintegration theory for fast and accurate visual-inertial navigation. *arXiv preprint arXiv :1512.02363*, 2015.

[27] Simon Lynen, Torsten Sattler, Michael Bosse, Joel Hesch, Marc Pollefeys et Roland Siegwart : Get out of my lab : Large-scale, real-time visual-inertial localization. *In Robotics : Science and Systems*, 2015.

[28] Kuka Robotics. Kuka Navigation Solution. `http://www.kuka-robotics.com/res/robotics/Products/PDF/EN/KUKA_Navigation_Solution_EN.pdf`, 2016. [Online ; accessed 19-July-2016].

[29] Mark Maimone, Yang Cheng et Larry Matthies : Two years of visual odometry on the mars exploration rovers. *Journal of Field Robotics*, 24(3):169–186, 2007.

[30] IEEE Spectrum. AutomatonRoboticsHome Robots Dyson's Robot Vacuum Has 360-Degree Camera, Tank Treads, Cyclone Suction. `http://spectrum.ieee.org/automaton/robotics/home-robots/dyson-the-360-eye-robot-vacuum`, 2016. [Online ; accessed 19-July-2016].

[31] Google. Project tango. `https://get.google.com/tango/`, 2016. [Online ; accessed 19-July-2016].

[32] Feng Lu et Evangelos Milios : Globally consistent range scan alignment for environment mapping. *Autonomous robots*, 4(4):333–349, 1997.

[33] J-S Gutmann et Kurt Konolige : Incremental mapping of large cyclic environments. *In Computational Intelligence in Robotics and Automation, 1999. CIRA'99. Proceedings. 1999 IEEE International Symposium on*, pages 318–325. IEEE, 1999.

[34] Frank R Kschischang, Brendan J Frey et H-A Loeliger : Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[35] Sebastian Thrun : Probabilistic robotics. *Commun. ACM*, 45(3):52–57, 2002.

[36] Randall Smith, Matthew Self et Peter Cheeseman : Estimating uncertain spatial relationships in robotics. *In Autonomous robot vehicles*, pages 167–193. Springer, 1990.

[37] Michael Bosse, Paul Newman, John Leonard, Martin Soika, Wendelin Feiten et Seth Teller : An atlas framework for scalable mapping. *In Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 2, pages 1899–1906. IEEE, 2003.

[38] Jose E Guivant : *Efficient simultaneous localization and mapping in large environments*. Thèse de doctorat, Citeseer, 2002.

[39] Lina M Paz, Pedro Piniés, Juan D Tardós et José Neira : Large-scale 6-dof slam with stereo-in-hand. *IEEE transactions on robotics*, 24(5):946–957, 2008.

[40] M. Montemerlo, S. Thrun, D. Koller et B. Wegbreit : FastSLAM : A factored solution to the simultaneous localization and mapping problem. *In Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI.

[41] M. Montemerlo, S. Thrun, D. Koller et B. Wegbreit : FastSLAM 2.0 : An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003. IJCAI.

[42] A.J. Davison, I.D. Reid, N.D. Molton et O. Stasse : MonoSLAM : Real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1052–1067, 2007.

[43] Andrew P Gee, Denis Chekhlov, Andrew Calway et Walterio Mayol-Cuevas : Discovering higher level structure in visual slam. *IEEE Transactions on Robotics*, 24(5):980–990, 2008.

[44] Ethan Eade et Tom Drummond : Scalable monocular slam. *In 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 1, pages 469–476. IEEE, 2006.

[45] Laura A Clemente, Andrew J Davison, Ian D Reid, José Neira et Juan D Tardós : Mapping large loops with a single hand-held camera. *In Robotics : Science and Systems*, volume 2, page 2, 2007.

[46] A.J. Davison : Real-time simultaneous localisation and mapping with a single camera. *In IEEE Int. Conf. on Computer Vision*, pages 1403–1410, 2003.

[47] Javier Civera, Andrew J Davison et JM Martinez Montiel : Inverse depth parametrization for monocular slam. *IEEE transactions on robotics*, 24(5):932–945, 2008.

[48] Marc Pollefeys, Luc Van Gool, Maarten Vergauwen, Frank Verbiest, Kurt Cornelis, Jan Tops et Reinhard Koch : Visual modeling with a hand-held camera. *International Journal of Computer Vision*, 59(3):207–232, 2004.

[49] Bill Triggs, Philip F McLauchlan, Richard I Hartley et Andrew W Fitzgibbon : Bundle adjustment—a modern synthesis. *In International workshop on vision algorithms*, pages 298–372. Springer, 1999.

[50] David Nistér, Oleg Naroditsky et James Bergen : Visual odometry. *In Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–652. IEEE, 2004.

[51] Etienne Mouragnon, Maxime Lhuillier, Michel Dhome, Fabien Dekeyser et Patrick Sayd : Monocular vision based slam for mobile robots. *In 18th International Conference on Pattern Recognition (ICPR'06)*, volume 3, pages 1027–1031. IEEE, 2006.

[52] Georg Klein et David Murray : Parallel tracking and mapping for small ar workspaces. *In Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234. IEEE, 2007.

[53]

[54] Kurt Konolige et Motilal Agrawal : Frameslam : From bundle adjustment to real-time visual mapping. *IEEE Transactions on Robotics*, 24(5):1066–1077, 2008.

[55] Kurt Konolige, James Bowman, JD Chen, Patrick Mihelich, Michael Calonder, Vincent Lepetit et Pascal Fua : View-based maps. *The International Journal of Robotics Research*, 2010.

[56] Hauke Strasdat, JMM Montiel et Andrew J Davison : Real-time monocular slam : Why filter ? *In Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2657–2664. IEEE, 2010.

[57] Michael J Milford, Gordon F Wyeth et DF Rasser : Ratslam : a hippocampal model for simultaneous localization and mapping. *In Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 1, pages 403–408. IEEE, 2004.

[58] Michael J Milford et Gordon F Wyeth : Mapping a suburb with a single camera using a biologically inspired slam system. *IEEE Transactions on Robotics*, 24(5):1038–1053, 2008.

[59] Arren J Glover, William P Maddern, Michael J Milford et Gordon F Wyeth : Fab-map+ ratslam : Appearance-based slam for multiple times of day. *In Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3507–3512. IEEE, 2010.

[60] Matthew Collett : How desert ants use a visual landmark for guidance along a habitual route. *Proceedings of the National Academy of Sciences*, 107(25):11638–11643, 2010.

[61] Shih-An Li, Chen-Chien Hsu, Wen-Ling Lin et Jui-Pin Wang : Hardware/software co-design of particle filter and its application in object tracking. *In IEEE International Conference on System Science and Engineering (ICSSE)*, pages 87–91, 2011.

[62] M Moyers, David Stevens, Vassilios Chouliaras et David Mulvaney : Implementation of a fixed-point fastslam 2.0 algorithm on a configurable and extensible vliw processor. *In IEEE international conference on eletronics circuits and systems (ICECS), Tunisia*, 2009.

[63] Thomas CP Chau, Xinyu Niu, Alison Eele, Wayne Luk, Peter YK Cheung et Jan Maciejowski : Heterogeneous reconfigurable system for adaptive particle filters in real-time applications. *In Reconfigurable Computing : Architectures, Tools and Applications*, pages 1–12. Springer, 2013.

[64] Oğuz Tosun *et al.* : Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures. *In Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 820–826, Baden-Baden, Germany, 2011. IEEE.

[65] S. Maskell, Ben Alun-Jones et Malcolm Macleod : A single instruction multiple data particle filter. *In IEEE Nonlinear Statistical Signal Processing Workshop*, pages 51–54, Sept 2006.

[66] Gustaf Hendeby, Rickard Karlsson et Fredrik Gustafsson : Particle filtering : The need for speed. *EURASIP J. Adv. Signal Process*, 2010:22 :1–22 :9, février 2010. ISSN 1110-8657.

[67] Haiyang Zhang et Fred Martin : Cuda accelerated robot localization and mapping. *In IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, 2013.

[68] D. T. Tertei, J. Piat et M. Devy : Fpga design and implementation of a matrix multiplier based accelerator for 3d ekf slam. *In 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014.

[69] Biruk G Sileshi, Juan Oliver, R Toledo, Jose Gonçalves et Pedro Costa : Particle filter slam on fpga : A case study on robot@ factory competition. *In Robot 2015 : Second Iberian Robotics Conference*, pages 411–423. Springer, 2016.

[70] Lu Ma, Juan M Falquez, Steve McGuire et Gabe Sibley : Large scale dense visual inertial slam. *In Field and Service Robotics*, pages 141–155. Springer, 2016.

[71] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley *et al.* : Introducing slambench, a performance and accuracy benchmarking methodology for slam. *In 2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5783–5790. IEEE, 2015.

[72] A.J. Davison : Real-time simultaneous localisation and mapping with a single camera. *In IEEE International Conference on Computer Vision*, pages 1403–1410, 2003.

[73] Georg Klein et David Murray : Parallel tracking and mapping for small AR workspaces. *In International Symposium on Mixed and Augmented Realit*, Nara, Japan, 2007.

[74] D. Botero, A. Gonzalez, M. Devy *et al.* : Architecture embarquée pour le slam monoculaire. *In Reconnaissance des Formes et Intelligence Artificielle*, 2012.

[75] V. Bonato, E. Marques et G.A. Constantinides : A floating-point extended kalman filter implementation for autonomous mobile robots. *Journal of Signal Processing Systems*, 56(1):41–50, 2009.

[76] A. Ratter, C. Sammut et M. McGill : Gpu accelerated graph slam and occupancy voxel based icp for encoder-free mobile robots. *In 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 540–547, Nov 2013.

[77] Mohd Yamani Idna Idris, Noorzaily Mohamed Noor, Emran Mohd Tamil, Zaidi Razak et Hamzah Arof : Parallel matrix multiplication design for monocular slam. *In 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, pages 492–497. IEEE, 2010.

[78] Georg Klein et David Murray : Parallel tracking and mapping for small ar workspaces. *In Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, pages 225–234. IEEE, 2007.

[79] Christopher M Gifford, Russell Webb, James Bley, Daniel Leung, Mark Calnon, Joe Makarewicz, Bryan Banz et Arvin Agah : Low-cost multi-robot exploration and mapping. *In 2008 IEEE International Conference on Technologies for Practical Robot Applications*, pages 74–79. IEEE, 2008.

[80] V. Bonato, E. Marques et G.A. Constantinides : A floating-point extended kalman filter implementation for autonomous mobile robots. *Journal of Signal Processing Systems*, 56(1):41–50, 2009.

[81] S. Magnenat, V. Longchamp, M. Bonani, P. Rétornaz, P. Germano, H. Bleuler et F. Mondada : Affordable slam through the co-design of hardware and methodology. *In IEEE International Conference on Robotics and Automation*, pages 5395–5401, 2010.

[82] D. Botero, A. Gonzalez, M. Devy *et al.* : Architecture embarquée pour le slam monoculaire. *In Reconnaissance des Formes et Intelligence Artificielle*, 2012.

[83] Alan Mathison Turing : On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[84] M. Gondran et M. Minoux : *Graphes et algorithmes*. Eyrolles, 1969.

[85] Charles Antony Richard Hoare *et al.* : *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[86] Jack Bonnell Dennis : Data flow supercomputers. *Computer*, (11):48–56, 1980.

[87] A.F. Dias : Contribution à l'implantation optimisée d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-fpga à l'aide d'une méthodologie d'adéquation algorithme architecture. Mémoire de D.E.A., Université de Paris-Sud, Orsay, France, 2000.

[88] Y. Sorel C. Lavarenne : Modèle unifié pour la conception conjointe logiciel-matériel. *In Journées Adéquation Algorithme Architecture en Traitement du Signal et Images, CNES, Toulouse France*, January 1996.

[89] C Lavarenne et Y Sorel : Modele unifié pour la conception conjointe logiciel-matériel a unified model for software-hardware co-design. *Traitement du signal*, 14(6):569–578, 1997.

[90] AILTON F Dias, MOHAMED AKIL, CHRISTOPHE LAVARENNE et YVES SOREL : Adéquation algorithme architecture appliquée aux circuits reconfigurables. *Actes des Quatrièmes Journées AAA en Traitement du Signal et des Images*, 1998.

[91] Albert Y Zomaya *et al.* : *Parallel and distributed computing handbook*, volume 204. McGraw-Hill New York, 1996.

[92] Steve Heath : *Microprocessor Architectures : RISC, CISC and DSP.* Elsevier, 2014.

[93] Thierry GRANDPIERRE : *Modélisation d'architectures parralléles hétérogènes pour la génération automatique d'exécutifs distribuées temps réel optimisées.* Thèse de doctorat, Université de Paris-Sud U.F.R. SCIENTIFIQUE D'ORSAY.

[94] Kurt Jensen : Eatcs monographs on theoretical computer science. *EACTS-Monographs on Theoretical Computer Science*, 1992.

[95] F Gécseg : Products of automata eatcs monographs on theoretical computer science. *EATCS Monographs on Theoretical Computer Science*, 7, 1986.

[96] Ailton Dias : *Contribution l'implantation optimise d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-FPGA l'aide d'une mthodologie d'adquation algorithme-architecture.* Thèse de doctorat, Universit de Paris Sud, 2000.

[97] B. Vincke : *Architectures for simultaneous localization and mapping Systems.* Thèse de doctorat, 2012.

[98] Andrew Howard et Nicholas Roy : The robotics data set repository (radish). 2003.

[99] Computational Vision Group : Pets : Performance evaluation of tracking and surveillance http ://www.cvg.reading.ac.uk. 2006.

[100] D. Scharstein et R. Szeliski : Stereo vision research page : http ://www.middlebury.edu/stereo. 2006.

