



Localisation et cartographie simultanées par optimisation de graphe sur architectures hétérogènes pour l'embarqué

Abdelhamid Dine

► To cite this version:

Abdelhamid Dine. Localisation et cartographie simultanées par optimisation de graphe sur architectures hétérogènes pour l'embarqué. Systèmes embarqués. Université Paris-Saclay, 2016. Français. <NNT : 2016SACL303>. <tel-01552178>

HAL Id: tel-01552178

<https://tel.archives-ouvertes.fr/tel-01552178>

Submitted on 1 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

NNT : 2016SACLS303

THESE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PREPAREE A

UNIVERSITE PARIS-SUD
LABORATOIRE DES SYSTEMES ET APPLICATIONS DES TECHNOLOGIES
DE L'INFORMATION ET DE L'ENERGIE

ECOLE DOCTORALE N° 580
Sciences et Technologies de l'Information et de la Communication

Spécialité de doctorat
Robotique

Par

M. Abdelhamid Dine

Localisation et cartographie simultanées par optimisation de graphe
sur architectures hétérogènes pour l'embarqué

Thèse présentée et soutenue à Digiteo Labs, Gif-sur-Yvette, le 05/10/2016 :

Composition du Jury :

M. Rives Patrick	Directeur de recherche, INRIA Sophia Antipolis	Président du jury
M. Bodin François	Professeur, Université de Rennes, Rennes	Rapporteur
M. Houzet Dominique	Professeur, Grenoble-INP, Grenoble	Rapporteur
M. Vincke Bastien	Maître de conférences, Université Paris-Sud, Orsay	Examinateur
M. Bouaziz Samir	Professeur, Université Paris-Sud, Orsay	Directeur de thèse
M. Elouardi Abdelhafid	Maître de conférences, Université Paris-Sud, Orsay	Co-encadrant de thèse

Titre : Localisation et cartographie simultanées par optimisation de graphe sur architectures hétérogènes pour l'embarqué

Mots clés : SLAM par optimisation de graphe, complexité de calcul, structure de données, architectures hétérogènes embarquées, évaluation de performances.

Résumé : La localisation et cartographie simultanées connue, communément, sous le nom de SLAM (*Simultaneous Localization And Mapping*) est un processus qui permet à un robot explorant un environnement inconnu de reconstruire une carte de celui-ci tout en se localisant, en même temps, sur cette carte.

Dans ce travail de thèse, nous nous intéressons au SLAM par optimisation de graphe. Celui-ci utilise un graphe pour représenter et résoudre le problème de SLAM. Une optimisation de graphe consiste à trouver une configuration de graphe (trajectoire et carte) qui correspond le mieux aux contraintes introduites par les mesures capteurs. L'optimisation de graphe présente une forte complexité algorithmique et requiert des ressources de calcul et de mémoire importantes, particulièrement si l'on veut explorer de larges zones. Cela limite l'utilisation de

cette méthode dans des systèmes embarqués temps-réel.

Les travaux de cette thèse contribuent à l'atténuation de la complexité de calcul du SLAM par optimisation de graphe. Notre approche s'appuie sur deux axes complémentaires : la représentation mémoire des données et l'implantation sur architectures hétérogènes embarquées. Dans le premier axe, nous proposons une structure de données incrémentale pour représenter puis optimiser efficacement le graphe. Dans le second axe, nous explorons l'utilisation des architectures hétérogènes récentes pour accélérer le SLAM par optimisation de graphe. Nous proposons, donc, un modèle d'implantation adéquat aux applications embarquées en mettant en évidence les avantages et les inconvénients des architectures évaluées, à savoir SoCs à base de GPU et FPGA.

Title: Embedded graph-based simultaneous localization and mapping on heterogeneous architectures

Keywords: Graph-based SLAM, computational complexity, data structure, embedded heterogeneous architectures, performances evaluation.

Abstract: Simultaneous Localization And Mapping is the process that allows a robot to build a map of an unknown environment while at the same time it determines the robot position on this map.

In this work, we are interested in graph-based SLAM method. This method uses a graph to represent and solve the SLAM problem. A graph optimization consists in finding a graph configuration (trajectory and map) that better matches the constraints introduced by the sensors measurements. Graph optimization is characterized by a high computational complexity that requires high computational and memory resources, particularly to explore large areas. This limits the use of graph-based SLAM in real-time embedded systems.

embedded heterogeneous architectures. In the first axis, we propose an incremental data structure to efficiently represent and then optimize the graph. In the second axis, we explore the use of the recent heterogeneous architectures to speed up graph-based SLAM. We propose an efficient implementation model for embedded applications. We highlight the advantages and disadvantages of the evaluated architectures, namely GPU-based and FPGA-based Systems-On-Chips.

This thesis contributes to the reduction of the graph-based computational complexity. Our approach is based on two complementary axes: data representation in memory and implementation on

Remerciements

Mes louanges vont à Dieu de m'avoir donné le courage et la volonté de finir ce travail.

Je voudrais remercier les membres de mon jury de thèse qui ont bien voulu juger ce travail. Je les remercie pour la discussion scientifique et toutes les remarques et suggestions qui permettront d'étendre le travail effectué.

Je remercie vivement Monsieur Rives Patrick d'avoir accepté de présider mon jury de soutenance.

Je remercie Monsieur Bodin François et Monsieur Dominique Houzet pour avoir rapporté ce manuscrit.

Mes vifs remerciements accompagnés de toutes mes gratitude s'adressent à Monsieur Samir Bouaïz, mon directeur de thèse, pour m'avoir dirigé et aidé tout au long de ces trois ans de travail. Je le remercie pour sa disponibilité, ses conseils avisés, et surtout sa sympathie.

Je remercie Monsieur Abdelhafid Elouardi, mon co-encadrant de thèse, pour la relecture des papiers scientifiques et du manuscrit de thèse. Ses conseils m'ont été d'une grande aide pour l'achèvement de ce travail.

Je remercie Monsieur Bastien Vincke d'avoir participé au jury en tant qu'examinateur. Je le remercie pour tout le temps qu'il a consacré à relire mes papiers scientifiques. Son expertise sur les algorithmes de SLAM m'a été précieuse.

Je remercie également mes amis et collègues à Paris avec qui j'ai partagé de bons moments.

Enfin, je remercie ma famille et notamment mes parents pour leur soutien indéfectible. Sans eux, je n'aurais rien fait de tout cela.

Publications

Revue internationale avec actes et comités de lecture

1. "Graph-Based Simultaneous Localization and Mapping : Computational Complexity Reduction on a Multicore Heterogeneous Architecture"
A. Dine, A. Elouardi, B. Vincke, and S. Bouaziz, IEEE Robotics & Automation Magazine (RAM), 2016, to appear.

Congrès internationaux avec actes et comités de lecture

1. "Speeding up graph-based SLAM algorithm : A GPU-based heterogeneous architecture study,"
A. Dine, A. Elouardi, B. Vincke and S. Bouaziz, IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), Toronto, ON, 2015, pp. 72-73.
2. "Graph-based SLAM embedded implementation on low-cost architectures : A practical approach,"
A. Dine, A. Elouardi, B. Vincke and S. Bouaziz, IEEE International Conference on Robotics and Automation (ICRA), Seattle, WA, 2015, pp. 4612-4619.
3. "Efficient implementation of the graph-based SLAM on an OMAP processor,"
A. Dine, A. Elouardi, B. Vincke and S. Bouaziz, IEEE International Conference on Control Automation Robotics & Vision (ICARCV), Singapore, 2014, pp. 1935-1940.

Table des matières

1 SLAM par optimisation de graphe	7
1.1 Introduction	9
1.2 Processus de SLAM	11
1.2.1 Extraction des amers	11
1.2.2 Estimation des mesures	11
1.2.3 Mise en correspondance	12
1.2.4 Correction	13
1.3 Méthode de SLAM	13
1.3.1 EKF-SLAM	13
1.3.2 Fast-SLAM	15
1.3.3 GraphSLAM	16
1.3.4 SLAM ensembliste	18
1.3.5 Comparaison	19
1.4 Présentation du GraphSLAM	20
1.4.1 Définitions	20
1.4.2 GraphSLAM comme un problème de moindres carrés	21
1.4.3 Résolution du problème de moindres carrés	22
1.4.4 Algorithme GraphSLAM et partitionnement en blocs fonctionnels	23
1.4.5 Calcul des erreurs	27

1.4.6	Construction du système linéaire	29
1.4.7	Marginalisation des amers et complément de Schur	30
1.4.8	Construction du format CCS	31
1.4.9	Résolution du système	32
1.4.10	Calcul des incréments des amers	34
1.4.11	Mise à jour de l'état du système	34
1.5	Approches de réduction de la complexité de calcul	34
1.5.1	Rapidité de convergence	35
1.5.2	Élagage	36
1.5.3	Sous-cartographie et fenêtre glissante	37
1.5.4	Incrémentalité	37
1.5.5	Méthodes de résolution des NLS	38
1.5.6	Optimisation logicielle et matérielle	38
1.5.7	Notre approche	39
1.6	Conclusion	40
2	Architectures et outils de synthèse de haut niveau	41
2.1	Introduction	42
2.2	Processeurs graphiques	43
2.2.1	Architecture du Tegra K1	44
2.2.2	Programmation des processeurs graphiques	46
2.2.3	Processeurs graphiques pour l'accélération du SLAM	47
2.3	Architectures reconfigurables	48
2.3.1	Vers des FPGAs intégrant des processeurs hardcores	49
2.3.2	Architecture système-sur-puce du Cyclone V	50

2.3.3	FPGA pour le SLAM	51
2.4	Flux de conception sur FPGA : Outils de synthèse de haut niveau	52
2.4.1	Description manuelle	52
2.4.2	Synthèse de haut niveau	53
2.4.3	Présentation de LegUp	55
2.4.4	Approche OpenCL	58
2.5	Conclusion	62
3	Représentation mémoire pour le GraphSLAM incrémental	63
3.1	Introduction	64
3.2	Représentation mémoire du GraphSLAM	64
3.3	Structure de données à index (IDS)	66
3.4	Structure de données compacte et incrémentale	68
3.4.1	Représentation des éléments du graphe	68
3.4.2	Connectivité : connexions entre noeuds	69
3.4.3	Complément de Schur	70
3.4.4	Système linéaire	72
3.4.5	Construction incrémentale du graphe	73
3.5	Évaluations et résultats temporels	73
3.5.1	Présentation des jeux de données	75
3.5.2	Modalités d'évaluation	75
3.5.3	Résultats fonctionnels	77
3.5.4	Présentation des architectures d'évaluation	78
3.5.5	Métrique d'évaluation : Temps de Traitement par Élément de graphe (TPE) .	79
3.5.6	Charges temporelles des blocs fonctionnels	79

3.5.7	Évaluations temporelles sur un PC Intel	81
3.5.8	Évaluations temporelles sur architectures embarquées	84
3.5.9	Complément de Schur	86
3.5.10	Évolution du temps de traitement : TPE	87
3.6	Conclusion	91
4	Étude et implantation sur architectures hétérogènes à base de GPU	92
4.1	Introduction	93
4.2	Parallélisation de l'algorithme	94
4.2.1	Calcul des erreurs	94
4.2.2	Construction du système linéaire	94
4.2.3	Complément de Schur	97
4.2.4	Résolution du système linéaire	98
4.2.5	Calcul des incrément des amers	98
4.3	Effet de la fonctionnalité zéro-copie	98
4.4	Adéquation Algorithme-Architecture	101
4.4.1	Scalabilité	102
4.4.2	Analyse de l'évolution du temps par arête	103
4.4.3	Discussion et partitionnement CPU/GPU	105
4.4.4	Architecture mémoire	110
4.5	Résultats expérimentaux	111
4.5.1	Répartition des charges temporelles	111
4.5.2	Optimisation de graphe en mode batch	113
4.5.3	Optimisation de graphe en mode incrémental	114
4.6	Conclusion	116

5 Étude et implantation sur architectures hétérogènes à base de FPGA	117
5.1 Introduction	118
5.2 Architecture et optimisation de ressources	118
5.2.1 SLAM 2D	119
5.2.2 Calcul analytique des Jacobiennes	119
5.2.3 Unification des calculs	120
5.2.4 Incrémentalité	120
5.2.5 Calcul trigonométrique sur CPU	120
5.2.6 Calcul flottant	120
5.2.7 Modèle d'implantation HPS-FPGA	121
5.3 Approche OpenCL	121
5.3.1 Mémoire partagée	123
5.3.2 Ressources FPGA	124
5.3.3 Évaluation temporelle	124
5.4 Approche LegUp	126
5.4.1 Architecture	127
5.4.2 Évaluation temporelle	128
5.5 Conclusion	131
A Formulation probabiliste du GraphSLAM	138
B Présentation de la méthode de Cholesky	141
B.1 Renumérotation	141
B.2 Factorisation symbolique	142
B.3 Factorisation numérique	143
B.4 Résolution	145
C Validation fonctionnelle	146

Table des figures

1.1	Notations et illustration du problème de SLAM [Durrant-Whyte and Bailey, 2006].	9
1.2	Illustration du GraphSLAM.	16
1.3	Composantes du GraphSLAM : construction de graphe (Front-end) et optimisation de graphe (Back-end).	17
1.4	Erreur de mesure entre la pose x_i et la pose x_j . L'opérateur de soustraction dépend du modèle de mouvement (ou d'observation).	21
1.5	Repère global et mobile.	25
1.6	Partitionnement en blocs fonctionnels.	27
1.7	Construction incrémentale de la matrice d'information [Thrun and Montemerlo, 2006].	30
1.8	Exemple illustratif de la réduction de graphe.	32
2.1	Flexibilité vs performances.	42
2.2	Architecture Fermi de Nvidia (16 SM).	44
2.3	Architecture du Tegra K1 [Nvidia, 2013].	45
2.4	Architecture Kepler du Tegra K1 [Nvidia, 2013].	46
2.5	Structure interne d'un FPGA [Njiki, 2013].	49
2.6	Diagramme de blocs du Cyclone V.	51
2.7	Flux de synthèse de haut niveau.	53
2.8	Flux de conception de LegUp [Canis et al., 2011].	56
2.9	Architecture d'un système hétérogène généré par LegUp.	57

2.10	Hiérarchie d'instanciation des accélérateurs avec LegUp. A gauche l'arbre d'appels ; à droite l'architecture instanciée.	57
2.11	Exemple d'architectures générées par AOCL.	60
2.12	Parallélisme pipeliné dans AOCL.	61
3.1	Un exemple de graphe avec la structure éparse par blocs de la matrice d'information associée.	65
3.2	Représentation mémoire du graphe dans IDS. (a) Poses, (b) Arêtes de mouvement, (c) Arêtes d'observation, (d) Amers, (f) Table d'index ; valeurs non nulles en gris. (g) Graphe correspondant.	67
3.3	Illustration de la structure de données de LS.	69
3.4	Construction et mise à jour de la table <i>SchurNonZero</i> .	71
3.5	Jeux de données utilisés dans l'évaluation.	76
3.6	Erreurs euclidiennes : LS vs g^2o .	78
3.7	Répartition des charges, sans complément de Schur, pour le graphe de Victoria Park.	80
3.8	Répartition des charges, avec complément de Schur, pour le graphe de Victoria Park.	81
3.9	Temps de traitement par arête sur le PC Intel.	82
3.10	Temps de traitement par arête sur le TK1.	85
3.11	Temps de traitement par arête sur l'Exynos 5422.	87
3.12	Optimisation de graphe, avec complément de Schur, toutes 10 les étapes (s).	87
3.13	Évolution du TPE sans complément de Schur.	89
3.14	Évolution du TPE avec complément de Schur.	90
4.1	Plateforme Jetson de Nvidia.	94
4.2	Modification concurrente de $H(m_i, m_i)$ et b_{m_i} par 3 threads, pour la mise à jour par arête de la matrice et vecteur d'information. Les données modifiées par un thread sont distinguées par une couleur différente.	96
4.3	Mise à jour par nœud de la matrice et vecteur d'information. Dans l'illustration, 4 threads sont utilisés. Un thread par nœud. Aucun mécanisme de synchronisation n'est requis. Les données modifiées par un thread sont distinguées par une couleur différente.	96

4.4	Mémoire physique unifiée.	99
4.5	Temps de construction du format CCS (FB5) sur le TK1.	100
4.6	Variation du temps de construction du système linéaire en fonction du nombre de threads sur CPU.	103
4.7	Variation du temps de construction du système linéaire en fonction du nombre de threads sur GPU.	103
4.8	Exemple d'un environnement synthétique utilisé pour les évaluations [Vincke, 2012].	104
4.9	Évolution de la structure du graphe de l'environnement simulé.	105
4.10	Évolution du TPE.	106
4.11	Évolution du TPE dans le complément de Schur sur le TK1.	107
4.12	TPEs moyens des blocs fonctionnels sur le TK1.	108
4.13	Modèle CPU/GPU.	109
4.14	Ordonnancement des blocs fonctionnels dans l'implantation hétérogène CPU/GPU. .	110
4.15	Charges des blocs fonctionnels.	112
4.16	Comparaison des temps moyens par étape des blocs fonctionnels avec complément de Schur.	114
4.17	Comparaison des temps moyens par étape des blocs fonctionnels.	115
5.1	Modèle d'implantation CPU (HPS)/FPGA.	122
5.2	Temps de transfert HPS-FPGA. D : Device (FPGA), H : Host (HPS), SH : Shared Host (zone non paginée en mémoire HPS), SD : Shared Device (zone non paginée mappée dans l'espace d'adressage du FPGA).	123
5.3	Comparaison des TPEs.	125
5.4	Évolution du CPE.	126
5.5	Architecture générée par LegUp.	129
5.6	Nombre de cycles par étape.	129
5.7	Comportement du CPE de la brique FB2-FB3 sur l'architecture produite par LegUp.	130
C.1	Validation sur un jeu de données simulé.	146
C.2	Validation sur le jeu de données Pavin-Faust Dine et al., 2015a.	147

Liste des tableaux

1.1	Notations.	10
1.2	Comparaison entre les méthodes de SLAM. p : nombre de poses robot. K : nombre d'amers. r : nombre de particules.	20
1.3	Exemple d'un format CCS.	32
2.1	Specifications du Tegra K1.	45
2.2	Spécifications du FPGA de la plateforme DE1-SoC.	51
3.1	Structures de graphe des jeux de données.	75
3.2	Erreur euclidienne par rapport à la vérité du terrain.	78
3.3	Architectures d'évaluation.	79
3.4	Temps d'optimisation en mode batch sur le PC Intel (s).	83
3.5	Temps de traitement total en mode incrémental sur le PC Intel.	84
3.6	Temps de traitement total sur le TK1.	85
3.7	Temps de traitement total sur l'Exynos 5422.	86
3.8	Évaluation avec complément de Schur.	88
4.1	Temps des blocs fonctionnels après 1 itération (ms).	111
4.2	Temps moyens des blocs fonctionnels après 15 itérations (ms).	111
4.4	Temps de traitement total en mode incrémental. Optimisation de graphe effectuée toutes les 10 étapes.	113

4.3	Temps d'optimisation en mode batch sur le TK1 (s).	113
4.5	Temps cumulatif en utilisant le complément de Schur (s).	114
5.1	Ressources consommées de l'architecture produite par OpenCL.	124
5.2	Temps cumulatifs des sous-blocs de FB2 et FB3.	125
5.3	Ressources consommées, après synthèse, par l'architecture générée par LegUp.	128

Introduction générale

La localisation est un problème omniprésent qui a, depuis toujours, accompagné l'Homme. Pour se géo-localiser, l'Homme utilisait des repères statiques qu'il pouvait observer avec ses yeux (arbres, rivières, montagnes,...). Pour se souvenir de son chemin, l'Homme avait tendance à marquer sa trajectoire par le biais de signes visibles. De la même manière que pour l'Homme, la localisation constitue une fonctionnalité vitale pour les robots autonomes. Ceux-ci sont utilisés pour accomplir des tâches très variées telles que la transportation, les opérations de secours en environnements hostiles, le déminage, la surveillance, etc. D'autre part, durant la dernière décennie, les systèmes avancés d'aide à la conduite (Advanced Driver Assistance Systems) sont en pleine expansion. La voiture totalement autonome pourrait devenir, dans un futur proche, une réalité pour le grand public. L'accomplissement de ce type de tâches est assuré grâce à plusieurs fonctionnalités dont un robot autonome devrait disposer. La capacité de se localiser et percevoir un environnement, à priori inconnu, est l'une des principales fonctionnalités en robotique autonome.