[101] Giulio Fontana Matteo Matteucci Domenico Giorgio Sorrenti Andrea Bonarini, Wolfram Burgard et Juan Domingo Tardos : Rawseeds : Robotics advancement through web-publishing of sensorial and elaborated extensive data sets. *In In proceedings of IROS'06 Workshop on Benchmarks in Robotics Research*, 2006.

[102] Alessandro Giusti Daniele Marzorati Matteo Matteucci Davide Migliore Davide Rizzi Domenico G. Sorrenti Pierluigi Taddei Simone Ceriani, Giulio Fontana : Rawseeds ground truth collection systems for indoor self-localization and mapping. *Autonomous Robots*, 27(4):353–371, 2009.

[103] Mike Smith, Ian Baldwin, Winston Churchill, Rohan Paul et Paul Newman : The new college vision and laser data set. *The International Journal of Robotics Research*, 28(5):595–599, 2009.

[104] Andreas Geiger, Philip Lenz, Christoph Stiller et Raquel Urtasun : Vision meets robotics : The kitti dataset. *The International Journal of Robotics Research*, page 0278364913491297, 2013.

[105] Bin Lu, Xin Wu, H. Figueroa et A Monti : A low-cost real-time hardware-in-the-loop testing approach of power electronics controls. *Industrial Electronics, IEEE Transactions on*, 54(2):919–931, April 2007. ISSN 0278-0046.

[106] Taehun Hwang, Jihoon Roh, Kihong Park, Jeongho Hwang, Kyu Hoon Lee, Kang-Won Lee, Soo-Jin Lee et Young-Jun Kim : Development of hils systems for active brake control systems. *In SICE-ICASE, 2006. International Joint Conference*, pages 4404–4408, Oct 2006.

[107] Nvidia's next generation cuda compute architecture : Kepler tm gk110/210. `http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`. Accessed : Novembre 2016.

[108] Deshanand Singh : Implementing fpga design with the opencl standard, tech. report, altera corporation. November 2011.

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

[110] D.G. Lowe : Distinctive image features from scale-invariant keypoints. *Computer Vision*, 60 (2):91–110, 2004.

[111] C. Harris et M. Stephens : A combined corner and edge detection. *In The Fourth Alvey Vision Conference*, pages 147–151, 1988.

[112] J. Shi et C. Tomasi : Good features to track. *In IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[113] E. Rosten, R. Porter et T. Drummond : Faster and better : A machine learning approach to corner detection. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, pages 105–119, 2009.

[114] C. Schröter, H.J. Böhme et H.M. Gross : Memory-efficient gridmaps in rao-blackwellized particle filters for slam using sonar range sensors. *In European Conf. on Mobile Robots*, pages 138–143, 2007.

[115] C. Brenneke, O. Wulf et B. Wagner : Using 3d laser range data for slam in outdoor environments. *In IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 188–193, 2003.

[116] M. Montemerlo, S. Thrun, D. Koller et B. Wegbreit : FastSLAM 2.0 : An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. *In International Joint Conference on Artificial Intelligence*, pages 1151–1156, 2003.

[117] M. Montemerlo, S. Thrun, D. Koller et B. Wegbreit : FastSLAM : A factored solution to the simultaneous localization and mapping problem. *In National Conference on Artificial Intelligence*, pages 593–598, 2002.

[118] Hauke Strasdat, Andrew J Davison, JMM Montiel et Kurt Konolige : Double window optimisation for constant time visual slam. *In 2011 International Conference on Computer Vision*, pages 2352–2359. IEEE, 2011.

[119] Hauke Strasdat, JMM Montiel et Andrew J Davison : Scale drift-aware large scale monocular slam. *Robotics : Science and Systems VI*, 2010.

[120] Dorian Gálvez-López et Juan D Tardos : Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[121] Self organizing continous attractor networks et path integration : two-dimensional models of place cells : Hippocampal models for simultaneous localisation and mapping on an autonomous robot. *Computational Neural Systems.*, 13:429–446, 2002.

[122] A. Arleo et W. Gerstner : Modeling rodent head-direction cells and place cells for spatial learning in biomimetic robotics. *presented at Sixth International Conference on the Simulation of Adaptive Behaviour (Animals to Animats6)*, 2000.

[123] M. J. Milford et G. Wyeth : Hippocampal models for simultaneous localisation and mapping on an autonomous robot. *presented at Australian Conference on Robotics and Automation, Brisbane, Australia*, 2003.

[124] D. Prasser et G. Wyeth : Hippocampal models for simultaneous localisation and mapping on an autonomous robot. *in Proceedings of the 2003 IEEE International Conference on Robotics and Automation.*, 2003.

[125] Liang Zhao, Shoudong Huang et Gamini Dissanayake : Linear slam : A linear solution to the feature-based and pose graph slam based on submap joining. *In Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 24–30. IEEE, 2013.

[126] Shoudong Huang, Zhan Wang, Gamini Dissanayake et Udo Frese : Iterated slsjf : A sparse local submap joining algorithm with improved consistency. 2008.

[127] E. Eade et Tom Drummond : Scalable monocular slam. *In IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 469–476, June 2006.

[128] Jan-Michael Frahm, Pierre Fite-Georgel, David Gallup, Tim Johnson, Rahul Raguram, Changchang Wu, Yi-Hung Jen, Enrique Dunn, Brian Clipp, Svetlana Lazebnik *et al.* : Building rome on a cloudless day. *In European Conference on Computer Vision*, pages 368–381. Springer, 2010.

[129] Siddharth Choudhary, Shubham Gupta et PJ Narayanan : Practical time bundle adjustment for 3d reconstruction on the gpu. *In Trends and Topics in Computer Vision*, pages 423–435. Springer, 2010.

[130] Lina María Paz, José E Guivant, Juan D Tardós et José Neira : Data association in o (n) for divide and conquer slam. *In Robotics : Science and Systems*, volume 3, page 281, 2007.

[131] Jose E Guivant et Eduardo Mario Nebot : Optimization of the simultaneous localization and map-building algorithm for real-time implementation. *IEEE transactions on Robotics and Automation*, 17(3):242–257, 2001.

[132] Arian Maghazeh, Unmesh D Bordoloi, Petru Eles et Zebo Peng : General purpose computing on low-power embedded gpus : Has it come of age ? *In Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 1–10, Agios Konstantinos, Greece, 2013. IEEE.

[133] Peter Hellekalek : Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5):485–505, 1998.

[134] A De Matteis et Simonetta Pagnutti : Parallelization of random number generators and long-range correlations. *Numerische Mathematik*, 53(5):595–608, 1988.

[135] Chih Jeng Kenneth Tan : The plfg parallel pseudo-random number generator. *Future Generation Computer Systems*, 18(5):693–698, 2002.

[136] Peter Hellekalek : Don't trust parallel monte carlo ! *In ACM SIGSIM Simulation Digest*, volume 28, pages 82–89, Banff, Alberta, Canada, 1998. IEEE Computer Society.

[137] Myles Sussman, William Crutchfield et Matthew Papakipos : Pseudorandom number generation on the gpu. *In Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 87–94. ACM, 2006.

[138] M. Njiki, S. Bouaziz, A. Elouardi, O. Casula et O. Roy : A multi-fpga implementation of real-time reconstruction using total focusing method. *In IEEE 3rd Annual International Conference on Cyber Technology in Automation, Control and Intelligent Systems (CYBER)*, pages 468–473, May 2013.

[139] A. Vicard : *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués*. Thèse de doctorat, Université Paris XIII, Juillet 1999.

[140] M. Kraft, A. Schmidt et A. Kasinski : High-speed image feature detection using fpga implementation of fast algorithm. *In International Conference on Computer Vision Theory and Applications*, pages 174–179, 2008.

[141] A. Schmidt, M. Kraft et A. Kasiński : An evaluation of image feature detectors and descriptors for robot navigation. *Computer Vision and Graphics*, pages 251–259, 2010.

[142] J.M.M. Montiel, J. Civera et A.J. Davison : Unified inverse depth parametrization for monocular slam. *In Int. Conf. on Robotics : Science and Systems*, pages 16–19, 2006.

[143] N. Sunderhauf, S. Lange et P. Protzel : Using the unscented kalman filter in mono-slam with inverse depth parametrization for autonomous airship control. *In IEEE International Workshop on Safety, Security and Rescue Robotics*, pages 1–6, 2007.

[144] P. Piniés, T. Lupton, S. Sukkarieh et J.D. Tardós : Inertial aiding of inverse depth slam using a monocular camera. *In IEEE International Conference on Robotics and Automation*, pages 2797–2802, 2007.

[145] J.M.M. Montiel, J. Civera et A.J. Davison : Unified inverse depth parametrization for monocular SLAM. *In International Conference on Robotics : Science and Systems*, pages 16–19, 2006.

[146] William G. Madow et Lillian H. Madow : On the theory of systematic sampling, i. *The Annals of Mathematical Statistics*, 15(1):1–24, 03 1944.

[147] Jinxia Yu, Wenjing Liu et Y. Tang : Improved particle filter algorithms based on partial systematic resampling. *In Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, pages 483–487, Oct 2010.

[148] Qifeng Gan, J.M.P. Langlois et Y. Savaria : A reformulated systematic resampling algorithm for particle filters and its parallel implementation in an application-specific instruction-set processor. *In Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, pages 1415–1418, Aug 2013.

Chapter 7

Annexes

7.1 Annexe A

7.1.1 Codeur Optique

La plupart des encodeurs pour robots mobiles utilisent des capteurs optiques. L'idée est de placer un disque alternant des zones transparentes et opaques devant un capteur de lumière et de rentre le disque solidaire de l'axe de rotation de la roue. Lorsque le disque tourne, les segments opaques bloquent la lumière alors que les segments transparents la laissent passer. Ceci génère des impulsions d'onde carrée qui peuvent ensuite être interprétées comme position ou mouvement. L'encodeur rotatif incrémentale comporte deux pistes de code dont les secteurs sont décalés de 90 degrés d'une piste à l'autre. Ces deux pistes génèrent deux signaux de sortie montrés dans la figure 7.1. Si le premier signal devance le second alors le disque tourne dans le sens des aiguilles d'une montre et dans l'autre sens dans le cas contraire. Par conséquent, en mesurant à la fois le nombre d'impulsions et les phases relatives des deux signaux on peut mesurer la position et la direction de la rotation des roues du robot mobiles.

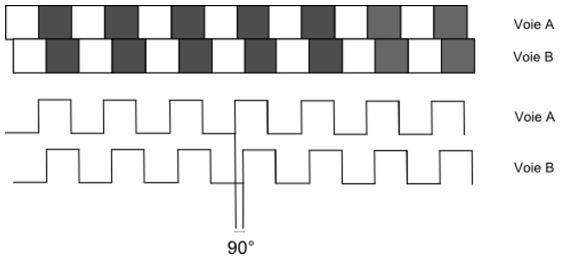


Figure 7.1: *Signaux de sorties émis par un odomètre*

7.1.2 Télémètre Laser

Un rayon laser est projeté sur une cible qui renvoie à son tour le rayon lumineux. Pour mesurer la distance, on calcule l'angle de réflexion avec lequel revient le faisceau laser. Autrement dit, le laser émet une nappe de lumière dans le proche infrarouge (par balayage très rapide du faisceau). La lumière se réfléchit sur l'obstacle. Sa distance par rapport au télémètre est calculée par la méthode de triangulation. La lumière réfléchie atteint un élément récepteur sous un angle α qui est fonction de la distance d. La distance d entre l'objet de mesure et le télémètre est calculée dans le capteur à partir de la position du point lumineux sur l'élément récepteur et à partir de la distance b séparant

l'émetteur du récepteur. La distance d est donnée par la formule : $d = b * \tan(\alpha)$, où b est la distance séparant l'émetteur du récepteur, et α l'angle de retour du faisceau laser (figure 7.2).

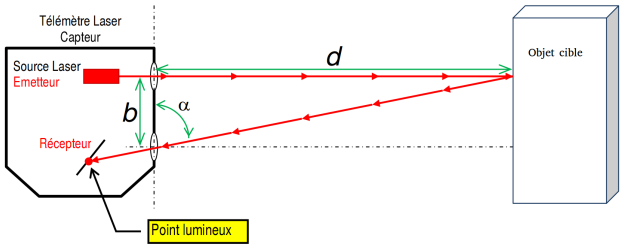


FIGURE 7.2: *principe de fonctionnement du télémètre laser*

7.2 Annexe B

7.2.1 Exemple de factorisation algorithmique

Le produit d'une matrice $M \in \mathbf{R}^m \times \mathbf{R}^m$ par un vecteur $V \in \mathbf{R}^n$ donne un vecteur $C \in \mathbf{R}^m$, et peut s'écrire sous la forme factorisée suivante :

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = C = [c_i]_{i=1}^{m} = \left[\Sigma_{j=1}^{n} m_{ij} v_j\right]_{i=1}^{m} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Dans le cas ou $m = n = 3$, cette équation peut être défactorisée partiellement par ligne :

$$C = [m_{i1}v_1 + m_{i2}v_2 + m_{i3}v_3]_{i=1}^{3}$$

Soit totalement, dans ce cas on peut écrire :

$$C = \begin{bmatrix} m_{11}v_1 + m_{12}v_2 + m_{13}v_3 \\ m_{21}v_1 + m_{22}v_2 + m_{23}v_3 \\ m_{31}v_1 + m_{32}v_2 + m_{33}v_3 \end{bmatrix} \tag{7.1}$$

Le graphe simplifié qui correspond au produit matrice-vecteur est illustré dans la figure 7.3-a.En effet, le produit d'une matrice 3×3 par un vecteur de dimension 3 peut se décomposer en 3 produits scalaires qui peuvent se décomposer chacun en une somme de 3 produits (7.1). La figure 7.3-b présente le sous-graphe du graphe (MxV), qui calcule un des produit matrice vecteur correspondant à trois répétitions du produit scalaires V. La figure 7.4 présente le sous-graphe du graphe (V), qui effectue un produit scalaire correspondant à trois répétitions des opérations multiplication-accumulation. Ce sous-graphe utilise trois dépendances inter-répétions finies pour effectuer l'accumulation à partir du résultat de la somme calculée à la répétition précédente.