Nous nous intéressons, dans ce travail de thèse, au problème de localisation et cartographie simultanées, connu communément sous le nom de SLAM (Simultaneous Localization And Mapping). Le SLAM permet à un robot mobile explorant un environnement inconnu de reconstruire une carte de celui-ci tout en déterminant, en même temps, la position du robot sur cette carte. Celle-ci est constituée d'éléments particuliers de l'environnement que l'on appelle *amers*. Pour accomplir le processus de SLAM, le robot utilise des capteurs proprioceptifs (odomètres, centrale inertuelle,...) pour s'informer sur son propre déplacement. Il utilise également des capteurs extéroceptifs (Télémètre laser, SONAR, caméra,...) pour percevoir l'environnement. A cause du bruit inhérent sur les mesures capteurs, la trajectoire reconstruite, à partir des données proprioceptives, peut diverger. En d'autres termes, elle ne reflète pas la trajectoire réelle suivie par le robot. Cela peut provoquer des décisions dangereuses pour le robot et l'être humain. De ce fait, il n'est pas possible de localiser le robot sans la carte. Celle-ci comprend des repères qui permettent de corriger la position du robot. En même temps, pour reconstruire la carte, il faut connaître depuis quelle position du robot, les amers ont été observés. Cela explique la nécessité d'effectuer simultanément et en un seul processus la localisation et la cartographie.

Durant les deux dernières décennies, le SLAM a reçu une attention considérable de la communauté de recherche en robotique mobile. Plusieurs approches ont été proposées pour résoudre le problème de SLAM. Ces approches peuvent être classifiées comme approches de filtrage ou de lissage. L'objectif des approches par filtrage est, en général, de déterminer la position courante du robot ainsi que la carte des amers qui, ensemble, constituent l'état du système. Ces approches permettent une estimation en ligne de l'état du système à chaque nouvelle mesure. Plusieurs méthodes ont été proposées dans cette catégorie. [Dissanayake et al., 2001] ont proposé l'utilisation du Filtre de Kalman Etendu (EKF). Le FastSLAM [Montemerlo et al., 2002] est une autre solution basée sur le filtre particulaire. Ces méthodes de filtrage souffrent d'un problème de consistance. Celle-ci peut être définie par la qualité de l'incertitude de localisation fournie par le filtre par rapport à l'erreur réelle. Les approches de lissage estiment, en plus de la carte, toute la trajectoire du robot en utilisant toutes les mesures capteurs. Le lissage permet de raffiner continuellement toute la trajectoire du robot. Ces approches s'appuient sur un problème d'optimisation qui peut être formulé par un problème de moindres carrés (Non-linear Least Squares). Le SLAM par optimisation de graphe, souvent appelé GraphSLAM [Thrun and

Montemerlo, 2006], est une approche de lissage. Le GraphSLAM permet de faire face au problème de consistance présent dans les approches de filtrage. Le GraphSLAM s'impose de plus en plus, au sein de la communauté scientifique, comme un premier choix pour le SLAM sur de larges zones.

Nous nous intéressons particulièrement au SLAM par optimisation de graphe. Celui-ci utilise un graphe pour représenter le problème de SLAM. La trajectoire et la carte de l'environnement sont modélisées par des nœuds. Une arête modélise une contrainte spatiale entre les nœuds reliés. Les contraintes spatiales sont obtenues au moyen des capteurs proprioceptifs et extéroceptifs. Une optimisation de graphe consiste à trouver une configuration de graphe (trajectoire et carte) qui correspond le mieux aux contraintes introduites par les mesures capteurs. L'estimation de la trajectoire et la carte se ramène à un problème de moindres carrés. La résolution de celui-ci présente une forte complexité algorithmique. En effet, la dimension du graphe s'accroît au fur et à mesure que le robot explore l'environnement. Cela requiert des ressources de calcul et de mémoire importantes pour optimiser le graphe, particulièrement si l'on veut explorer de larges zones. Cela limite l'utilisation d'architectures embarquées pour réaliser des systèmes de SLAM par optimisation de graphe.

Objectif et contributions

Le travail de cette thèse a pour objectif de réduire la complexité de calcul du GraphSLAM. Dans ce contexte, plusieurs approches ont été proposées par la communauté scientifique en vue d'accélérer les traitements du GraphSLAM et permettre un SLAM à large-échelle. De nombreux travaux se sont intéressés à la complexité du graphe lui-même. L'idée est globalement de limiter le nombre de nœuds et la connectivité dans le graphe en utilisant plusieurs techniques d'élagage. D'autres approches utilisent la stratégie de sous-cartographie. Le graphe est divisé en plusieurs sous-graphes (sous-cartes) où chacun est indépendamment optimisé. Les sous-graphes sont ensuite joints afin de retrouver le graphe initial. Les approches incrémentales exploitent le caractère incrémental du GraphSLAM. C'est-à-dire, l'historique des optimisations de graphe au fil de la navigation. Elle permettent, ainsi, de mettre à jour d'une manière incrémentale le système linéaire associé au problème de moindres carrés.

Pour réduire encore la complexité de calcul, notre approche s'appuie sur deux axes complémentaires : la représentation mémoire des données et l'implantation sur architectures hétérogènes embarquées. Ce document présente essentiellement deux contributions :

- Dans la première contribution, nous proposons une structure de données adéquate pour représenter le graphe en mémoire et résoudre efficacement le problème de moindres carrés associé. La représentation mémoire est couplée à une construction de graphe incrémentale. La structure de données est, donc, mise à jour d'une manière incrémentale lors la mise à jour du graphe. L'organisation de données en mémoire, dans le contexte du GraphSLAM, a été très peu étudiée. Nous montrerons à travers cette contribution l'accélération importante que peut apporter une structure de données adéquate.
- Dans la seconde contribution, nous explorons l'utilisation d'architectures hétérogènes embarquées pour accélérer le GraphSLAM dans un contexte d'Adéquation Algorithme-Architecture

(AAA). L'émergence des architectures hétérogènes récentes devrait mener à une grande avancée dans la conception des systèmes embarqués dédiés à la robotique mobile. Ce type d'architectures intègre, sur une même puce, différents types de calculateurs. Parmi ces architectures, on trouve les systèmes-sur-puce à base de FPGAs et des coeurs ARM (Cyclone V, Zynq-7020), ainsi que les systèmes-sur-puce à base de processeurs graphiques et des coeurs ARM (Tegra K1). En se basant sur notre expertise acquise sur l'algorithme du GraphSLAM et des architectures hétérogènes, nous définissons des modèles d'implantation adéquats aux architectures hétérogènes embarquées basées GPU et FPGA.

Par ailleurs, le travail se veut également une étude des performances des nouvelles architectures hétérogènes ainsi que leurs outils connexes, en particulier, les outils de synthèse de haut niveau.

Structure du manuscrit

Chapitre 1 : Dans ce chapitre, nous donnons une introduction au processus de SLAM ainsi qu'aux principales méthodes de SLAM. Nous présentons, ensuite, l'algorithme du GraphSLAM avec tous les détails essentiels à la compréhension du manuscrit. Pour mieux étudier et implanter l'algorithme, celui-ci est partitionné en blocs fonctionnels. Un bloc fonctionnel peut être défini comme un ensemble d'instructions destinées à accomplir, en un temps fini, le traitement d'une tâche donnée de l'algorithme. Nous concluons ce chapitre par la présentation des différentes approches et techniques utilisées par la communauté scientifique en vue de réduire la complexité de calcul du GraphSLAM.

Chapitre 2 : Ce chapitre présente les architectures de calcul utilisées dans ce travail avec un retour sur leur utilisation dans les systèmes de SLAM. Nous présentons, en particulier, les processeurs graphiques et les architectures reconfigurables. Pour ces dernières, nous avons utilisé des outils de synthèse de haut niveau. Ceux-ci sont discutés en mettant en avant leurs principes ainsi que leurs avantages et inconvénients.

Chapitre 3 : Nous présentons dans ce chapitre deux structures de données que nous avons proposées pour représenter le graphe en mémoire et résoudre le problème de moindres carrés. Les deux structures données sont comparées à deux implantations de l'état de l'art. Nous donnons des évaluations détaillées des quatre implantations sur trois architectures de calcul. Les évaluations temporelles portent sur un large ensemble de jeux de données.

Chapitre 4 : Ce chapitre est consacré à l'étude d'une implantation du GraphSLAM sur une architecture hétérogène embarquée basée GPU. Nous décrivons les principales transformations algorithmiques nécessaires pour porter l'algorithme sur une architecture parallèle. Partant d'une analyse incrémentale du temps de calcul au niveau des blocs fonctionnels, nous définissons puis évaluons le modèle d'implantation hétérogène CPU/GPU.

Chapitre 5 : Ce chapitre étudie l'utilisation d'architectures hétérogènes basées FPGA pour l'implantation du GraphSLAM. Pour ce faire, nous avons utilisé des outils de synthèse de haut niveau, à savoir LegUp et OpenCL d'Altera. Nous discutons le modèle d'implantation CPU/FPGA défini ainsi que les instances architecturales produites par chacun des deux outils.

Nous conclurons ce document par une conclusion sur les différentes contributions et nous donnerons nos perspectives de recherche.

Chapitre 1

SLAM par optimisation de graphe

Sommaire

1.1	Introduction	9
1.2	Processus de SLAM	11
1.2.1	Extraction des amers	11
1.2.2	Estimation des mesures	11
1.2.3	Mise en correspondance	12
1.2.4	Correction	13
1.3	Méthode de SLAM	13
1.3.1	EKF-SLAM	13
1.3.2	Fast-SLAM	15
1.3.3	GraphSLAM	16
1.3.4	SLAM ensembliste	18
1.3.5	Comparaison	19
1.4	Présentation du GraphSLAM	20
1.4.1	Définitions	20
1.4.2	GraphSLAM comme un problème de moindres carrés	21
1.4.3	Résolution du problème de moindres carrés	22
1.4.4	Algorithme GraphSLAM et partitionnement en blocs fonctionnels	23
1.4.5	Calcul des erreurs	27
1.4.6	Construction du système linéaire	29
1.4.7	Marginalisation des amers et complément de Schur	30
1.4.8	Construction du format CCS	31
1.4.9	Résolution du système	32
1.4.10	Calcul des incrément des amers	34
1.4.11	Mise à jour de l'état du système	34
1.5	Approches de réduction de la complexité de calcul	34
1.5.1	Rapidité de convergence	35
1.5.2	Élagage	36

1.5.3	Sous-cartographie et fenêtre glissante	37
1.5.4	Incrémentalité	37
1.5.5	Méthodes de résolution des NLS	38
1.5.6	Optimisation logicielle et matérielle	38
1.5.7	Notre approche	39
1.6	Conclusion	40

1.1 Introduction

Le terme SLAM (Simultaneous Localization and Mapping) est apparu dans les travaux de [[Leonard and Durrant-Whyte, 1991](#)]. Le processus de localisation et cartographie simultanées consiste à reconstruire la carte de l'environnement exploré tout en localisant, en même temps, le robot sur celle-ci. Initialement, le robot n'a aucune information à priori sur l'environnement et sa position.

Considérons un robot qui se déplace dans un environnement inconnu. L'objectif du SLAM est d'estimer la position du robot et la carte des amers à chaque instant t . Le tableau 1.1 résume les principales notations qui seront utilisées tout au long de ce document. La figure 1.1 illustre le problème de SLAM. Au fur et à mesure que le robot se déplace, il acquiert des données sur son propre déplacement ($u_{1:t}$) moyennant des capteurs proprioceptifs (odométrie, centrale inertuelle,...). Ces données peuvent être exploitées pour reconstruire la trajectoire du robot ($x_{0:t}$). L'une des difficultés est que les mesures fournies par les capteurs sont intrinsèquement bruitées. La trajectoire donnée par les capteurs proprioceptifs ne correspond souvent pas à la trajectoire réelle du robot. Pour corriger cette estimation, le robot dispose de capteurs extéroceptifs (télémètre Laser, caméra, ultrason, Lidar,...). Ceux-ci permettent de percevoir l'environnement et d'y détecter des éléments particuliers que l'on appelle amers. Ces derniers constituent la carte m de l'environnement exploré. Les mesures extéroceptives $z_{1:t}$ et proprioceptives ($u_{1:t}$) sont fusionnées via une méthode de SLAM pour corriger simultanément la position du robot et la carte des amers.

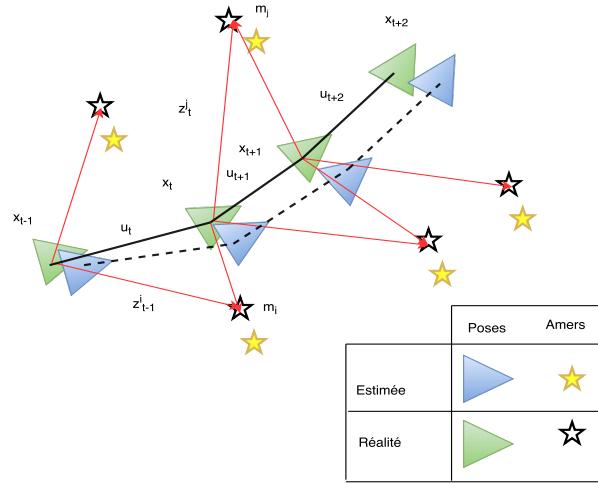


FIGURE 1.1 – Notations et illustration du problème de SLAM [[Durrant-Whyte and Bailey, 2006](#)].

Par ailleurs, les méthodes de SLAM les plus utilisées dans l'état de l'art sont probabilistes. Des modèles probabilistes pour le robot et son environnement sont utilisés. Les méthodes probabilistes s'appuient sur l'inférence Bayésienne pour estimer la position du robot et reconstruire la carte de l'environnement. En général, le SLAM est décrit par une distribution probabiliste [[Dissanayake et al., 2001](#), [Durrant-Whyte and Bailey, 2006](#)] :

$$p(x_t, m | z_{1:t}, u_{1:t}, x_0) \quad (1.1)$$

L'équation 1.1 décrit la densité conjointe à posteriori de la carte et la position courante du robot sachant l'historique des observations, des commandes de mouvement ainsi que la position initiale.

TABLE 1.1 – Notations.

Notation	Description
x_t	Pose du robot à l'instant t
$x_{0:t}$	Ensemble des poses du robot jusqu'à l'instant t
u_t	Commande de mouvement entre les instants $t - 1$ et t
$u_{1:t}$	Historique des commandes de mouvement jusqu'à l'instant t
m	Ensemble des amers (carte)
m_j	Position de l'amer j
z_t	Ensembles des mesures d'observation à l'instant t
$z_{1:t}$	Historique des mesures d'observation jusqu'à l'instant t
z_t^j	Mesure de l'amer j à l'instant t
z_{ij}	Mesure capteur associée à l'arête reliant les noeuds i et j
\hat{z}_{ij}	Mesure prédictive associée à l'arête reliant les noeuds i et j
Ω_{ij}	Inverse de la matrice de variances-covariances du bruit sur la mesure de l'arête ij
e_{ij}	Erreur sur l'arête reliant les noeuds i et j
C_{ij}	Contrainte spatiale entre les noeuds i et j
g	Modèle de mouvement
h	Modèle d'observation

Plusieurs approches ont été proposées dans la littérature pour résoudre le problème de SLAM. Celles-ci peuvent être classifiées, dans un premier lieu, en approches probabilistes et ensemblistes. L'approche dominante est l'approche probabiliste. Elle cherche à estimer, en général, la densité à posteriori de la carte et la position du robot. Par ailleurs, on peut diviser encore les approches de SLAM en approches de filtrage et approches de lissage (ou approches d'optimisation). Les approches de filtrage corrigent la position courante et la carte. Dans cette catégorie, on trouve principalement l'EKF-SLAM (Extended Kalman Filter for SLAM) [Smith and Cheeseman, 1986, Smith et al., 1990, Dissanayake et al., 2001] et le FastSLAM (Filtre particulaire) [Montemerlo et al., 2002]. Les approches de lissage permettent de corriger la carte des amers et toute la trajectoire depuis l'état initial. Le SLAM par optimisation de graphe [Lu and Milios, 1997] est une méthode de lissage. Ce travail de thèse s'intéresse particulièrement à cette dernière méthode pour ses avantages par rapport à l'EKF-SLAM et le FastSLAM. Dans ce manuscrit de thèse, nous utilisons l'appellation *GraphSLAM* pour désigner le SLAM par optimisation de graphe. Le terme GraphSLAM [Thrun and Montemerlo, 2006, Grisetti et al., 2010] est le plus populaire, au sein de la communauté scientifique, pour faire référence aux méthodes de SLAM basées sur l'optimisation de graphe.

Nous allons présenter, dans un premier temps, ces approches en expliquant leurs principes, et en mettant en évidence les avantages et les inconvénients de chacune d'entre elles. Nous donnons ensuite une description détaillée du GraphSLAM. Nous étudierons, ensuite, les différentes approches suivies pour réduire la complexité de calcul du GraphSLAM.

1.2 Processus de SLAM

Dans la pratique, on peut découper le processus de SLAM en quatre étapes :

- Extractions des amers
- Estimation des mesures
- Mise en correspondance
- Correction

1.2.1 Extraction des amers

Cette étape consiste en l'extraction des amers après acquisition des données par un ou plusieurs capteurs extéroceptifs. Les amers sont des éléments particuliers de l'environnement identifiables sans ambiguïté par le robot. Un amer est, en général, un point d'intérêt de l'environnement. Il peut aussi avoir d'autres formes géométriques (rectangle, segments, voire des objets [Salas-Moreno et al., 2013]). Différents types de capteurs peuvent être utilisés pour la détection des amers. Selon le type du capteur utilisé, on distingue, en général, trois catégories : SLAM basé Laser [Hahnel et al., 2003, Holz and Behnke, 2016], SLAM basé SONAR (SOund Navigation And Ranging) [Kleeman, 2003, Jaulin, 2006, Sliwka et al., 2011] et SLAM basé vision. Des critères comme la précision, le type de la carte à reconstruire ainsi que la nature de l'environnement influencent le choix des capteurs.

Dans le SLAM basé Laser, la perception de l'environnement est, souvent, assurée par des télémètres Laser. Ceux-ci se distinguent par une grande vitesse d'acquisition couplée à une haute précision. Cela a fait des télémètres Laser un choix incontournable dans les premiers systèmes de SLAM. Le SLAM visuel [DeSouza and Kak, 2002] utilise une ou plusieurs caméras comme capteurs extéroceptifs. Une image constitue une source très riche en informations en comparaison avec les capteurs Laser. Les caméras sont également adaptées aux systèmes embarquées : elles sont souvent légères, bas-coût et moins consommatrices en énergie que les capteurs Laser. Les amers sont extraits suite à la détection de points d'intérêt dans les images acquises. Pour ce faire, il existe plusieurs algorithmes de détection et de description : Harris [Harris and Stephens, 1988], SIFT [Lowe, 1999], SURF [Bay et al., 2008], FAST [Rosten et al., 2010], BRIEF [Calonder et al., 2010], ORB [Rublee et al., 2011], BRISK [Leutenegger et al., 2011], FRICK [Alahi et al., 2012]. Finalement, pour faire face au bruit et à l'insuffisance d'informations, on peut combiner plusieurs types de capteurs. Cela donne lieu à un SLAM hybride [Biber et al., 2004, Fu et al., 2007, Gallegos et al., 2010, Shen et al., 2011, Oh et al., 2015, Karakaya et al., 2015].