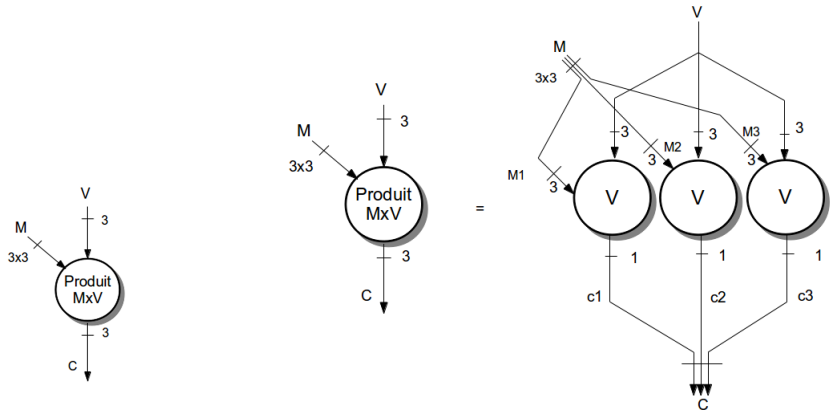


FIGURE 7.3: *Graphe composé du produit matrice MxV*

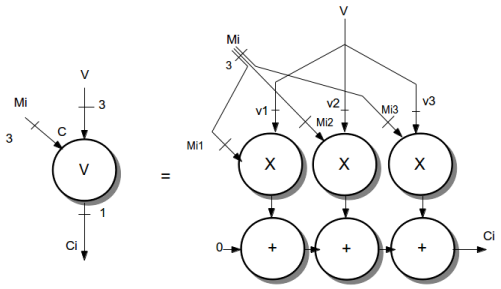


FIGURE 7.4: *Graphe décomposé du produit scalaire V, qu'il s'agit de trois répétitions finies de de multiplication et de sommation*

La figure 7.5 représente la spécification factorisée de l'algorithme faisant le produit matrice-vecteur MxV vu ci-dessus. En considérant qu'on effectue tout d'abord les produits scalaires entre les lignes de la matrice M et le vecteur V, et ensuite on collecte les résultats de ces produits scalaires sous la forme d'un vecteur qui correspond au vecteur C résultant du produit MxV. La première factorisation qu'on peut faire est illustré dans la figure 7.5-a, qui correspond au graphe factorisé du produit MxV illustré dans la figure 7.4. Cette factorisation fait apparaître trois sommets spéciaux (D pour "Diffusion", F pour "Fork", et J pour "Join") qui délimitent la frontière. la figure 7.5-b montre la factorisation du produit scalaire V illustré dans la figure 7.4. Quant à cette factorisation, elle fait apparaitre de nouveau sommet frontière I (correspondant à "Iterate"). Soit $\mathbf{O}_{diff}$, $\mathbf{O}_{Fork}$, $\mathbf{O}_{Join}$, $\mathbf{O}_{Iter}$ des sous-ensemble de $\mathbf{O}$. Chaque sommet frontière spécifie ainsi une manière différente de traverser la frontière de factorisation, distribuée comme le suivant par rapport à l'exemple produit matrice-vecteur MxV.

— Le sommet D, entre en diffusant le vecteur V à tous les produits scalaires.
— Le sommet F entre en défactorisant le groupe d'arcs d'entrée
— Le sommet J sort en factorisant le groupe d'arcs de sortie
— le sommet I entre et sort (par 0 et c_i) en factorisant le groupe d'arcs inter-motifs.

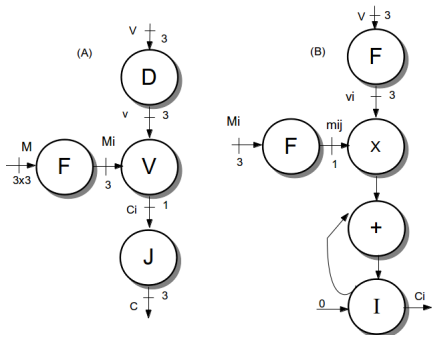


Figure 7.5: *(A) Graphe du produit matrice-vecteur factorisé à l'aide des sommets frontières de factorisation, (B) Graphe du produit scalaire factorisé à l'aide des sommets frontières de factorisation*

Ainsi, le graphe résultant (figure 7.6) possédé deux frontière de factorisation (F_1 et F_2), correspondant au deux étapes du produit matrice-vecteur (produit scalaire + multiplication-sommation). La factorisation du graphe ne change en rien sa sémantique opératoire, elle ne fait que de réduire la taille de la spécification du graphe et mettre en évidence ses parties régulières dont la connaissance ouvre la voie au possibilités d'optimisation. Les sommets frontières ne sont que des délimiteurs d'une syntaxe graphique. De la même manière que pour deux paires de parenthèse, deux frontière de factorisation ne peuvent être que soit l'une dans l'autre. soit l'une à coté de l'autre, elle ne peuvent être en relation d'intersection. Un opération à l'intérieur d'une frontière de données différentes se réalise directement avec un seul opérateur utilisé itérativement, autant de fois qu'il y a d'opération dans le groupe factorisé. Chaque groupe factorisé de données doit être multiplié correspondant que sommets frontière doivent réaliser le multiplexage sauf, l'opérateur "D" qui se contente de fourni la même valeur à chaque itération.

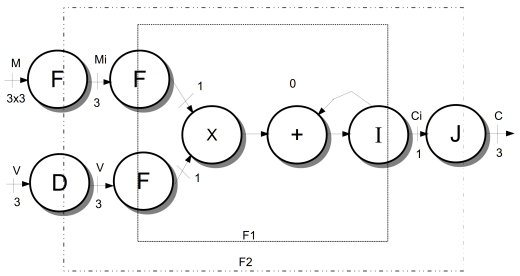


Figure 7.6: *Graphe finale factorisé du produit matrice-vecteur*

7.2.2 Exemple de modélisation architecturale

La figure 7.7 présente le graphe d'architecture correspondant à la modélisation détaillée du processeur de traitement du signal TMS320C40 conçu par Texas Instrument. Ici comme toutes les connexions sont bi-directionelles, chaque couple de flèche de sens opposées sont représentées par un seul segment. Le CPU, son contrôleur mémoire et son unité de calcul sont modélisés par un opérateur. Comme il est capable d'accéder simultanément à deux mémoires internes et/ou externes modélisées par des sommets RAM (R_0 et R_1pour les mémoires internes, R_{loc} et R_{glob} pour les mémoires externes éventuelles), l'opérateur est connecté à deux bus/mux/démux (b_7 et b_8) qui sélectionnent les mémoires. Comme ces mémoires sont aussi accessibles par chaque canal du DMA modélisé par un communica-

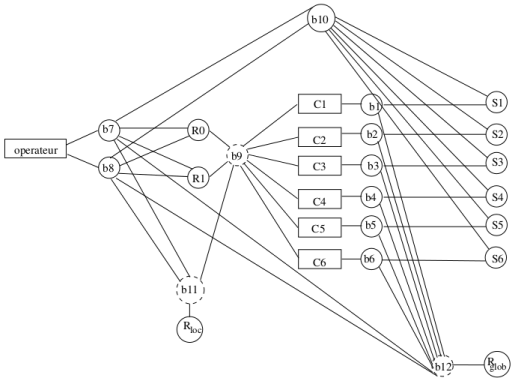


Figure 7.7: *Graphe d'architecture modélisé par l'approche A3*

teur (C_1 à C_6), chaque communicateur est connecté à ces mémoires internes par intermédiaire d'un bus/mux/démux/arbitre (b_9) qui arbitre l'accès entre tous les communicateurs. Chaque port de communication est modélisé par une SMA. Chaque SAM étant à la fois accessible par un canal DMA ou par le CPU, elles sont connectées d'une part à un communicateur et d'autre part à l'opérateur. Le bus/mux/démux b_{10} permet de sélectionner laquelle des SAM est accédée par l'opérateur. Les deux ports de mémoires externes permenttent à l'opérateur et au communicateurs d'accéder aux deux mémoires externes, il y a donc arbitrage qui est modélisé par deux sommets bus/muux/démux/arbitre (b_{11} et b_{12}) entre les RAM externes, l'opérateur et les communicateurs. Les bus/mux/démux b_1 à b_6 modélise la capacité, pour chaque communicateur, d'accéder soit à une SLAM, soit à la mémoire externe.

7.2.3 Formalisme mathématique du graphe

Comme mentionné précédemment, les sommets du graphe sont les opérations de calcules de l'algorithme ou bien les schémas blocs. Les arcs sont les dépendances de données entre opérations, également appelées dépendance de données inter-opérations. Une opération peut s'exécuter lorsque ses données d'entrée sont présentes. Elle consomme ses données d'entrée et produit des données en sortie qui sont ensuite utilisées par ses successeurs.

Soit G_{algo}, le graphe de l'algorithme définit par le couplet $(\mathbf{O}, \mathbf{D})$ [93]

avec :

— $\mathbf{O}$ est l'ensemble des opérations, appelées "opération de calcule" du dit graphe G_{algo} tel que, $\mathbf{O} = \{o_i\}_{1 \leq i \leq n}$.

— $\mathbf{D}$ est l'ensemble des arcs du graphe de l'algorithme, appelés dépendances de données inter-opérations tel que, $\mathbf{D} = \{d_i\}_{1 \leq i \leq k}$

— d_i est donnée par, $d_i = \left(o_{i_1}, \left\{o_{i_2}, \ldots, o_{i_{k(i)}}\right\}_{2 \leq k(i) \leq n}\right)$

On associe à l'ensemble des dépendances $\mathbf{D}$, la fonction γ^{-1}qui, à chaque dépendance d_i associe sont opération émettrice $\gamma^{-1}(d_i)$:

$$\begin{gathered} \gamma^{-1} : \mathbf{D} \rightarrow \mathbf{O} \\ d_i \rightarrow \gamma^{-1}(d_i) = o_{i_1}/d_i = \left(o_{i_1}, \left\{o_{i_2}, \ldots, o_{i_{k(i)}}\right\}_{2 \leq k(i) \leq n}\right) \end{gathered} \tag{7.2}$$

On définit aussi la fonction γ qui, à chaque dépendance d_i, associe l'ensemble des opérations réceptrice appelées aussi consommatrices :

$$\begin{gathered}\gamma : \mathbf{D} \rightarrow \mathbf{O} \\ d_i \rightarrow \gamma\left(d_i\right) = \left\{o_{i_j}\right\}_{2 \leq j \leq n} / d_i = \left(o_{i_1}, \left\{o_{i_2}, \ldots, o_{i_{k(i)}}\right\}_{2 \leq k(i) \leq n}\right)\end{gathered} \tag{7.3}$$

Le graphe de l'algorithme est sans circuit. Il n'existe pas de chemin ayant à la fois même origine et même destination. Chaque dépendance de données a un et un seul émetteur et au moins un récepteur, ainsi :

$$\forall d_i \in \mathbf{D} / \gamma^{-1}\left(d_i\right) = 1, \gamma\left(d_i\right) \geq 1 \tag{7.4}$$

Ainsi, associé au graphe G_{algo}, ont définit les notions de successeur et de descendant, ainsi que de prédécesseur et d'ancêtre d'une opération de graphe :

— Successeur [84] : Soit $\Gamma\left(o_i\right)$, l'ensemble des successeurs d'une opération o_i donné par :

$$\Gamma\left(o_i\right) = \left\{o_j \in \mathbf{O} / \exists d_k \in \mathbf{D}\right\} \rightarrow o_i = \gamma^{-1}\left(d_i\right) / o_j \subseteq \gamma\left(d_k\right) \tag{7.5}$$

— Descendants [84] : Soit $\hat{\Gamma}\left(o_i\right)$, l'ensemble des descendants d'une opération o_i. Il s'agit de la fermeture transitive de l'ensemble des successeurs, autrement dit :

$$\hat{\Gamma}\left(o_i\right) = \cup_{k=1}^{n} \Gamma^k\left(o_i\right) \tag{7.6}$$

Les opération qui n'ont pas des successeurs sont appelées des *"sorties"*. L'ensemble des sorties d'un graphe est défini par :

$$Sorties\left(G_{al}\right) = \left\{o_i \in \mathbf{O} / \Gamma\left(o_i\right) = \emptyset\right\} \tag{7.7}$$

— Prédécesseurs [84] : Soit $\Gamma^{-1}\left(o_i\right)$ l'ensemble des prédécesseurs d'une opération o_i donnée par

$$\Gamma^{-1}\left(o_i\right) = \left\{o_j \in \mathbf{O} / \exists d_k \in \mathbf{D}\right\} \rightarrow o_j = \gamma^{-1}\left(d_i\right) / o_i \subseteq \gamma\left(d_k\right) \tag{7.8}$$

— Ancêtres [84]: soit $\hat{\Gamma}^{-1}\left(o_i\right)$ l'ensemble des ancêtres d'une opération o_i. Il s'agit de la fermeture transisitive de l'ensemble des prédécesseurs, autrement dit:

$$\hat{\Gamma}^{-1}\left(o_i\right) = \cup_{k=1}^{n} \left(\Gamma^{-1}\right)^k\left(o_i\right) \tag{7.9}$$

Les opération qui n'ont pas des prédécesseurs sont appelées des *"entrées"*. L'ensemble des entrées d'un graphe est défini par :

$$Entrées\left(G_{al}\right) = \left\{o_i \in \mathbf{O} / \Gamma^{-1}\left(o_i\right) = \emptyset\right\} \tag{7.10}$$

7.2.4 Formalisation mathématique de la distribution

comme décrite dans [93] est résumée comme le suivant :

Soit Π l'application qui, à chaque opération de calcul, associe l'opérateur de calcul sur lequel elle est distribuée :

$$\Pi \rightarrow \{\mathbf{O} \rightarrow S_{cal}\}$$
$$o_i \rightarrow \Pi(o_i) = p_j$$

Soit Π^{-1}, l'application réciproque de Π, qui associe à chaque opérateur, l'ensemble des opérations de calcul distribuées sur cet opérateur :

$$\Pi^{-1} \rightarrow S_{cal} \rightarrow P(\mathbf{O})$$
$$p_j \rightarrow \Pi^{-1}(p_j) = \{o_i \in \mathbf{O}/\mathbf{\Pi}(o_i) = p_j\}$$

Ainsi, $\Pi^{-1}(p)$ correspond à l'ensemble, noté $\mathbf{O}_p$, des opérations distribuées sur l'opérateur p. Ce partitionnement donne lieu à deux types de dépendances de données :

— les dépendances de données intra-partition D_p(appelées aussi intra-processeur) sont définies par [139] :

$$D_p \subseteq \mathbf{O}_p \times P(\mathbf{O}_p) \, \forall p \in S_{cal}$$

— les dépendances de données inter-partitions (appelées aussi inter-processeurs), notées D_r, elles correspondent aux dépendances de données entre opérations distribuées sur des opérateurs différents. Elles vont donner lieu à un transfert de données sur chaque média qui compose la route r, joignant les opérateurs émetteur et récepteur.

$$D_r = \cup(\mathbf{O}_{p_k} \times P(\cup_{p_l \in r, p_l \neq p_k} \mathbf{O}_{p_l}))$$

— On note D_r l'ensemble des dépendances entre opérations dépendantes et distribuées sur des opérations différents. A partire d'un graphe d'algorithme et d'un graphe d'architecture routé, le partitionnement permet de construire les graphes partitionnés G_{part}.

$$(G_{al}, G_{ar}) \rightarrow (G_{part}, G_{ar})(.)$$
$$((\mathbf{O}, \mathbf{D}), (S, Rd)) \rightarrow (\cup_{p \in S_{cal}} (\mathbf{O}_p, \mathbf{D}_p), \cup_{r \in R} D_r)$$

7.2.5 Formalisme mathématique de l'ordonnancement

Soit R_{ordo} la relation qui permet de renforcer l'ordre partiel noté $\preceq$ en un ordre totale noté $\prec$entre les opération de chaque partition. Cette relation correspond à l'allocation temporelle du graphe distribué, sur le graphe de l'architecture. La formalisation comme décrite dans [93] est résumée comme le suivant :

— on renforce donc l'ordre partiel $D_p(\preceq_p)$ de chaque partition associée à un opérateur de calcul d'un processeur p, en un ordre total $D_P(\prec_p)$ à l'aide de dépendances d'exécution sans données appelées arcs de précédence, notés $D_p^{''}$, entre les opérations $\mathbf{O}$ du graphe d'algorithme $\left(\overline{D}_p = D_p \cup D_p^{''}\right)$.

— de même on renforce l'ordre partiel $D_{p,c}^{'}(\preceq_{p,c})$ de chaque opérateur de communication c de chaque processeur p en un ordre total $\overline{D}_{p,c}^{'}(\preceq)$ à l'aide de précédences $D_{p,c}^{''}$ entre les opérations de communications $\mathbf{O}$ $\left(\overline{D}_{p,c} = D_{p,c}^{'} \cup D_{p,c}^{''}\right)$

— l'ordre partiel $\preceq_c$ donné par les dépendances de données D_p^* est inchangé.

Après ordonnancement, l'ensemble des dépendances du graphe distribué et ordonnancé appelé aussi graphe d'implantation est mathématiquement défini par :

$$D = \left(\cup_{p \in S_{cal}} \overline{D}_p\right) \cup \left(\cup_{p \in S_{cal}, c \in S_{com}} \overline{D}'_{p,c}\right) \cup D^*_p$$

7.3 Annexe C

7.3.1 Détection à partir d'un flux Laser

La figure 7.8 montre les résultats d'extraction d'amers. Les points en noire représentent le flux de données brutes fournit par un Laser. Les points en rouge représentent les amers extraits en utilisant l'algorithme d'extraction. L'algorithme calcule aussi la matrice de covariance résultante (les ellipses en bleu). On note que le nombre de points détectés dépend du seuil de segmentation γ. Plus le seuil est grand plus le nombre des amers extrait diminue. La figure 7.8-a montre une détection avec un seuil inférieur égale à $\gamma = 1$, où le nombre de points détectés est supérieur que le nombre de points détectés dans la figure 7.8-b avec un seuil γ égale à 8. La valeur du seuil est déterminée en fonction des résultats de localisation et de cartographie, plus le nombre des amers extraits et appariés est grand, plus les résultats sont consistants.

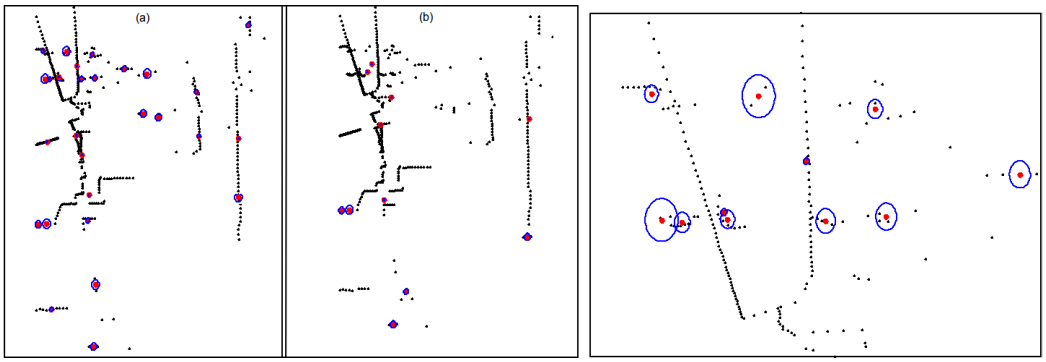


FIGURE 7.8: *Détection des amers à partir d'un flux de données Laser par segmentation et filtre d'information, les ellipses en bleu représente l'incertitude de détection des amers*

7.3.2 Détection des amers par FAST

Pour chaque pixel p de l'image, des tests sont réalisés sur les pixels appartenant à un cercle discrétisé de rayon 3 pixels. La figure 7.9 représente les 16 pixels en bleu qui sont testés pour détecter si le pixel p est un amer potentiel.

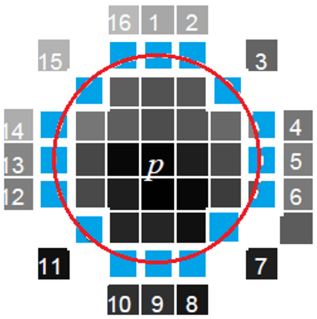


Figure 7.9: *Pixels considérés pour le test du pixel p*

Si n (classiquement $n = 9$) pixels voisins consécutif sont plus "lumineux" d'au moins ε que $I_p + \varepsilon$ ou plus "sombre" d'au moins ε que $I_p - \varepsilon$ où (I_p est l'intensité du pixel), alors le centre (pixel p) est considéré comme un point d'intérêt potentiel. En utilisant ce test, il est fréquent de détecter plusieurs points d'intérêts potentiels dans une zone très restreinte. Un filtrage est réalisé pour ne garder que les maximas locaux. Enfin, on supprime les non-maxima locaux dans un voisinage de 5 pixels. Les points détectés sont stables et que son détecteur est très rapide. La figure 7.10 présente le résultat de l'algorithme sur une image extraite d'une séquence de test. Plus le seuil est élevé, plus le nombre de points d'intérêts détecté est faible : la première image montre une détection FAST avec un seuil égale à 50, tendis que la deuxième montre une détection avec un seuil égale à 30. Le nombre de points détectés dans la deuxième image est plus grand que le nombre détectés dans la première image. Cette figure montre aussi les non maximas locaux (pixel en rouge) qui sont supprimés par le filtrage réalisé. Le détecteur FAST a été défini en 2008. Depuis, il est fréquemment utilisé par des algorithmes de SLAM monoculaire ou plus généralement de traitement d'images. Il a fait l'objet de deux implantations importantes. La première, utilisée par Rosten *et al.* [113], consiste à utiliser un algorithme de "Machine Learning" pour accélérer les calculs. Cette implantation est utilisée par nos algorithmes de SLAM, le code du programme étant disponible sur le site de l'auteur. La seconde est une implantation sur une architecture programmable type FPGA [140]. Dans notre implémentation on a visé la première implémentation optimisée par "Machine Learning", qui est adéquate avec l'architecture embarquée visée.

Figure 7.10: *Détection de points d'intérêts sur des images expérimentales avec un seuil =50, 30*

7.3.3 Appariement des amers

7.3.3.1 Calcule de la distance de Mahalanobis pour l'appariement dans le contexte SLAM Laser

En statistique, la distance de Mahalanobis est une mesure de distance pour déterminer la similarité entre une série de données. Contrairement à la distance euclidienne, La distance de Mahalanobis prend en compte la variance et la corrélation d'une distribution de données. La distance de Mahalanobis est souvent utilisée pour la détection de données aberrantes dans un jeu de données, ou bien pour déterminer la cohérence de données fournies par un capteur : cette distance est calculée entre les données reçues et celles prédites par un modèle (modèle extéroceptif). Cette distance est utili-

sée dans notre implémentation de l'algorithme de SLAM pour la mise en correspondance entre les observations extraites de flux de données du laser et les amers prédits par son modèle extéroceptifs. On considère par z_t et $\hat{z}_t$ respectivement l'observation et la prédiction de l'observation donnée par le modèle d'observation (1.5), tel que :

$$z_t = \begin{pmatrix} r \\ \phi \end{pmatrix}, \hat{z}_t = \begin{pmatrix} \hat{r} \\ \hat{\phi}_t \end{pmatrix}$$

La distance de Mahalanobis entre ces deux valeurs est donnée par :

$$\boldsymbol{\Psi} = (z_t - \hat{z}_t)^T \mathbf{Z}^{-1} (z_t - \hat{z}_t) \tag{7.11}$$

$\mathbf{Z}$ est la covariance d'innovation donnée par la relation :

$$\mathbf{Z} = \mathbf{H}_f \mathbf{C} \mathbf{H}_f + \mathbf{R} \tag{7.12}$$

$\mathbf{C}$ est la matrice de covariance de l'amer, et $\mathbf{R}$ est la matrice de covariance du bruit de mesure :

$$\mathbf{R} = \begin{pmatrix} \sigma_r & 0 \\ 0 & \sigma_\theta \end{pmatrix}$$

$\mathbf{H}_f$ est la jacobienne du modèle d'observation dérivée par rapport au cordonnées cartésienne de l'amer (x_f, y_f).

$$\mathbf{H}_f \rightarrow h_l (x_f, y_f) \in \mathbb{R}^2 \Longleftrightarrow (r, \phi) \in \mathbb{R}^2$$

La décision sur la nature de l'observation est difficile. En fait, si un amer est détecté il faut déterminer de quoi peut-il s'agir, si c'est un nouvel amer à rajouter dans la carte ou bien un amer déjà existant à corriger dans la phase d'estimation. En effet, on calcule la distance de Mahalanobis entre l'observation acquise et tous les amers dans la carte. Si les distances calculées sont tous supérieures à un seuil γ_n, l'amer est ajouté dans la carte et ses cordonnées initiales sont calculées. Si non l'observation est considérée mise en correspondance avec l'amer dont la distance est la plus petite. La valeur du seuil γ_n est obtenue par la loi inverse χ^2 où loi du χ^2 inverse (en anglais : inverse-chi-squared distribution).

L'incertitude sur la position du robot et le bruit de mesure affectent l'identification de l'amer observé. En effet, si le bruit de mesure est suffisamment grand, la distribution des observations des amers proches les uns aux autres chevauchent sensiblement. Ce chevauchement conduit a une ambiguïté dans l'identification des observations. Touts fausse association entre l'observation et l'amer engendre des erreurs dans la carte et la localisation du robot. Pour remédier à ce problème, on a utilisé une technique qui élimine les correspondances multiples. En effet, si un amer présent plusieurs observations comme correspondance, il ne sera pas utilisé dans la phase d'estimation et les observations seront ignorées. Autrement dit, seuls les candidats uniques sont pris en compte. Ce ce est assuré en appliquant le filtre standard du plus proche voisin (NNSF : The nearest neighbor standard filter). Les résultats d'appariement et du filtrage sont montrés dans les figures (7.11-a,7.11-b). La première figure montre un robot qui explore un environnement interne. Les amers (points rouges) sont extraits du flux de données laser, initialisés et rajoutés dans la carte avec une incertitude initiale (ellipse en

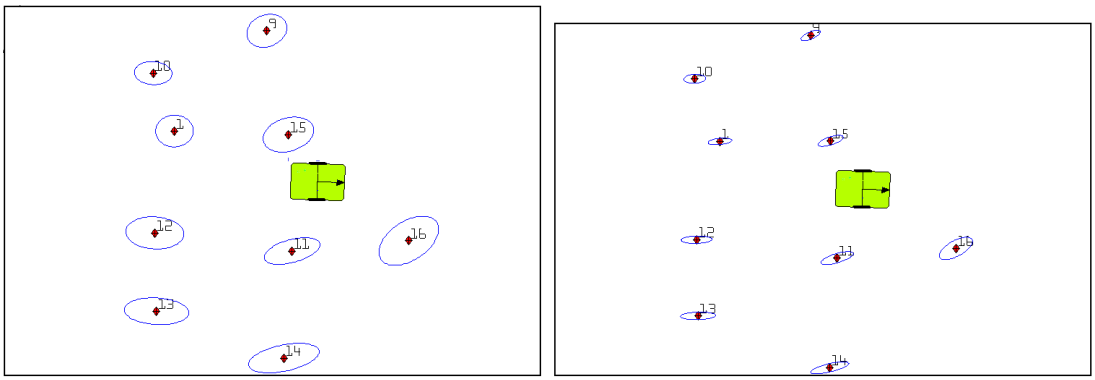


FIGURE 7.11: *amers extraits du flux de données Laser et initialisés par une incertitude*

bleu). Dans la deuxième figure, les mêmes amers sont redétectés et appariés avec les observations. La méthode du filtrage permet un appariement fiable et robuste des amers et enlève toute ambiguïté sur l'association. Ceci est approuvé dans la deuxième figure où l'incertitude des amers diminue et corrigée puisque les amers sont redétectés et appariés plusieurs fois.