1.2.2 Estimation des mesures

Dans cette étape, on effectue une première estimation de la position courante du robot et la carte en utilisant, respectivement, le modèle d'observation et le modèle de mouvement. Ceux-ci sont des modèles mathématiques dépendant des capteurs utilisés. Pour l'odométrie, la position courante du

robot est obtenue par intégration successive des données odométriques. Dans le SLAM 3D où les amers résident dans un plan 3D, l'initialisation de la profondeur d'un amer constitue un grand défi particulièrement dans le SLAM monoculaire [DeSouza and Kak, 2002, Davison, 2003, Celik et al., 2008, Hwang and Song, 2011, Scaramuzza, 2011, Choi et al., 2011, Nourani-Vatani and Borges, 2011]. Celui-ci utilise une seule caméra. Celle-ci étant un capteur projectif, la profondeur ne peut être obtenue que si l'amer a été observé depuis, au moins, deux positions différentes. [Davison, 2003] a introduit l'initialisation retardée des amers. L'initialisation est raffinée en traquant le point sur plusieurs images. Un point 2D n'est considéré comme un amer 3D que si son initialisation est assez bonne. L'introduction de l'initialisation non retardée [Sola et al., 2005] et puis la paramétrisation par inverse de profondeur (inverse depth parametrisation) [Civera et al., 2006] ont beaucoup contribué à la résolution du problème d'initialisation d'amers pour le SLAM monoculaire. [Davison et al., 2007] ont proposé un nouvel algorithme de SLAM appelé MonoSLAM. Celui-ci permet d'estimer la trajectoire 3D d'une caméra sans utiliser de capteurs proprioceptifs. La profondeur est obtenue suite au mouvement de la caméra. D'autres travaux [Klein and Murray, 2007, Kitt et al., 2010, Geiger et al., 2011, Ventura et al., 2014] utilisent la technique de triangulation, entre images consécutives, pour initialiser les amers.

Dans le SLAM monoculaire, la trajectoire et la carte de l'environnement sont données à un facteur d'échelle près. Celui-ci résulte des projections des points 3D de l'environnement sur un plan 2D de la caméra. Un point 2D sur l'image peut correspondre à un nombre infini de points constituant une droite. Pour pallier à ce problème, plusieurs techniques ont été proposées dans la littérature. [Royer et al., 2005] utilisent un GPS différentiel pour corriger le facteur d'échelle. [Kitt et al., 2010, Geiger et al., 2011] s'appuient sur la détection d'un certain nombre de points sur le sol afin de corriger le facteur d'échelle. Similairement, [de la Escalera et al., 2016] détectent et traquent des points statiques sur le sol pour calculer le mouvement relatif entre les images.

Dans le même contexte et pour simplifier l'initialisation des amers 3D, un système stéréo peut être utilisé [Agrawal et al., 2005, Carrasco et al., 2016, Herath et al., 2007, Lemaire et al., 2007, Moreno et al., 2015, Zhang et al., 2015, Takezawa et al., 2004]. Les caméras stéréoscopiques fournissent la profondeur ou la distance aux objets dans l'environnement. Cela facilite l'initialisation des amers. Par ailleurs, l'émergence de la nouvelle génération des caméras RGB-D (La Kinect, Primesense Carmine, Asus Xtion) a aussi contribué à l'évolution rapide du SLAM 3D [Henry et al., 2010, Engelhard et al., 2011, Endres et al., 2012, Engel et al., 2014, Melbouci et al., 2015]. Ces caméras utilisent des projections infrarouges pour calculer la profondeur.

1.2.3 Mise en correspondance

Pour une meilleure correction de la position du robot et la carte, un amer devrait être observé plusieurs fois au cours de la navigation du robot. Il est donc nécessaire de mettre en correspondance, après chaque extraction, les amers détectés avec ceux déjà existants dans la carte. Cette opération est possible en associant un descripteur unique à chaque amer détecté. Un descripteur peut être assimilé à une carte d'identité permettant d'identifier l'amer lors de la navigation. Dans le SLAM visuel, la

manière la plus intuitive est d'utiliser le voisinage du point d'intérêt comme descripteur [Davison, 2003]. D'autres types de descripteurs sont plus complexes. Ils sont également calculés en utilisant le voisinage du point d'intérêt (BRIEF, ORB, BRISK, FRICK). Une distance de similarité est ensuite calculée entre l'amer détecté et ceux déjà dans la carte. L'amer est associé à l'amer offrant la meilleure similarité et dépassant un certain seuil.

1.2.4 Correction

La phase de correction est le cœur du processus de SLAM. Elle corrige la position du robot et la carte en fusionnant les mesures des amers et du déplacement. Pour les méthodes probabilistes, on trouve essentiellement trois méthodes : l'EKF-SLAM, le FastSLAM et le SLAM par optimisation de graphe. L'EKF-SLAM se base sur un filtre de Kalman Étendu. Le FastSLAM s'appuie, à son tour, sur un filtre particulier. Le SLAM par optimisation de graphe ou GraphSLAM utilise une représentation graphique du problème de SLAM. D'autre part, il existe l'approche ensembliste. Celle-ci utilise l'analyse par intervalle dans un objectif de garantir les résultats de localisation et de cartographie contrairement aux approches Bayésiennes où la position du robot (ou la carte) est donné avec une incertitude (probabilité).

1.3 Méthode de SLAM

1.3.1 EKF-SLAM

La première application du Filtre de Kalman Étendu (EKF) pour la localisation et cartographie simultanée a été introduite dans les travaux de [Smith and Cheeseman, 1986, Smith et al., 1990]. L'EKF a été donc la première approche à utiliser pour résoudre le problème de SLAM. Plusieurs travaux ont succédé pour améliorer le SLAM [Castellanos et al., 1999, Dissanayake et al., 2001, Durrant-Whyte and Bailey, 2006, Bailey and Durrant-Whyte, 2006, Durrant-Whyte et al., 2003]. L'EKF est une extension du filtre de Kalman qui prend en compte les systèmes non linéaires. En effet, les modèles de mouvement et d'observation sont, en général, non linéaires. L'EKF-SLAM est une approche Bayésienne récursive qui nécessite plusieurs hypothèses :

- La trajectoire du robot forment une chaîne de Markov de premier ordre. La position courante du robot x_t ne dépend que de la position précédente et de la commande u_t .
- Les observations des amers sont indépendantes entre elles.
- L'observation d'un amer à l'instant t ne dépend que de la position du robot à cet instant.
- Les mesures capteurs sont affectées d'un bruit blanc Gaussien.

Sous ces hypothèses, la formulation Bayésienne de l'EKF-SLAM est donnée par :

$$p(x_t, m | z_{1:t}, u_{1:t}) = \eta p(z_t | x_t, m) \int p(x_t | x_{t-1}, u_t) p(x_{t-1}, m | z_{1:t-1}, u_{1:t-1}, x_0) dx_{t-1} \quad (1.2)$$

, où η est une constante de normalisation.

a. Principe

Le principe de cette méthode consiste en la représentation de la position du robot et des amers par un vecteur d'état X . Les mesures étant affectées d'un bruit Gaussien, le vecteur d'état est donc une variable normale aléatoire. Par conséquent, on associe au vecteur d'état une matrice de variances-covariances P caractérisant les incertitudes sur les positions et les corrélations entre les variables du vecteur d'état. Les moyennes de cette matrice représentent la position du robot et celles des amers.

$$X = [xm_1 \ m_2 \ \dots \ m_n]^T \quad (1.3)$$

$$P = \begin{pmatrix} P_{xx} & p_{xm_1} & \dots & P_{xm_n} \\ p_{m_1x} & P_{m_1m_1} & \dots & p_{m_1m_n} \\ \vdots & \vdots & \vdots & \vdots \\ P_{m_nx} & p_{m_nm_1} & \dots & P_{m_nm_n} \end{pmatrix} \quad (1.4)$$

Avec :

x : la position courante du robot

P_{ij} : la covariance entre les variables i et j . Cette matrice est une approximation Gaussienne résultant de la linéarisation autour d'un point (état du système) supposé proche de la solution réelle. Au fur à mesure que le robot se déplace, le filtre de Kalman met à jour le vecteur d'état et la matrice de variances-covariances.

Dans la pratique, l'estimation de l'état du système s'effectue en deux temps :

- Prédiction : Dans cette phase, le modèle d'évolution prédit la position courante à partir de la position précédente et les données proprioceptives.
- Correction : La position prédictive du robot et la carte sont corrigées en utilisant les observations d'amers.

b. Avantages et inconvénients

Dans l'approche EKF-SLAM, les déterminants de chaque sous-matrice de la matrice de variances-covariances décroissent à chaque observation. L'incertitude sur la position du robot et les amers a donc tendance à diminuer avec le temps.

Cependant, l'inconvénient principal de l'EKF-SLAM reste sa complexité de calcul. La première hypothèse Markovienne, pour laquelle on ne prend en compte dans l'estimation que la dernière position du robot, implique que les amers deviennent corrélés entre eux. La matrice de variances-covariances devient de plus en plus dense. Cela requiert un stockage mémoire et un calcul quadratiques en nombre

d'amers. Cette complexité restreint l'utilisation de cette approche qu'à des zones relativement petites ou incluant quelques centaines d'amers [Thrun et al., 2004]. De nombreux travaux ont été effectués pour atténuer cette complexité en appliquant l'EKF-SLAM sur plusieurs sous-cartes [Bosse et al., 2003, Paz et al., 2007].

D'autre part, l'EKF-SLAM souffre du problème de consistance. Celle-ci peut être vue comme la qualité de l'incertitude fournie par le filtre par rapport à l'erreur réelle. L'inconsistance provient de la linéarisation approximative autour d'un état estimé plutôt que l'état réel [Julier and Uhlmann, 2001, Bailey et al., 2006, Huang and Dissanayake, 2007]. Pour améliorer la consistance, une variante de l'EKF appelée UKF (Unscented Kalman Filter) a été proposée [Wan and Van Der Merwe, 2000]. Celle-ci évite la linéarisation des modèles non linéaires pour améliorer la consistance.