7.3.3.2 Calcule de la distance ZMSSD (Zero-Mean Sum of Squared Distance) pour l'appariement dans le contexte SLAM visuel

La camera fournit une image de la scène contenant un ensemble des pixels qui constituent les amers potentiels détectés. Un descripteur est associé à chaque amer. il consiste soit en une fenêtre de pixels entourant le point soit en un détecteur plus complexe calculé à l'aide des pixels voisins. L'appariement est fait on calculant la disparité au niveau de chaque pixel voisin. Autrement dit, on calcule un taux de similarité entre le descripteur de l'amer dans l'image de référence et les descripteurs des pixels dans l'image cible pour trouver son homologue. L'amer est donc sera apparié avec le pixel correspondant à une distance qui minimise l'erreur associée et maximise la similarité. Le descripteur utilisé dans notre implémentation est une fenêtre d'image entourant le point d'intérêt de dimension 16×16 pixels, déja utilisé par Davison [72]. Il existe plusieurs distances utilisées pour la mise en correspondance de ce type de descripteurs à savoir : SAD (Sum of Absolute Differences), NCC (Normalized Cross Correlation), SSD (Sum of Squared Differences), LSSD (Locally scaled Sum of Squared Differences), SHD (Sum of Hamming Distances). D'après Schmidt et al. ([141]), SSD est la distance qui donne les meilleurs résultats d'appariement. Pour améliorer le comportement de la distance utilisée par rapport aux changements de luminosité, nous utiliserons une version légèrement modifiée du SSD, le ZMSSD (Zero Mean Sum of Squared Differences) donnée par :

$$\sum_{i,j} \left(I_{lmk}(i,j) - m_d - I_f(x+i, y+i) + m_f\right)^2 \tag{7.13}$$

Avec : m_d et m_f sont respectivement la moyenne du descripteur du pixel candidat et de la fenêtre d'image, I_f est l'intensité des pixels de la fenêtre d'image, x, y est la position du pixel détecté, i, j indices des pixels voisins du pixel candidat, I_{lmk} est l'intensité du pixel candidat. La figure 7.12 représente la mise en correspondance des points d'intérêts entre deux images consécutives dans un environnement interne. Les amers de la carte déjà cartographiés et initialisés (points rouges) sont projetés dans l'image acquise à l'instant courant (l'image de gauche). La projection définie une zone de recherche de correspondance (les carrés bleus). Cette zone de recherche est calculée précisément

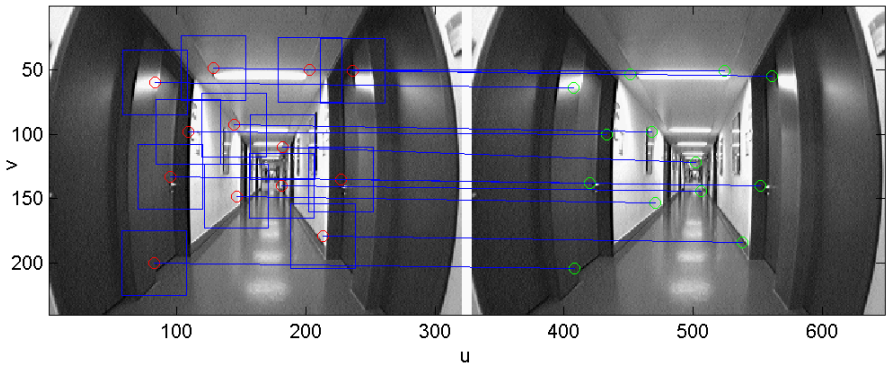


Figure 7.12: *Résultats de la mise en correspondance des amers*

dans notre implémentation de l'algorithme de SLAM, à partir de la carte et de la position du robot. Les observations (les points en verts) sont ensuite sélectionnées en calculant la distance ZMSSD entre les points détectés par FAST et les pixels dans la zone de recherche.

7.3.4 Initialisation des amers

7.3.4.1 Initialisation par inverse du modèle d'observation (Laser)

Les capteurs LASER permet la mesure de la profondeur des amers. Donc l'initialisation de l'amer est évidente et ne pose aucune difficulté. Ceci est obtenu en inversant le modèle d'observation démontré auparavant (1.5). Si l'observation (r, ϕ) s'agit d'un nouvel amer, ses coordonnées initiales $\boldsymbol{X}_f = (x_f, y_f)$ dans le repère du monde sont données par l'équation 7.14.

$$\mathbf{X}_f = \begin{pmatrix} x_f \\ y_f \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + r \begin{pmatrix} cos\,(\theta + \phi) \\ sin\,(\theta + \phi) \end{pmatrix} \tag{7.14}$$

avec $s_t\,(x, y, \theta)$ sont la position du robot.

7.3.4.2 Initialisation par inverse de profondeur (Camera)

La camera monoculaire est un capteur projectif incapable de mesurer la profondeur des amers dans la scène. Pour calculer la position initiale de l'amer, il faut estimer tout d'abord sa profondeur. Pour cela, il faut détecter l'amer dans plusieurs images et dans différentes vues pour fournir un parallaxe suffisant et estimer la profondeur. Il existe deux catégories de méthodes pour estimer la profondeur de l'amer et l'initialiser. La première catégorie c'est une initialisation retardée. En effet, il consiste à rajouter l'amer dans la carte après deux ou plusieurs observations dans différentes vues. L'angle (parallaxe) entre les deux observations dans les deux vues permet l'estimation de la profondeur de l'amer [46]. La deuxième méthode c'est une initialisation non retardée. En effet, une seule observation suffit pour rajouter l'amer dans la carte. Néanmoins, les coordonnées de l'amer sont modélisées par l'inverse de la profondeur ρ_f et non la profondeur d_f ($\rho_f = \frac{1}{d_f}$) [142]. Cette paramétrisation, communément utilisée dans les algorithmes de SLAM ([143, 144]), produit des équations d'observation ayant un plus haut degré de linéarité qu'une simple paramétrisation $(x_f y_f z_f)$. Dans notre implémentation nous avons utilisé l'initialisation par inverse de profondeur puisque il est plus adaptée avec le FastSLAM2.0 [127]. Les cordonnées de l'amer sont $(x_c, y_c, \theta_f, \varphi_f, \rho_f)$, illustrées dans la figure 7.13avec :

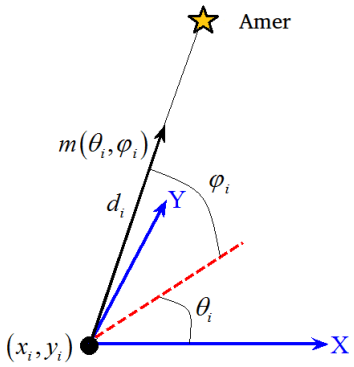


FIGURE 7.13: *Paramétrisation par inverse de profondeur*

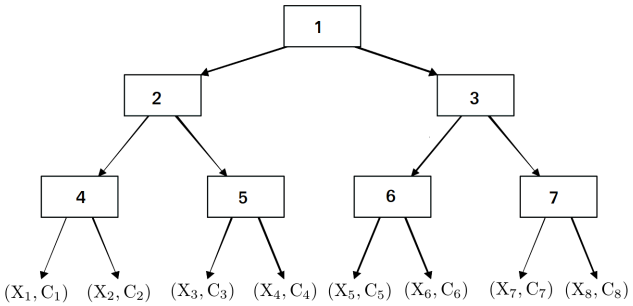


FIGURE 7.14: *Représentation par arbre binaire des amers*

— x_c et y_c sont les coordonnées initiales de la camera lors de la première observation.
— θ_f est l'azimuth
— ϕ_f est l'élévation
— ρ_f est l'inverse de la profondeur. fixée à 0.26 et $\sigma_\rho = 0.25$ comme proposé par Montiel *et al.* [145].

Les paramètres (équation 7.15 et 7.16) de l'amer sont initialisés avec une grande incertitude modélisée par une distribution gaussienne, corrigée par la suite (étape d'estimation) avec le filtre de Kalman étendu.

$$\theta_f = -\arctan\left(\frac{y_c - y_f}{x_c - x_f}\right) \tag{7.15}$$

$$\varphi_f = \arctan\left(\frac{z_f}{\sqrt{(x_f - x_c)^2 + (y_f - y_c)^2}}\right) \tag{7.16}$$

7.3.5 Représentation en arbre binaire

la figure 7.14 illustre le cas d'un huit amers pour une seule particule. Les paramètres d'amer $(\mathrm{X_n}, \mathrm{C_n})$ se situent aux feuilles de l'arbre. Chaque nœud non-feuille dans l'arbre contient des pointeurs de deux sous-arbres au maximum. Un sous-arbre peut être partagé entre plusieurs arbres de plusieurs particules

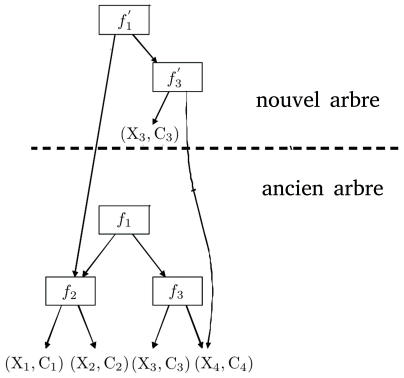


Figure 7.15: *Mise à jour l'arbre binaire (amer numéro 4)*

La figure 7.15 représente la mise à jour des paramètres de l'amer 4 sur la carte d'une particule donnée. Pour mettre à jour les paramètres d'un amer, l'algorithme va créer des feuilles intermédiaires ($f_1^{'}$,$f_3^{'}$). Ces dernières ont une branche qui pointe vers le reste de l'arbre qui ne change pas (respectivement f_2, $(\mathrm{X}_3, \mathrm{C}_3)$) et une branche qui pointe vers les nouvelles parties créées ($f_1^{'}$ vers $f_3^{'}$, $f_3^{'}$ vers $(\mathrm{X}_4, \mathrm{C}_4)$). La feuille finale $f_3^{'}$ contient les nouveaux paramètres de l'amer. La recopie partielle est indispensable à cause de l'étape de rééchantillonnage. En effet, après cette étape, plusieurs particules peuvent partager des sous-parties d'arbre. A la fin d'une mise à jour, il est important de désalouer l'ensemble des feuilles inutilisées. Cette implantation diminue considérablement les besoins en terme de mémoire pour stocker les paramètres des amers : une particule peut avoir des feuilles de l'arbre en commun avec d'autres particules. En effet, les cartes ne sont pas copiées, seul le pointeur vers l'arbre est mis à jour.

7.3.6 Méthode linéaire pour la fusion de deux cartes locales

Le problème de la fusion de deux cartes locales comme [126] l'a démontré, consiste à minimiser la fonction objective suivante :

$$f\left(\mathrm{X}^{G_{12}}\right) = ||e_1||^2_{I^{L_1}} + ||e_2||^2_{I^{L_2}}$$

$$\left| \begin{array}{c} R_1\left(t_0^{G_{12}} - t_1^{G_{12}}\right) - \hat{t}_0^{L_1} \\ r^{-1}\left(R_0 R_1^T\right) - \hat{\mathbf{r}}_0^{L_1} \\ R_1\left(\mathrm{X}_{\mathrm{f_1}}^{\mathrm{G_{12}}} - t_1^{G_{12}}\right) - \hat{\mathrm{X}}_{\mathrm{f1}}^{\mathrm{L_1}} \\ R_1\left(\mathrm{X}_{\mathrm{f_{12}}}^{\mathrm{G_{12}}} - t_1^{G_{12}}\right) - \hat{\mathrm{X}}_{\mathrm{f12}}^{\mathrm{L_1}} \end{array} \right|^2_{I^{L_1}} + \left| \begin{array}{c} -R_1 t_1^{G_{12}} - \hat{t}_2^{L_2} \\ r^{-1}\left(R_1^T\right) - \hat{r}_2^{L_2} \\ R_1\left(X_{f_2}^{G_{12}} - t_1^{G_{12}}\right) - \hat{\mathrm{X}}_{\mathrm{f_2}}^{\mathrm{L_2}} \\ R_1\left(X_{f_{12}}^{G_{12}} - t_1^{G_{12}}\right) - \hat{X}_{f_{12}}^{L_2} \end{array} \right|^2_{I^{L_2}} \tag{7.17}$$

avec : $||e_i||^2_{I^{L_i}} = e_i^T I^{L_i} e_i$ $(i = 1, 2)$ représente la norme pondérée du vecteur e_i avec une matrice d'information donnée I^{L_i}.