1.3.2 Fast-SLAM

La forte complexité algorithmique de l'EKF-SLAM a poussé la communauté scientifique à chercher d'autres alternatives. Le FastSLAM [Montemerlo et al., 2002] utilise un filtre particulaire à la place du filtre de Kalman. Le filtre particulaire consiste à échantillonner la densité à posteriori de la trajectoire en un ensemble de particules. Chaque particule de la trajectoire est associée à un ensemble d'amers. Sous l'hypothèse stipulant que, connaissant la trajectoire, les observations sont indépendantes, la densité à posteriori de la trajectoire et de la carte est formulée comme suit :

$$p(x_{0:t}, m|z_{1:t}, u_{1:t}) = p(x_{0:t}|z_{1:t}, u_{1:t}, x_0) \prod_{i=1}^l p(m_i|x_{0:t}, z_{1:t}) \quad (1.5)$$

, où l est le nombre d'amers. Cette densité est le produit de la probabilité à posteriori de la trajectoire et l'estimateur des amers.

a. Principe

Dans la pratique, le principe du FastSLAM consiste à générer stochastiquement plusieurs hypothèses (particules) pour la trajectoire. Pour chaque particule, les amers peuvent être estimés indépendamment les uns des autres. On associe alors à chaque amer un filtre de Kalman étendu (EKF). Celui-ci permet d'estimer les paramètres de l'amer correspondant en utilisant les mesures d'observation. Globalement, les étapes à suivre dans le FastSLAM sont les suivantes :

- Échantillonnage : Génération de N particules. Le nombre de particules est à fixer de telle manière à réduire le temps de calcul, mais également à éviter l'épuisement des particules. Ce dernier peut conduire à une divergence dans la localisation.
- Pondération : Chaque particule est pondérée en fonction de sa vraisemblance avec les mesures.
- Ré-échantillonnage : Les particules sont ré-échantillonnées une seconde fois en fonction de leurs poids pour en éliminer les moins probables.
- Estimation de la carte : Pour chaque particule, on estime la carte correspondante.

b. Avantages et inconvénients

Par le biais d'un filtre particulaire, le FastSLAM réduit la complexité algorithmique par rapport à l'EKF-SLAM. Le fait d'appliquer un filtre de Kalman par amer donne une complexité de $O(NM)$, où N et M sont respectivement le nombre de particules et d'amers.

Cependant, cette approche souffre également du problème de consistance. La convergence du filtre particulaire dépend du nombre de particules générées. Si les mesures proprioceptives sont très bruitées, la plupart des particules sont éliminées lors de la phase de ré-échantillonnage. Cela peut conduire à une pauvre représentation en termes de particules et pose, donc, à un problème de consistance pour les particules restantes [Bailey et al., 2006, Zhang et al., 2009]. Notons que le FastSLAM 2.0 [Montemerlo et al., 2003] améliore le FastSLAM original en prenant en compte les mesures les plus récentes lors de la phase de ré-échantillonnage. De cette façon, le FastSLAM donne de meilleures performances en convergence tout en gardant le même nombre de particules.

1.3.3 GraphSLAM

Le GraphSLAM a été introduit pour la première fois par Lu et Milios en 1997 [Lu and Milios, 1997]. Cette approche constitue depuis une dizaine d'années un axe de recherche très actif au sein de la communauté de robotique. L'estimation de l'état du système est formulée par un problème d'optimisation. Ce dernier fait appel à plusieurs techniques issues essentiellement de l'algèbre linéaire et de la théorie du graphe. Le GraphSLAM ou SLAM basé optimisation de graphe modélise le problème de SLAM à l'aide d'un graphe. Comme l'illustre la figure 1.2, la trajectoire et la carte des amers sont représentées par des nœuds. Associées à un bruit Gaussien, les mesures capteurs donnent des contraintes spatiales entre les nœuds. Ces contraintes spatiales sont modélisées par des arêtes. On distingue deux types d'arêtes : arêtes de mouvement et arêtes d'observation. Une arête de mouvement relie deux nœuds robot (pose) consécutifs. Une arête d'observation provient d'une observation d'un amer. Un amer est relié à une pose (nœud robot) s'il a été observé depuis celle-ci.

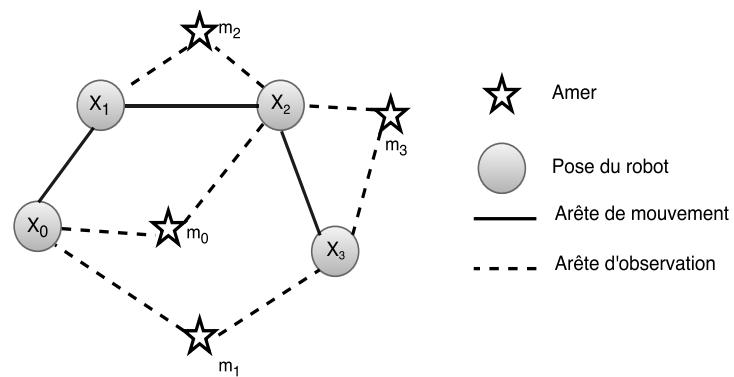


FIGURE 1.2 – Illustration du GraphSLAM.

Le GraphSLAM est une approche de lissage qui consiste, contrairement à l'EKF-SLAM, à réestimer, en plus de la position courante, toute la trajectoire depuis l'état initial en prenant en compte tout

l'historique des mesures. De même que les autres approches Bayésiennes, le GraphSLAM suppose que les bruits sont Gaussiens. Le GraphSLAM estime la densité $p(x_{1:t}, m|z_{1:t}, u_{1:t}, x_0)$. A la différence du filtre particulaire qui génère plusieurs hypothèses de trajectoire, le GraphSLAM modélise toutes les contraintes spatiales entre les nœuds. Dans la pratique, le GraphSLAM est divisé en deux parties (Fig. 1.3) : Front-end (partie en-amont) et Back-end (partie en-aval).

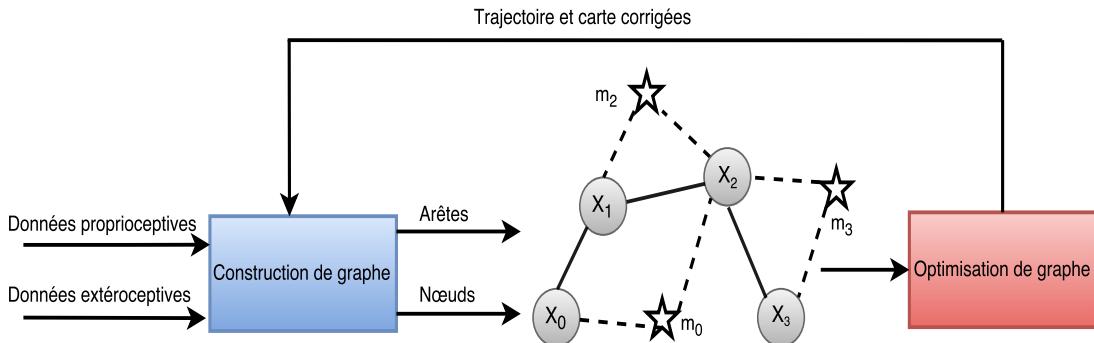


FIGURE 1.3 – Composantes du GraphSLAM : construction de graphe (Front-end) et optimisation de graphe (Back-end).

a. Front-end

Il s'agit principalement dans cette partie de traiter les données brutes des capteurs proprioceptifs et extéroceptifs afin de construire le graphe. A chaque nouvelle mesure proprioceptive ou extéroceptive, le graphe est augmenté en insérant de nouveaux nœuds et arêtes. Une mesure proprioceptive résulte en l'ajout d'une pose (nœud robot) associée à une arête de mouvement. Les mesures extéroceptives sont traitées pour en extraire des amers. Une observation d'un amer met à jour le graphe en ajoutant une arête d'observation et éventuellement l'amer en question s'il n'est pas encore dans le graphe. Par ailleurs, cette partie est aussi en charge de l'identification de fermetures de boucles. Une fermeture de boucle implique que le robot repasse par une zone déjà visitée. Elle permet ainsi d'augmenter la précision de localisation et de cartographie.

b. Back-end

Une fois le graphe construit, l'état du système peut être estimé via une optimisation globale du graphe. Celle-ci consiste à trouver une configuration (trajectoire et carte) qui correspond le mieux aux contraintes introduites par les arêtes. Ce problème d'optimisation peut être formulé, dans la pratique, par un problème de moindres carrés ou NLS (Nonlinear Least Squares). Ce dernier est souvent résolu par une méthode itérative (Gauss-Newton, Levenberg-Marquadt, Dog-Leg,...).

Ce travail de thèse se focalise sur l'optimisation de graphe. Il s'agit d'étudier et de réduire la complexité de l'optimisation de graphe indépendamment de la première partie (Front-end).

c. Avantages et inconvénients

Les approches de lissage comme le GraphSLAM, réestiment à chaque optimisation de graphe tout l'état du système. Celui-ci est raffiné à chaque nouvelle optimisation de graphe. La linéarisation est de même effectuée en utilisant la dernière estimée de l'état du système. Contrairement aux approches de filtrage, cela permet d'éviter la propagation des erreurs de linéarisation, et donc évite l'inconsistance des résultats.

La résolution du NLS associé au GraphSLAM finit par la résolution d'un système linéaire par blocs. Chaque bloc correspond à un élément dans le graphe (noeud, arête). De plus, ce système est, par construction, symétrique défini positif et souvent creux étant donné qu'en général les graphes obtenus sont épars. En termes de calcul, cela a l'avantage de permettre l'utilisation de méthodes éparses telles que Cholesky. Celle-ci réduit considérablement le temps de calcul. En termes de stockage mémoire et de consistance, le GraphSLAM s'avère plus adapté que l'EKF-SLAM et le FastSLAM pour le SLAM à grande-échelle.

Bien que le GraphSLAM fournisse des résultats de localisation et de cartographie consistants, sa complexité algorithmique reste très contraignante à cause de la caractéristique de lissage. D'autre part, la formulation du GraphSLAM par un problème de moindres carrés implique que le temps de calcul dépend de la rapidité de convergence¹. Celle-ci dépend principalement du point de linéarisation obtenu par les modèles de mouvement et d'observation. Plus le point de linéarisation est proche de l'état du système réel, plus la convergence est rapide.

Par ailleurs, la résolution du NLS dépend de plusieurs autres paramètres relatifs au graphe. Le nombre de noeuds et les connexions entre ceux-ci déterminent la structure du système linéaire par blocs. Le temps de résolution de ce système dépend étroitement des dimensions (nombre de noeuds), mais également de la répartition des blocs non nuls dans la matrice. Plus le graphe est dense, plus l'optimisation de graphe est coûteuse en temps.

De part ses avantages indéniables, l'utilisation du GraphSLAM connaît un grand essor. En parallèle de l'amélioration de la précision de localisation et de cartographie, de nombreux travaux de recherche ont traité de la complexité de calcul de cette méthode. L'objectif est de généraliser l'utilisation du GraphSLAM dans des systèmes temps réel à large-échelle.