A partir de l'équation 7.17, on peut définir les variables suivantes :

$$
\begin{aligned}
\overline{\mathbf{t}}_0^{G_{12}} &= R_1\left(t_0^{G_{12}} - t_1^{G_{12}}\right) \\
\overline{\mathbf{r}}_0^{G_{12}} &= r^{-1}\left(R_0 R_1^T\right) \\
\overline{\mathbf{t}}_2^{G_{12}} &= -R_1 t_1^{G_{12}} \\
\overline{\mathbf{r}}_2^{G_{12}} &= r^{-1}\left(R_1^T\right) \\
\overline{\mathbf{X}}_{\mathrm{f1}}^{\mathrm{G_{12}}} &= R_1\left(\mathrm{X}_{\mathrm{f_1}}^{\mathrm{G_{12}}} - t_1^{G_{12}}\right) \\
\overline{\mathbf{X}}_{\mathrm{f2}}^{\mathrm{G_{12}}} &= R_1\left(X_{f_2}^{G_{12}} - t_1^{G_{12}}\right) \\
\overline{\mathbf{X}}_{\mathrm{f12}}^{\mathrm{G_{12}}} &= R_1\left(\mathrm{X}_{\mathrm{f_{12}}}^{\mathrm{G_{12}}} - t_1^{G_{12}}\right)
\end{aligned}
\tag{7.18}
$$

ce qui implique le nouveau vecteur d'état donné par :

$$
\overline{\mathbf{X}}^{G_{12}} = g\left(\mathbf{X}\right) = \left[\overline{\mathbf{t}}_0^{G_{12}}, \overline{\mathbf{r}}_0^{G_{12}}, \overline{\mathbf{t}}_2^{G_{12}}, \overline{\mathbf{r}}_2^{G_{12}}, \overline{\mathbf{X}}_{\mathrm{f1}}^{\mathrm{G_{12}}}, \overline{\mathbf{X}}_{f2}^{G_{12}}, \overline{\mathbf{X}}_{\mathrm{f12}}^{\mathrm{G_{12}}}\right] \tag{7.19}
$$

par conséquence le problème de moindre carré non-linéaire pour minimiser la fonction de l'équation (7.17) devient un problème de moindre carré linéaire pour minimiser la fonction objective suivante :

$$
\overline{f}\left(\overline{\mathbf{X}}^{G_{12}}\right) = \left|\begin{array}{c} \overline{\mathbf{t}}_0^{G_{12}} - \hat{\mathbf{t}}_0^{L_1} \\ \overline{\mathbf{r}}_0^{G_{12}} - \hat{\mathbf{r}}_0^{L_1} \\ \overline{\mathbf{X}}_{\mathrm{f1}}^{G_{12}} - \hat{\mathbf{X}}_{\mathrm{f1}}^{L_1} \\ \overline{\mathbf{X}}_{\mathrm{f12}}^{G_{12}} - \hat{\mathbf{X}}_{\mathrm{f_{12}}}^{\mathrm{L_1}} \end{array}\right|_{I^{L_1}}^2 + \left|\begin{array}{c} \overline{\mathbf{t}}_2^{G_{12}} - \hat{\mathbf{t}}_2^{L_2} \\ \overline{\mathbf{r}}_2^{G_{12}} - \hat{\mathbf{r}}_2^{L_2} \\ \overline{\mathbf{X}}_{\mathrm{f2}}^{G_{12}} - \hat{\mathbf{X}}_{\mathrm{f2}}^{L_2} \\ \overline{\mathbf{X}}_{\mathrm{f12}}^{G_{12}} - \hat{\mathbf{X}}_{\mathrm{f12}}^{L_2} \end{array}\right|_{I^{L_2}}^2 \tag{7.20}
$$

Ce problème de moindre peut s'écrire sous la forme factorisée suivante :

$$
\overline{f}\left(\overline{\mathbf{X}}^{G_{12}}\right) = ||A\overline{\mathbf{X}}^{G_{12}} - \mathbf{Z}||_{I_Z}^2 \tag{7.21}
$$

avec $\mathbf{Z}$ est le vecteur constat qui regroupe l'estimation des vecteurs d'états des deux carte.

$$
\mathbf{Z} = \left[\hat{\mathbf{X}}^{\mathrm{L_1}}, \hat{\mathbf{X}}^{\mathrm{L_2}}\right]
$$

I_Z est la matrice d'information correspondante donnée

$$
I_Z = \begin{bmatrix} I^{L_1} & 0 \\ 0 & I^{L_2} \end{bmatrix} \tag{7.22}
$$

$\overline{\mathbf{X}}^{G_{12}}$ est le vecteur d'état qui représente la carte globale défini dans l'équation (7.19) le coefficient matriciel A est une matrice creuse donnée par :

$$\begin{bmatrix} I & & & & & & \\ & I & & & & & \\ & & I & & & & \\ & & & I & & & \\ & & & & I & & \\ & & & & & I & \\ & & & & & & I \end{bmatrix} \tag{7.23}$$

où I est la matrice d'identité, sa taille correspond aux différentes variables dans le vecteur $\overline{\mathbf{X}}^{G_{12}}$.

La solution optimale $\hat{\overline{\mathbf{X}}}^{G_{12}}$ du problème de moindre carrée linéaire (7.21) peut être obtenue en résolvant l'équation creuse linéaire suivante :

$$A^T I_Z A \overline{\mathbf{X}}^{G_{12}} = A^T I_Z \mathbf{Z} \tag{7.24}$$

et la matrice d'information correspondante est calculée par :

$$\overline{I}^{G_{12}} = A^T I_Z A \tag{7.25}$$

à partir de l'équation (7.19), on a $\mathbf{X}^{G_{12}} = g^{-1}\left(\overline{\mathbf{X}}^{G_{12}}\right)$ ce qui implique

$$\Longrightarrow \left\{ \begin{array}{c} \mathbf{t}_0^{G_{12}} = \overline{R}_2 \left(\bar{\mathbf{t}}_0^{G_{12}} - \bar{\mathbf{t}}_1^{G_{12}} \right) \\ \mathbf{r}_0^{G_{12}} = r^{-1}\left(\overline{R}_0 \overline{R}_2^T \right) \\ \mathbf{t}_1^{G_{12}} = -\overline{R}_2 \bar{\mathbf{t}}_2^{G_{12}} \\ \mathbf{r}_1^{G_{12}} = r^{-1}\left(\overline{R}_2^T \right) \\ \mathbf{X}_{\mathrm{f1}}^{\mathrm{G}_{12}} = \overline{R}_2 \left(\overline{\mathbf{X}}_{\mathrm{f}_1}^{\mathrm{G}_{12}} - \bar{\mathbf{t}}_2^{G_{12}} \right) \\ \mathbf{X}_{\mathrm{f2}}^{\mathrm{G}_{12}} = \overline{R}_2 \left(\overline{\mathbf{X}}_{f_2}^{G_{12}} - \bar{\mathbf{t}}_2^{G_{12}} \right) \\ \mathbf{X}_{\mathrm{f12}}^{\mathrm{G}_{12}} = \overline{R}_2 \left(\overline{\mathbf{X}}_{\mathrm{f}_{12}}^{\mathrm{G}_{12}} - \bar{\mathbf{t}}_2^{G_{12}} \right) \end{array} \right\} \tag{7.26}$$

où $\overline{R}_2 = r\left(\bar{\mathbf{r}}_2^{G_{12}}\right)$,$\overline{R}_0 = r\left(\bar{\mathbf{r}}_0^{G_{12}}\right)$ sont les matrices de rotation de la position $\mathbf{P}_2$ et la position $\mathbf{P}_0$ dans le vecteur d'état $\overline{\mathbf{X}}^{G_{12}}$.

Maintenant la solution optimale du problème de moindre carré non-linéaire (7.17) peut être obtenue par :

$$\hat{\mathbf{X}}^{G_{12}} = g^{-1}\left(\hat{\overline{\mathbf{X}}}^{G_{12}}\right) \tag{7.27}$$

La matrice d'information $I^{G_{12}}$ correspondante peut aussi être obtenue par :

$$I^{G_{12}} = \nabla^T \overline{I}^{G_{12}} \nabla \tag{7.28}$$

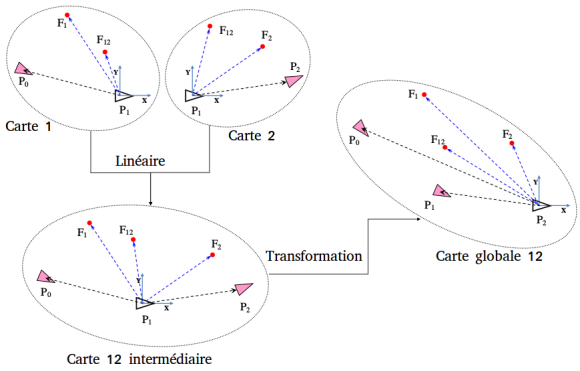


FIGURE 7.16: *principe du SLAM Linéaire*

avec ∇ est la jacobienne de $\overline{\mathbf{X}}^{G_{12}}$ donnée par :

$$\nabla = \frac{\partial g\left(\mathbf{X}^{G_{12}}\right)}{\partial \mathbf{X}^{G_{12}}}|_{\hat{\mathbf{X}}^{G_{12}}} \tag{7.29}$$

La carte globale résultante $\left(\hat{\mathbf{X}}^{G_{12}}, I^{G_{12}}\right)$ obtenue par la méthode linéaire est la même avec celle obtenue la méthode de moindre carré non-linéaire

En effet, la relation $\overline{\mathbf{X}}^{G_{12}} = g\left(\mathbf{X}^{G_{12}}\right)$ (dans les deux équations 7.18 et 7.19) est la fonction de changement de repère. Il transforme la position $\mathbf{P}_0$, $\mathbf{P}_1$ et l'amer $\mathbf{f}$ du repère d'origine $\mathbf{P}_2$ vers le repère d'origine $\mathbf{P}_0$. Il transforme aussi la position $\mathbf{P}_2$ et $\mathbf{f}$ vers le repère d'origine $\mathbf{P}_1$. En outre, la relation $\mathbf{X}^{G_{12}} = g^{-1}\left(\overline{\mathbf{X}}^{G_{12}}\right)$ de l'équation (7.26) est aussi une relation de transformation et de changement de repère, il transforme les position $\mathbf{P}_0$, $\mathbf{P}_2$ et $\mathbf{f}$ du repère d'origine $\mathbf{P}_1$ vers le repère d'origine $\mathbf{P}_2$. Donc du coups la méthode linéaire pour résoudre le problème de fusion de deux sous-cartes locales est équivalent à résoudre un problème de moindre carré linéaire plus un changement de repère, ceci est illustré dans la figure (7.16). Le fait que les deux cartes 1 et 2 sont construites par rapport au même repère, l'approche linéaire pour la fusion des deux cartes devient possible. C'est l'idée de base du SLAM linéaire.

7.3.7 Recherche active des amers dans le système ORB SLAM

Chaque amer observé dans K_1 et K_2 est recherché dans l'image courante comme le suivant :

- On calcule la projection de l'amer x dans l'image courante. L'amer est abandonné si la projection n'appartient pas à l'image.
- On calcul l'angle entre l'axe de vu courant v et le moyen de direction de vu n de l'amer. L'amer est abandonné si $n \times v \prec \cos(60)$
- On calcul la distance d qui sépare l'amer du centre de la caméra. L'amer est abandonné s'il n'appartient pas l'échelle de l'invariance de l'amer $d \notin [d_{min}, d_{max}]$.
- On calcul l'échelle de l'image par la relation d/d_{min}
- On compare le descripteur représentative D de l'amer dans la carte avec tous les amers non appariés dans l'image courante.

La position de la caméra est ensuite optimisé en utilisant les amers appariés.

7.4 Annexe D

7.4.1 Intégration de bruit dans la trajectoire odométrique

Les données odomètriques sont sujettes à des erreurs qu'on suppose qu'ils suivent une forme Gaussienne. L' erreurs sur les caractéristiques du véhicule tel que l'entraxe (d_b) ainsi que le rayon des roues (r). Vu qu'il sont sujettes à des variations légères au cours du temps. L' erreur sur les données provenant des capteurs odométriques, et l'erreur due au glissement des roues. Ces erreurs sont ingérées dans le modèle d'évolution pour calculer la position future des particules. La figure 7.17 représente le trajet des particules (200 particules) lors d'un mouvement rectiligne suivant l'axe y. Le trajet prend en compte les différents bruits qui peuvent affecter les données odométriques. En effet, au fur et à mesure du trajet, l'ensemble des particules se dispersent dans l'environnement, elles représentent l'ensemble des hypothèses du trajet du mobile. Plus le nombre de particules est élevées, plus la probabilité de représenter la trajectoire réelle est importante.

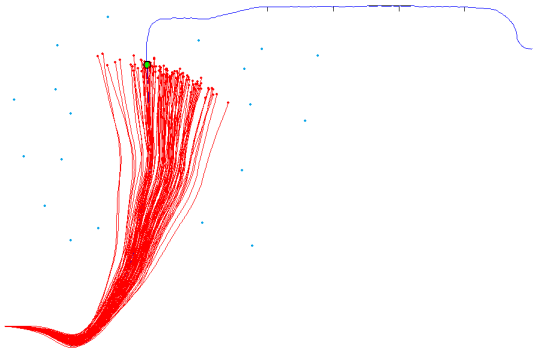


Figure 7.17: *Évolution de la position des particules en appliquant le modèle d'évolution et en intégrant les erreurs sur le système*

7.4.2 Calcule de la nouvelle distribution probabiliste

La nouvelle distribution proposée est donnée par l'équation (7.30).

$$\begin{aligned} \boldsymbol{\Sigma} &= [\mathbf{H}_p^T \boldsymbol{Z}_n^{-1} \mathbf{H}_p + (\boldsymbol{\Sigma})^{-1}]^{-1} \\ \boldsymbol{\mu} &= \boldsymbol{\mu} + \boldsymbol{\Sigma} \mathbf{H}_p^T \boldsymbol{Z}^{-1} (z_t - \hat{z}_t) \end{aligned} \tag{7.30}$$

avec :

— $\mathbf{H}_p$, $\mathbf{H}_f$ sont les jacobiennes du modèle d'observation dérivées respictivement par rapport à la position du robot s_t^m et celle de l'amer $\boldsymbol{X}_f$.