1.3.4 SLAM ensembliste

Cette approche est basée sur l'analyse par intervalle. Celle-ci est un outil mathématique développé par Moore durant les années 1960s afin de réaliser des calculs garantis [Moore and Bierbaum, 1979]. En effet, le résultat d'un calcul n'est plus une simple valeur mais tout un intervalle contenant de manière sûre la valeur réelle. Cet intervalle est défini par une borne inférieure et une autre supérieure. Dans

1. La rapidité de convergence fait référence au nombre d'itérations nécessaire pour la convergence du processus itératif (Gauss-Newton).

cette approche, on considère que toutes les erreurs sont bornées et leurs bornes sont connues. Cette approche a été premièrement utilisé par [Di Marco et al., 2001]. L'idée est de représenter l'état du système par des pavés. La pose du robot est représentée par une boîte tridimensionnelle. Quant aux amers, ils sont modélisés par des parallélogrammes. Ce travail a été suivi par d'autres travaux pour améliorer cette approche afin de l'utiliser en environnement extérieur et intérieur [Seignez et al., 2005, 2009, Lambert et al., 2009, Vincke and Lambert, 2011], ou encore pour la localisation sous-marine [Jaulin, 2006, 2009, Le Bars et al., 2010, Sliwka et al., 2011].

Pour corriger la trajectoire et la carte, le SLAM ensembliste passe par quatre étapes [Vincke, 2012] durant lesquelles, le calcul par intervalle est appliqué :

- Prédiction de la position du robot
- Correction de la position du robot en utilisant les observations
- Initialisation des nouveaux amers
- Correction de la carte des amers

a. Avantages et inconvénients

L'objectif du SLAM ensembliste est de pallier au problème de consistance inhérent dans les approches Bayésiennes telles que l'EKF-SLAM et le FastSLAM. L'analyse par intervalle évite la linéarisation des modèles non linéaires, l'échantillonnage, mais également les erreurs dues aux imprécisions de calculs.

En dépit de son avantage en termes de garantie de calcul, le SLAM ensembliste reste très compliqué à mettre en œuvre pour des systèmes temps-réel à cause de la forte complexité de calcul due à l'analyse par intervalle.

1.3.5 Comparaison

Nous résumons dans le tableau 1.2 les plus importantes différences entre les méthodes de SLAM présentées auparavant. La méthode ensembliste se distingue du reste en utilisant l'analyse par intervalle. Bien que celle-ci permette de garantir les calculs, elle est très prohibitive en termes de temps de calcul. D'autre part, les méthodes probabilistes EKF-SLAM et FastSLAM souffrent du problème de consistance. Cela est dû à la linéarisation approximative autour d'un état de système estimé plutôt que l'état réel. Le GraphSLAM est une approche de lissage qui réestime, à chaque fois, toute la trajectoire et la carte des amers. Il permet de pallier au problème de consistance en relinéarisant autour de la dernière estimée de l'état du système. En termes de complexité, le GraphSLAM peut être très gourmand en temps de calcul en comparaison avec l'EKF-SLAM et le FastSLAM. Cependant, le caractère épars et la symétrie des représentations graphiques peuvent atténuer considérablement sa complexité. Cela n'est pas le cas pour l'EKF-SLAM particulièrement pour la consommation mémoire. La complexité de stockage mémoire est $O(k^2)$ contre $O(p + k + e)$ pour le GraphSLAM. En effet, on est obligé de garder en mémoire toute la matrice de variances-covariances pour l'EKF-SLAM. Dans la pratique,

	EKF-SLAM	FastSLAM	GraphSLAM	SLAM ensembliste
Approche	Probabiliste	Probabiliste	Probabiliste	
Filtrage/Lissage	Filtrage	Filtrage	Lissage	Ensembliste
Complexité	$O(K^2)$	$O(r \log k)$	Itérative, $O((k+p)^3/3)$	Itérative, très complexe
Consistance	-	+	+++	++++
Scalabilité	-	+	+++	+

TABLE 1.2 – Comparaison entre les méthodes de SLAM. p : nombre de poses robot. K : nombre d'amers. r : nombre de particules.

les systèmes de SLAM basés EKF ne dépassent pas, en général, les centaines d'amers. Le GraphSLAM peut, par contre, traiter des dizaines de milliers de nœuds. **Comparé à l'EKF-SLAM et le FastSLAM**, le GraphSLAM est scalable. Cela revient principalement à sa consistance et sa consommation mémoire. Cependant, dans la pratique, les avantages du GraphSLAM sont contrariés par les ressources limitées des architectures de calcul embarquées. Il serait donc intéressant d'améliorer cette scalabilité, en réduisant la complexité de calcul, pour pouvoir localiser le robot (ou reconstruire l'environnement) sur de grandes zones.

1.4 Présentation du GraphSLAM

1.4.1 Définitions

Nous donnons dans cette section quelques définitions liées au GraphSLAM.

a. Les mesures dans le GraphSLAM

Au cours de son déplacement, le robot effectue deux types de mesures à l'aide de capteurs proprioceptifs et extéroceptifs. Une mesure z_{ij} dans le GraphSLAM fait référence à une distance spatiale séparant deux nœuds.

- Mesure proprioceptive : permet de calculer, suite à un déplacement, la distance spatiale entre deux positions consécutives du robot. Cette mesure est généralement obtenue par des odomètres ou une centrale inertuelle.
- Mesure extéroceptive : permet de calculer la distance spatiale entre la position du robot et celle de l'amer observé. Elle peut être donnée, par exemple, par une caméra ou un télémètre Laser (angle et profondeur).

b. Caractérisation d'une arête

Une arête reliant les nœuds y_i et y_j (Fig. 1.4) est caractérisée par son erreur e_{ij} ainsi qu'une matrice d'information Ω_{ij} pour la mesure.

- Erreur e_{ij} : l'erreur est la différence entre la mesure donnée par le capteur z_{ij} et la mesure prédictée \hat{z}_{ij} calculée à partir de la configuration courante des nœuds connectés en utilisant les modèles de mouvement et d'observation. La fonction d'erreur est appliquée à tout type d'arêtes.

$$e_{ij} = e(y_i, y_j) = \hat{z}_{ij} - z_{ij} \quad (1.6)$$

- Matrice d'information Ω_{ij} : cette matrice est l'inverse de la matrice de variances-covariances caractérisant le bruit sur la mesure capteur z_{ij} .

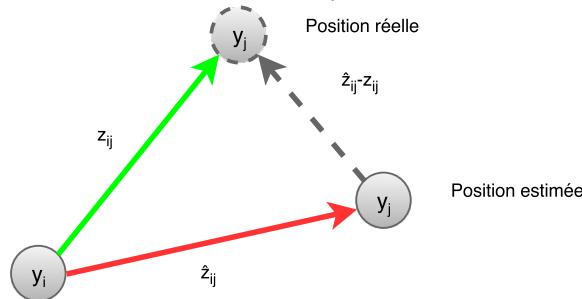


FIGURE 1.4 – Erreur de mesure entre la pose x_i et la pose x_j . L'opérateur de soustraction dépend du modèle de mouvement (ou d'observation).

1.4.2 GraphSLAM comme un problème de moindres carrés

Les bruits sur les mesures, dans le GraphSLAM, sont supposés Gaussiens. Une contrainte C_{ij} entre deux nœuds i et j dans le graphe est donnée par la formule suivante² [Grisetti et al., 2010] :

$$\underbrace{[\hat{z}_{ij} - z_{ij}]^T}_{e_{ij}^T} \Omega_{ij} \underbrace{[\hat{z}_{ij} - z_{ij}]}_{e_{ij}} \quad (1.7)$$

avec Ω_{ij} l'inverse de la matrice de variances-covariances caractérisant le bruit sur le modèle de mesure entre les nœuds i et j . L'optimisation de graphe consiste à trouver une configuration de graphe y^* qui minimise la fonction de coût $F(y)$:

$$F(y) = \sum_{ij} e_{ij}^T \Omega_{ij} e_{ij} \quad (1.8)$$

$$= \sum_{ij} F_{ij} \quad (1.9)$$

Pour trouver y^* , on résout le problème de moindres carrés suivant :

$$y^* = \operatorname{argmin}_y F(y) \quad (1.10)$$

2. Nous donnons dans l'annexe A une description détaillée de la formulation probabiliste du GraphSLAM.

1.4.3 Résolution du problème de moindres carrés

Le problème de moindres carrés (Eq. 1.10) ne peut être résolu par une méthode directe du fait que l'équation (Eq. 1.8) est non linéaire en y . Dans la pratique, on utilise une approche itérative. Celle-ci part d'une solution initiale supposée proche de la solution optimale. Cette approche réitère la résolution jusqu'à convergence vers une solution y^* que l'on espère optimale. Cela n'est cependant pas garanti car la fonction F (Eq. 1.10) n'est pas convexe. Parmi ces approches itératives, on cite : Gradient Conjugué, Gauss-Newton (GN), Levenberg-Marquardt (LM) et Dog-Leg [Powell, 1970, Mad-sen et al., 2004]. La méthode de Gauss-Newton et sa variante Levenberg-Marquardt (LM) restent les plus utilisées pour résoudre le problème de moindres carrés dérivés du GraphSLAM [Fletcher, 1987, Thrun and Montemerlo, 2006, Grisetti et al., 2010, Kummerle et al., 2011]. Récemment, on a commencé à utiliser l'approche Dog-Leg [Lourakis and Argyros, 2005, Rosen et al., 2012, Kaess et al., 2012]. Dans ce travail, nous avons choisi d'utiliser la méthode de Gauss-Newton. Nous allons montrer les différents étapes³ à suivre dans le processus de Gauss-Newton pour minimiser la fonction F .