— $\mathbf{H}_u$ est la jacobienne du modèle du mouvement dérivée par rapport au contrôle u_t :$\mathbf{H}_u \rightarrow$ f $(d_c, \delta_\theta) \in \mathbb{R}^2 \Longleftrightarrow (x, y, \theta) \in \mathbb{R}^2$.

— $\mathbf{Z}_n$ est la covariance d'innovation.

— $\mathbf{C}_f$ est la matrice de covariance de l'amer, $\mathbf{R}_n$ est le bruit de mesure, $\mathbf{Q}_f$ est le bruit du contrôle,

— $\boldsymbol{\mu}^m$ et $\boldsymbol{\Sigma}^m$ sont la moyenne et la matrice de covariance de la distribution proposée.

Algorithme 5 : computing the new proposal distribution

for Each Particle **do**
  $\mu_0^m = s_t^m$, $\Sigma_0^m = H_u Q_t H_u^T$;
  for Each Matched Landmark **do**
    $Z_n \leftarrow H_f C_n H_f^T + R_n$;
    $\Sigma_n^m = [H_p^T Z_n^{-1} H_p + (\Sigma_{n-1}^m)^{-1}]^{-1}$;
    $\mu_n^m = \mu_{n-1}^m + \Sigma_n^m H_p^T Z_n^{-1}(z_t - \hat{z}_t)$;
  end
  $s_t^m \sim N(\mu_n^m, \Sigma_n^m)$;
end

7.4.3 Estimation et correction des amers

Le vecteur de position de l'amer $(x_c, y_c, \theta_f, \phi_f, \rho_f)$ dans le repère global est calculé en utilisant la formule :

$$\mathbf{X}_f = \begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{glob} = \begin{pmatrix} x_c \\ y_c \\ 0 \end{pmatrix} + \frac{1}{\rho_f}\overrightarrow{m}(\theta_f, \varphi_f) \tag{7.31}$$

avec :

$$\overrightarrow{m}(\theta_f, \varphi_f) = \begin{pmatrix} \cos(\theta_f)\cos(\varphi_f) \\ -\sin(\theta_f)\cos(\varphi_f) \\ \sin(\varphi_f) \end{pmatrix} \tag{7.32}$$

En utilisant la position du robot $s_t = (x, y, \theta)$ et la position de l'amer X_f, la position de l'amer dans le repère caméra s'écrit :

$$\mathbf{X}_{f,cam} = \mathrm{T}_{glob,cam}(\mathbf{X}_{f,glob}) \tag{7.33}$$

avec : $T_{glob,cam}$ la transformation entre le repère globale et le repère camera définie par :

$$T_{glob,cam} = \mathrm{T}_{\mathrm{mob,cam}}\left(\mathrm{T}_{\mathrm{glob,mob}}\left(x_f, y_f, z_f\right)_{\mathrm{glob}}\right) \tag{7.34}$$

avec : $T_{glob,mob}$ la transformation entre le repère globale et le repère mobile définie par :

$$\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{mob} = \mathrm{T}_{\mathrm{glob,mob}}\,(x_f, y_f, z_f)_{\mathrm{glob}} = R_{glob,mob}\left(\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{glob} - t_{glob,mob}\right) \tag{7.35}$$

$$= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}\left(\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{glob} - \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}\right) \tag{7.36}$$

$T_{mob,cam}$ est la transformation entre le repère mobile et le repère camera.

$$\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{cam} = \mathrm{T}_{\mathrm{mob,cam}}\,(x_f, y_f, z_f)_{\mathrm{mob}} = R_{mob,cam}\left(\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{mob} - t_{mob,cam}\right) \tag{7.37}$$

$$= \begin{bmatrix} c(\gamma)c(\beta) & c(\gamma)s(\beta)s(\alpha) - s(\gamma)c(\alpha) & c(\gamma)s(\beta)c(\alpha) + s(\gamma)s(\alpha) \\ s(\gamma)c(\beta) & s(\gamma)s(\beta)s(\alpha) + c(\gamma)c(\alpha) & s(\gamma)s(\beta)c(\alpha) - s(\gamma)s(\alpha) \\ -s(\beta) & c(\beta)s(\alpha) & c(\beta)c(\alpha) \end{bmatrix}$$
$$\left(\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}_{mob} - \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}\right) \tag{7.38}$$

avec $c(*) = \cos(*)$ et $s(*) = \sin(*)$.
En remplaçant $\mathbf{X}_f$ dans l'expression précédente, on obtient :

$$\mathbf{X}_{f,cam} = T_{glob,cam}\left(\begin{pmatrix} x_c \\ y_c \\ 0 \end{pmatrix} + \frac{1}{\rho_f}\overrightarrow{\mathbf{m}}(\theta_f, \varphi_f)\right) \tag{7.39}$$

La caméra n'observe pas directement $\mathbf{X}_{f,cam}$ mais sa projection sur l'image. Grâce au modèle Pinhole, on obtient :

$$\mathbf{h}_c = \begin{pmatrix} \hat{u}_f \\ \hat{v}_f \end{pmatrix} = \begin{pmatrix} c_u - Fk_u\frac{x_{f,cam}}{z_{f,cam}} \\ c_v - Fk_v\frac{y_{f,cam}}{z_{f,cam}} \end{pmatrix} \tag{7.40}$$

avec :

— F : la focale de la caméra.
— c_u, c_v : les coordonnées de la projection du centre optique de la caméra sur l'image.
— k_u, k_v : les facteurs d'agrandissement de l'image.

Quand un amer est observé, ses coordonnées sont mises à jours pour chaque particule en utilisant les équations du filtre Kalman. On calcule d'abord l'innovation $\mathbf{Y}_i$:

$$\mathbf{Y} = \begin{pmatrix} u_f \\ v_f \end{pmatrix} - \begin{pmatrix} \hat{u}_f \\ \hat{v}_f \end{pmatrix} \tag{7.41}$$

La matrice de covariance de l'innovation $\mathbf{Z}$ est donnée par :

$$\mathbf{Z} = \mathbf{H}_f \mathbf{C}_f \mathbf{H}_f^T + \mathbf{R} \tag{7.42}$$

Avec $\mathbf{H}_f$ la jacobienne du modèle d'observation (Pin-Hole) $\mathbf{h}_c$ et $\mathbf{R}$ la matrice de covariance du bruit :

$$\boldsymbol{H}_f \rightarrow h_c\left(\theta_f, \phi_f, \rho_f\right) \in \mathbb{R}^3 \Longleftrightarrow (u, v) \in \mathbb{R}^2$$

tel que :

$$\boldsymbol{H}_f = \begin{pmatrix} \frac{\partial u}{\partial \theta_f} & \frac{\partial u}{\partial \phi_f} & \frac{\partial u}{\partial \rho_f} \\ \frac{\partial v}{\partial \theta_f} & \frac{\partial v}{\partial \phi_f} & \frac{\partial v}{\partial \rho_f} \end{pmatrix}, \boldsymbol{R} = \begin{pmatrix} \sigma_u & 0 \\ 0 & \sigma_v \end{pmatrix}$$

Ensuite, les équations d'estimations classiques du filtre de Kalman sont :

$$\begin{array}{rcl} \mathbf{K} & = & \mathbf{C}_f \mathbf{H}_f^T \mathbf{Z}^{-1} \\ \mathbf{x} & = & \mathbf{x}_i + \mathbf{K}\mathbf{Y} \\ \mathbf{C}_f & = & (\mathbf{I} - \mathbf{K}\mathbf{H}_f)\mathbf{C}_f \end{array} \tag{7.43}$$

Après cette étape, nous avons mis à jour la position d'un amer observé pour une particule : ces calculs doivent être réalisés pour chaque amer observé et pour chaque particule. On remarque que le filtre de Kalman n'estime que trois variables pour chaque amer : $\mathbf{x} = (\theta_f, \phi_f, \rho_f)$. Les variables x_c et y_c sont supposées être connues exactement, elles représentent la position de la particule lors de la première observation de l'amer.

7.4.3.1 Calcule du poids d'importance

Pour tenir en compte l'observation et permettre la correction de la localisation du mobile, on calcule ce qu'on appel le poids d'importance. IL détermine la particule qui convient le mieux et dont la carte est le plus en adéquation avec l'observation. Autrement dit, le poids d'importance représente la probabilité d'observer un amer à partir d'une particule bien déterminée. Le poids est donné par le rapport entre la distribution probabiliste cible et la distribution proposée :

$$\omega = \frac{p\left(s_t^m | z_t, u_t\right)}{p\left(s_{t-1}^m |, z_{t-1}, u_{t-1}\right) p\left(s_t^m | s_{t-1}^m, z_t, u_t\right)} \tag{7.44}$$

On calcule ensuite la covariance d'innovation améliorée par rapport à la nouvelle distribution proposée :

$$\mathbf{L}_{\mathrm{u}} = \mathbf{H}_p \mathbf{P}_t \mathbf{H}_p^T + \mathbf{Z} \tag{7.45}$$

avec : $\mathbf{Z}$ est donnée par l'équation (7.42) et $\mathbf{H}_p$ la jacobienne du modèle d'observation ($\mathbf{h}_c$ ou $\mathbf{h}_l$) dérivée par rapport à la position du robot :

$\mathbf{H}_p \rightarrow \mathbf{h}_c\,(x, y, \theta) \in \mathbb{R}^3 \Longleftrightarrow (u, v) \in \mathbb{R}^2$ tel que : $\mathbf{H}_p = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial \theta} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial \theta} \end{pmatrix}$

ou bien :

$\mathbf{H}_p \rightarrow \mathbf{h}_l\,(x, y, \theta) \in \mathbb{R}^3 \Longleftrightarrow (r, \phi) \in \mathbb{R}^2$ tel que $\mathbf{H}_p = \begin{pmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial \theta} \\ \frac{\partial \phi}{\partial x} & \frac{\partial \phi}{\partial y} & \frac{\partial \phi}{\partial \theta} \end{pmatrix}$

Donc le poids d'importance est donné par :

$$\omega = \frac{1}{\sqrt{|2\pi\mathbf{Z}|}} exp\left[-0.5\mathbf{Y}^{\mathrm{T}}\mathbf{L}_u\mathbf{Y}\right] \tag{7.46}$$

A la fin de cette étape, on obtient pour chaque particule un poids d'importance qui représente l'adéquation entre la particule, sa carte et les observations correspondantes. Si plus d'un amer est apparié le poids résultant (7.47) pour chaque particule est le produit des poids calculés pour chaque amer apparié. (N est le nombre des amers appariés).

$$\omega_{total} = \prod_{i=1}^{N} \omega_i \tag{7.47}$$

7.4.4 Rééchantillonnage

Admettons que la densité de probabilité postérieure $p(s|z_t, u_t)$ qu'on veut représenter est celle dessinée avec les deux sommets en gras " Fig.7.18-(a) ". A l'instant $t = 0$ la position du robot est inconnue, donc la densité probabiliste est une distribution uniforme. La densité uniforme est échantillonnée pour chaque particule à travers le modèle d'évolution, ce qui donne une nouvelle densité représentée dans la " Fig.7.18-(b) ". Puis pour chaque particule de la nouvelle densité on calcule le poids d'importance. Pour que la densité résultante soit similaire à celle qu'on souhaite obtenir il faut que la densité des particules dans n'importe quelle tranche de l'axe du temps soit proportionnel à la probabilité de cette tranche, autrement dit dans les régions où la probabilité est grande, la densité des particules qui leur correspond doit être grande aussi et ainsi de suite. Contrairement au " Fig.7.18-(b) " les échantillons sont espacés de manière équidistante. Pour les ajuster de manière appropriée, il s'avère indispensable de les rééchantillonner [146] [147] [148]. Le principe consiste à augmenter le nombre des particules dont le poids est supérieur et de diminuer celles dont le poids est minimum, ou négligeable, tout en gardant constant le nombre total des particules dans le filtre. Le résultat ressemble à la troisième courbe " Fig.7.18-(c) " dont les particules pondérées reconstituent bien la densité envisagée.

Après avoir calculé le poids des particules, les particules dont la probabilité est trop faible sont supprimées. En supprimant la particule, on la remplace par une copie de la particule ayant le poids le plus élevé. Enfin, on met à jour la carte de la nouvelle particule.

7.4.5 Implémentation OpenGL

7.4.5.1 Implémentation OpenGL FB1

L'opération de prédiction par le multipass est décrite dans l'algorithme 6. s est un tableau des texture qui stock l'identificateur des deux texture de ping-pong (s_{old}, s_{new}). L'option de ces deux textures est changée à au sein de la boucle par la fonction `swap()`. `glDrawbuffer` détermine les textures utilisée pour l'écriture. Les deux fonctions `glActiveTexture` et `glBindTexture` déterminent les textures utilisées pour la lecture. N est le nombre des données odométriques acquises dans une itération,

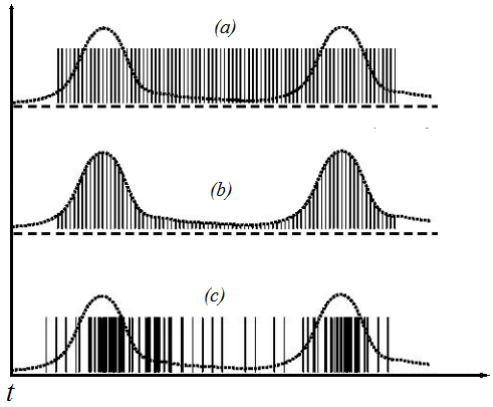


Figure 7.18: *La forme de la densité probabiliste après le rééchantillonnage*

`drawQuad()` est une fonction qui déclenche le calcul par la génération d'un rectangle convenable au type de parallélisation qu'on souhaite réaliser. La matrice de covariance initial P_m est calculée à la même manière en utilisant la technique de multipass.

Algorithme 6 : Prediction using multiple rendering pass

```
attachement[0] = GL_COLOR_ATTACHMENT0;
attachement[1] = GL_COLOR_ATTACHMENT1;
read = 1, write = 0;
Attach s[0] to attachement[0], and s[1] to attachement[1];
Transfer particles poses to s[1];
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE, u);
for m ← 1 to N do
    Transfer noisy encoders data to texture u;
    glDrawBuffer(attachement[write]);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_RECTANGLE, s[read]);
    drawQuad();
    swap(read,write);
end
```

7.4.5.2 Implémentation OpenGL FB3

L'implémentation parallèle de la procédure de mise à jour est décrite dans l'algorithme 7. Huit textures dont l'identificateur est stockées dans le tableau `pingpongTexID` sont attachées au frame buffer FBO. On détermine ensuite les textures à lecture-seule inchangées `LdmkTexID` qui vont stocker les paramètres des amers appariés. La première boucle de l'algorithme transfère les paramètres initiale de la distribution $\left(\mu_{old}^{m,t-1}, \Sigma_{old}^{m,t-1}\right)$ vers les quatre textures déjà attachées au FBO. La deuxième boucle est la boucle principale qui calcule la nouvelle distribution probabiliste en utilisant le multipass, avec N est le nombre des amers appariés. On transfère tous à chaque itération de la boucle de 'For' les paramètres des amers appariés vers les textures dont l'identificateur est stocké dans le tableau `LdmkTexID`. La fonction `glDrawBuffers` détermine les textures à écriture seule qui vont stocker les résultats de calcul d'un pass $\left(\mu_{new}^{m,t-1}, \Sigma_{new}^{m,t-1}\right)$. La boucle suivante détermine les textures à lecture seule qui vont stocker l'ancien paramètres de la distribution $\left(\mu_{old}^{m,t-1}, \Sigma_{old}^{m,t-1}\right)$. Dans le prochain pass , on échange l'option des huit textures pour une nouvelle opération de mise à jour. Ceci est répété

jusqu'à la construction complète de la nouvelle distribution probabiliste.

Algorithme 7 : GLSL Gaussian Construction

```
Attach eight ping-pong textures pingpongTexID[0..7] to FBO;
Set the four readable textures LdmkTexID[0..3];
for i ← 0 to 3 do
    Transfer the initial particles Gaussian state μ_old^{m,t-1}, Σ_old^{m,t-1} to textures pingpongTexID[i];
end
for n ← 1 to N do
    Transfer the n landmark state to textures LdmkTexID;
    glDrawBuffers(4,pingpongTexID);
    for i ← 0 to 3 do
        sets readable textures in pingpongTexID;
    end
    drawQuad();
    swap();
end
```

7.4.5.3 Implémentation OpenGL FB4

L'algorithme 8 décrit la procédure de parallélisation du bloc d'estimation FB4 sur le GPU. Chaque itération de la boucle 'For' corrige les paramètre d'un seul amer de la carte pour chaque particule en parallèle. Si plusieurs amer sont appariés, plusieurs itération seront nécessaire pour les corriger. Les résultats (paramètres mis à jour) sont transférer vers le CPU afin de permettre leur classification dans l'arbre binaire.

Algorithme 8 : GLSL estimation

for $n = 1$ **to** N **do**
Compute in parallel for each particle the updated state of the n **landmarks** :
- Transfer the matched landmark state $(u, v, x_0, y_0, \rho, \theta, \varphi, C_t)$ to four read-only input textures;
- Draw a filled rectangle to trigger the computation
- Download the corrected landmark state from four write-only output texutres to CPU

end

7.4.5.4 Implémentation OpenGL FB5

Pour exemplifier, la structure interne d'un shader de fragment, l'algorithme 9 décrit le source code du shader du fragment utilisé pour l'initialisation par inverse de profondeur.

`sampler2Drect` est un pointeur spécifique à l'unité texture activée. Donc, `partPose` point ver la texture qui stocke la position des particules. La première ligne du code récupère la position des particules dans la texture et les stocke dans un vecteur de quatre dimension `Pose_Particle` . Les deux lignes suivantes, calcule respectivement, la position de la camera et la position de l'amer dans le repère du monde. La position de la camera lors de la première observation (x_i, y_i), l'azimuthe θ_i et l'élévation ϕ_i sont stocké dans une variable spécifique appelé `gl_FragColor`.

Algorithme 9 : Fragment shader Inverse depth initialization

```
uniform sampler2DRect partPose;
uniform vec3 trans, uniform vec2 uv, uniform mat3 rot vec3 pos_cam, pos_lmk, vec4
pos_particle;
void main(void) {
pos_particle = texture2DRect(partPose, gl_TexCoord[0].st);
pos_came = rot*trans;
pos_lmk = compute_world_Ldmk_pose(uv);
gl_FragColor.x = pos_cam.x + pos_particle.x;
gl_FragColor.y = pos_cam.y + pos_particle.y;
gl_FragColor.z= - atan((pos_cam.x - pos_lmk.x)/(pos_cam.z - pos_lmk.z));
gl_FragColor.w = atan(pos_lmk.y/sqrt (pow(pos_lmk.x -pos_cam.x,2) + pow(pos_lmk.z -
pos_cam.z,2)));
}
```

7.4.6 Résultats de comparaison entre l'implémentation OpenCL et OpenGL

La figure 7.19 donne une évaluation quantitative des deux implémentations OpenCL et OpenGL et leur impact sur les différents blocs fonctionnels de l'algorithme.

Nous rappelons que pour l'implémentation du bloc FB1, nous avons utilisé deux méthodes différentes pour la génération des nombres aléatoires pour la randomisation du modèle d'évolution. L'implémentation OpenCL pour le bloc FB1 est plus rapide que celle en OpenGL quand M égale à 256, 1027 et 4096. Dans ce cas, on transfère 60 valeurs aléatoires du CPU vers le GPU, tandis que pour OpenGL on transfère quasiment M valeurs aléatoires générées pour chaque particule dans le filtre. Ceci est approuvé par l'implémentation avec $M = 16$ particules pour laquelle le transfert de 16 nombres aléatoires est plus rapide que le transfert de 60 nombres aléatoires. Par conséquence, l'implémentation OpenGL est plus avantageuse.

Pour l'implémentation du bloc FB3 par OpenCL et OpenGL, on transfère les paramètres des amers appariés pour toutes les particules. Malgré la quantité transférée égale pour les deux implémentations, l'implémentation OpenCL est plus rapide que celle en OpenGL dans la plus part des cas. En effet, la construction de la nouvelle distribution probabiliste par OpenCL est faite au sein du kernel. Une seule exécution du kernel permet la construction totale de cette distribution parce que la mémoire globale du GPU permet le stockage des paramètres de tous les amers appariés. Contrairement à l'implémentation OpenGL, la mémoire texture est limitée en termes de ressources. Elle ne permet pas le stockage des paramètres de tous les amers (32 textures au maximum). Donc les amers sont traités par le CPU, on transfère avant chaque exécution du shader du fragment, les paramètres d'un seul amer. La construction de toute la distribution probabiliste par OpenGL nécessite plusieurs transferts. Ce qui réduit les performances par rapport à l'implémentation OpenCL.

Les résultats obtenus par les deux implémentations OpenCL et OpenGL pour le bloc d'estimation FB4 peuvent être prévus. En effet, comme on a vu auparavant, pour l'implémentation OpenCL, l'exécution du bloc FB4 est faite sans aucun transfert ni avant ni après exécution parce que la mémoire globale du GPU contient déjà les paramètres de tous les amers appariés. Au contraire, le transfert est nécessaire pour l'implémentation OpenGL pour la même raison que précédemment (la limitation de la mémoire texture). En conséquence, l'implémentation OpenCL pour ce bloc est plus avantageuse.

Au contraire, l'implémentation du bloc FB5 par OpenGL et plus performante que celle en OpenCL

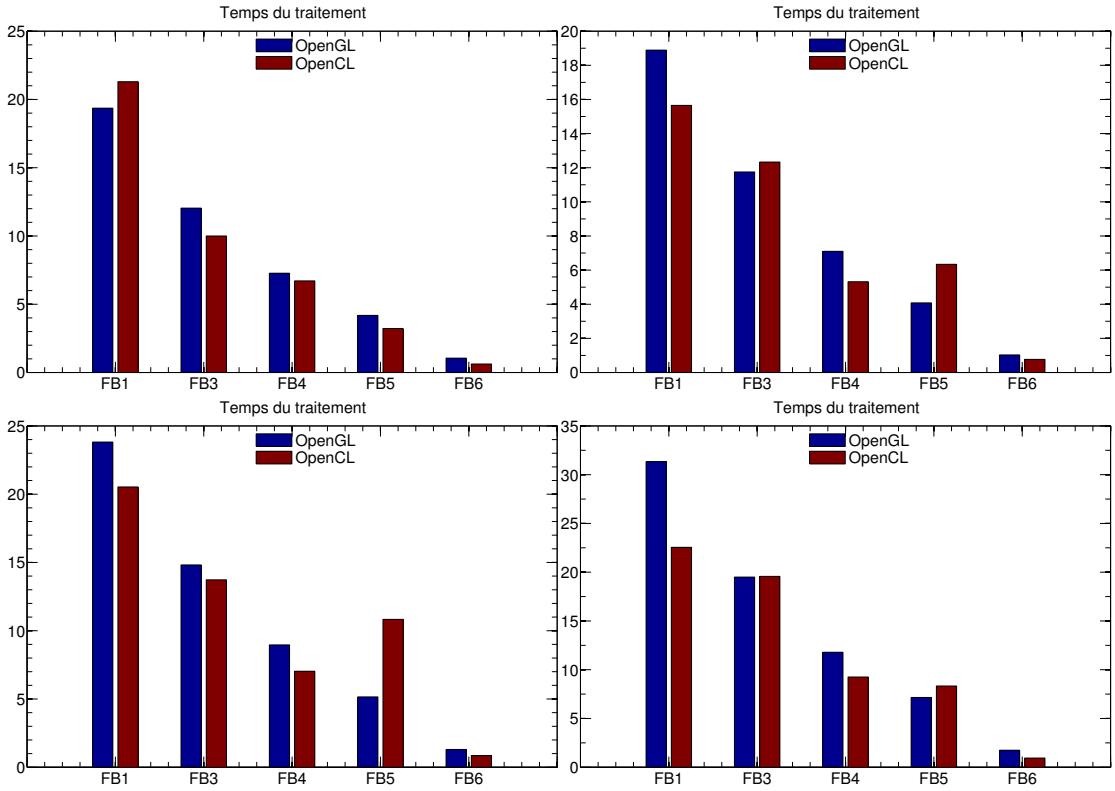


Figure 7.19: *Comparaison OpenCL-OpenGL pour 16, 256, 1024 et 4096 particules*

quand un grand nombre de particules est utilisé. En effet, dans l'implémentation OpenCL, on transfère, du GPU vers le CPU, toute la carte (les paramètres corrigés dans FB4 et initialisés dans FB5). En utilsant OpenGL, ceci est fait dans deux étapes : les paramètres corrigés sont transférés vers le CPU dans le bloc FB4 et ceux initialisés sont transférés vers le CPU dans FB5. Ceci réduit le coût du transfert. Le dernier bloc fonctionnel FB6 à été implémenté en utilisant les mêmes stratégies d'optimisation pour OpenCL et OpenGL. Pourtant, l'implémentation OpenCL est plus avantageuse.

En termes d'accélération globale, les résultats de la figure 7.20 montrent que l'implémentation OpenCL est plus intéressante que celle en OpenGL pour les différents nombres de particules. La figure 7.21 montre le facteur d'accélération des deux implémentations OpenCL et OpenGL calculé par rapport au temps d'exécution global sur le CPU multi-cœurs. L'implémentation OpenGL fournit un facteur d'accélération de 15x par rapport à l'implémentation multi-cœurs. Par contre l'implémentation OpenCL fournit un facteur d'accélération de 18x par rapport au CPU multi-cœurs . Plus le nombre de particules augmente, plus l'implémentation OpenCL est avantageuse.

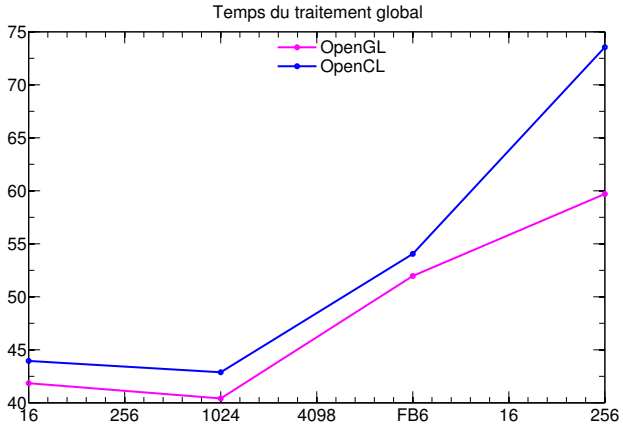


FIGURE 7.20: *Comparaison des temps d'exécution pour les implémentations OpenCL et OpenGL*

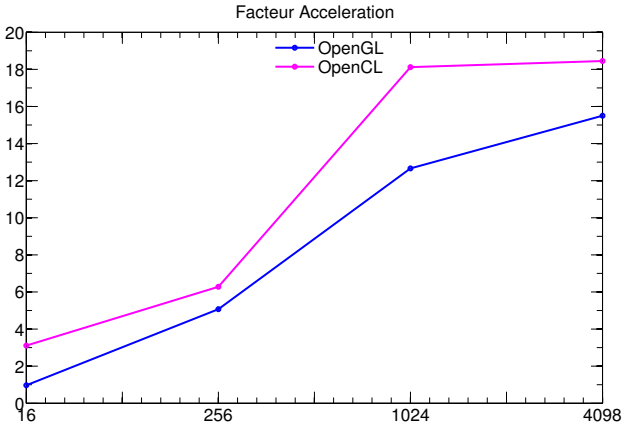


FIGURE 7.21: *Facteur d'accélération global*