Comme mentionné auparavant, la fonction F (Eq. 1.8) n'est malheureusement pas linéaire en y . Pour linéariser F , soit \check{y} un point de linéarisation supposé proche de la trajectoire et la carte réelles. L'idée est d'approximer la fonction d'erreur par son développement de Taylor au premier ordre. On a :

$$e_{ij}(\check{y}_i + \Delta y_i, \check{y}_j + \Delta y_j) = e_{ij}(\check{y} + \Delta y) \quad (1.11)$$

$$\simeq e_{ij} + J_{ij} \Delta y \quad (1.12)$$

avec J_{ij} la Jacobienne de $e_{ij}(y)$ calculée en \check{y} et $e_{ij} = \text{def } e_{ij}(\check{y})$. On remplace cette expression dans les termes F_{ij} de la fonction F (Eq.1.8) :

$$F_{ij}(\check{y} + \Delta y) = e_{ij}(\check{y} + \Delta y)^T \Omega_{ij} e_{ij}(\check{y} + \Delta y) \quad (1.13)$$

$$\simeq (e_{ij} + J_{ij} \Delta y)^T \Omega_{ij} (e_{ij} + J_{ij} \Delta y) \quad (1.14)$$

$$= \underbrace{e_{ij}^T \Omega_{ij} e_{ij}}_{c_{ij}} + 2 \underbrace{e_{ij}^T \Omega_{ij} J_{ij}}_{b_{ij}} \Delta y + \Delta y^T \underbrace{J_{ij}^T \Omega_{ij} J_{ij}}_{H_{ij}} \Delta y \quad (1.15)$$

$$= c_{ij} + 2 b_{ij} \Delta y + \Delta y^T H_{ij} \Delta y \quad (1.16)$$

Maintenant, on réécrit la fonction F (Eq. 1.8) avec cette approximation :

$$F(\check{y} + \Delta y) = \sum_{ij} F_{ij}(\check{y} + \Delta y) \quad (1.17)$$

$$\simeq \sum_{ij} c_{ij} + 2 \sum_{ij} b_{ij} \Delta y + \Delta y^T \sum_{ij} H_{ij} \Delta y \quad (1.18)$$

$$= c + 2 b \Delta y + \Delta y^T H \Delta y \quad (1.19)$$

où $c = \sum_{ij} c_{ij}$, $b = \sum_{ij} b_{ij}$, $H = \sum_{ij} H_{ij}$.

3. La description des étapes est une adaptation du tutoriel de [Grisetti et al., 2010].

En exploitant l'équation (Eq. 1.19), la fonction F (Eq. 1.8) peut être minimisée en résolvant le système linéaire suivant :

$$H \Delta y = -b \quad (1.20)$$

H est appelé matrice d'information ; b , quant à lui, est appelé vecteur d'information. Une fois le système résolu, la configuration \hat{y} est mise à jour par la formule⁴ :

$$y^* = \hat{y} + \Delta y \quad (1.21)$$

Le processus de Gauss-Newton (ou Levenberg-Marquardt) réitère la linéarisation, la résolution (Eq. 1.20) et la mise à jour (Eq. 1.20) jusqu'à convergence. A chaque étape, la solution précédente est utilisée comme point de linéarisation. Il arrive parfois qu'une itération du processus GN génère une solution qui est plus mauvaise que le point de linéarisation (ou la configuration initiale). Dans ce cas, le processus peut ne pas converger. Pour régler ce problème, on peut utiliser la variante LM qui permet de restaurer la solution de l'étape précédente. A chaque étape, au lieu de résoudre le système (Eq. 1.20), LM résout le système suivant :

$$(H + \lambda I) \Delta y = -b \quad (1.22)$$

avec $\lambda \in [0, 1]$. Plus λ est grand, plus les incrément sont petits (changement minime sur y). En général, l'algorithme LM converge. Mais, le minima (solution donnée) peut ne pas être global. D'autre part, la convergence de l'algorithme de Gauss-Newton n'est pas garanti et dépend principalement de la solution initiale (point de linéarisation). Plus celle-ci est proche de la solution optimale, plus l'algorithme converge plus vite.

Finalement, bien que notre travail se focalise sur l'algorithme de Gauss-Newton, notons qu'en termes d'implantation, le passage de l'algorithme GN à LM est direct et ne nécessite qu'une étape supplémentaire pour mettre à jour la diagonale de la matrice d'information.

1.4.4 Algorithme GraphSLAM et partitionnement en blocs fonctionnels

Nous allons présenter dans cette section l'instance algorithmique du GraphSLAM adoptée dans ce travail ainsi que son partitionnement en blocs fonctionnels.

a. Algorithme global

L'instance algorithmique utilisée dans cette thèse a été principalement tirée des travaux de [Thrun and Montemerlo, 2006, Grisetti et al., 2010, Kummerle et al., 2011]. La structure générale de cette instance algorithmique a été reprise dans la plupart des Frameworks de l'état de l'art [Kummerle et al.,

4. L'opérateur d'addition correspond aux modèles de mouvement et d'observation.

2011, Konolige et al., 2010b, Kaess et al., 2008, 2012]. L'algorithme 1 présente les différentes étapes de cette instance. Une étape d'optimisation de graphe est précédée par l'acquisition puis le traitement des données capteurs dans la partie Front-end du GraphSLAM. Cette dernière fournit en sortie, en plus d'une mesure proprioceptive, les différentes observations d'amers. Chaque observation est associée à un amer identifié. Une mise à jour de la structure du graphe est ensuite effectuée. Le graphe est augmenté en insérant la nouvelle pose, l'arête de mouvement associée, les arêtes d'observation et éventuellement de nouveaux amers. La correction de l'état du système est incrémentale. Cela implique que l'état du système peut être corrigé à chaque fois que l'on met à jour le graphe.

Algorithme 1 : GraphSLAM incrémental

```

1 while true do
2   DATA  $\leftarrow$  Sensor data acquisition;
3   updateGrapheStructure(G,DATA);
4   while  $\neg$  converged do
5     computeError(G);
6     if converged then
7       | break ;
8     end
9     ( $H, b$ )  $\leftarrow$  constructSystem(G) ;
10     $H_{00} \leftarrow H_{00} + I$ ;
11     $\Delta y \leftarrow sparseSolver(H, -b)$ ;
12    updateSystemState(G, $\Delta y$ );
13     $H_{00} \leftarrow H_{00} - I$ ;
14  end
15 end
```

Une fois le graphe mis à jour, une étape d'optimisation de graphe est lancée. On calcule, en premier lieu, les erreurs des arêtes. Il peut arriver qu'une configuration initiale du graphe (ou après un certain nombre d'itérations de Gauss-Newton) minimise déjà la somme des contraintes sur les arêtes (ou convergence). Dans ce cas, plus aucune optimisation de graphe n'est entreprise et on attend la réception des nouvelles mesures capteurs. En cas de non convergence, on commence la construction du système linéaire. Les erreurs sont tout d'abord linéarisées en calculant leurs Jacobiennes. En parallèle de la linéarisation, on peut remplir la matrice d'information H et le vecteur d'information b du système $H\Delta y = -b$.

Le système obtenu ne peut être résolu. En effet, le déterminant de la matrice sera nul. En réalité, on peut avoir une infinité de solutions. Supposons qu'il existe une solution unique. Avec celle-ci, on peut dessiner le graphe dans un repère 2D (ou 3D). Dans ce cas, chaque transformation rigide donne une configuration (positions de nœuds) qui conserve les distances entre les nœuds tout en étant, en même temps, une solution au système linéaire [Thrun and Montemerlo, 2006, Grisetti et al., 2010]. Afin d'éviter ce problème, il faut conditionner la matrice en fixant au moins un nœud. Intuitivement, ce nœud sera le nœud initial représentant la pose de départ du robot. En général, on choisit $(0, 0, 0)$ comme pose initiale. Pour fixer cette dernière, on ajoute la matrice d'identité à l'élément H_{00} de la matrice d'information H . Une autre alternative consiste à supprimer la première ligne et la première colonne du système. En résolvant le système linéaire, les incrémentations sont utilisées pour mettre à jour

les nœuds du graphe et ainsi obtenir une nouvelle estimée de l'état du système. Cette procédure est réitérée jusqu'à convergence ou jusqu'à satisfaction d'un critère d'arrêt.

b. Repères et modèle d'évolution

L'instance algorithmique adoptée dans ce travail considère un robot se déplaçant sur un plan bimensionnel. Le robot est équipé de deux odomètres pour mesurer le déplacement ainsi qu'un capteur extéroceptif (une caméra frontale ou un télémètre Laser). On définit trois repères illustrés dans la figure 1.5 :

- Repère Global : $R_{glob}(\vec{x}_{glob}, \vec{y}_{glob}, \vec{z}_{glob})$ définit par la position et l'orientation initiales du robot.
- Repère mobile : $R_{mob}(\vec{x}_{mob}, \vec{y}_{mob}, \vec{z}_{mob})$ donne la position d'un amer par rapport à la position courante du robot.

On considère que la troisième dimension z_{glob} du repère global est nulle. Dans le cas échéant (z_{glob} non nulle), on a affaire au SLAM 3D où l'on doit utiliser des quaternions pour représenter la pose du robot dans un espace 3D. Dans le repère global R_{glob} , la pose du robot est donc donnée par :

- Ses coordonnées 2D (x, y).
- Son orientation θ représentant l'angle orienté formé par le vecteur \vec{x}_{glob} et \vec{x}_{mob} .



FIGURE 1.5 – Repère global et mobile.

Pour mettre en œuvre le GraphSLAM, on a besoin de passer du repère global au repère mobile et inversement. Pour ce faire, il faut effectuer des transformations $(T_{glob,mob}, T_{mob,glob})$. Soit $(x_{glob}, y_{glob}, z_{glob})$ un point exprimé dans le repère global R_{glob} . Ses coordonnées $(x_{mob}, y_{mob}, z_{mob})$ dans le repère mobile R_{mob} sont :

$$\begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} = T_{glob,mob}(x_{glob}, y_{glob}, z_{glob}) \quad (1.23)$$

$$= \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} - \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \right) \quad (1.24)$$

Le passage du repère mobile au repère global est donnée par :

$$\begin{pmatrix} x_{glob} \\ y_{glob} \\ z_{glob} \end{pmatrix} = T_{mob,glob}(x_{mob}, y_{mob}, z_{mob}) \quad (1.25)$$

$$= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_{mob} \\ y_{mob} \\ z_{mob} \end{pmatrix} + \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \quad (1.26)$$

Soit $\Delta X_t = (\nabla x_t, \nabla y_t, \nabla \theta_t)$ la mesure proprioceptive à l'instant t . Celle-ci peut être obtenue en exploitant les données odométriques. La mesure ΔX_t est en réalité une pose exprimée dans le repère mobile R_{mob} . Pour avoir la pose à l'instant t , on passe dans le repère global. En omettant Z_{glob} , X_t est calculée comme suit :

$$\Delta X_t = \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} R(\nabla x_t \nabla y_t)^T \\ \nabla \theta_t \end{pmatrix} + \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} \quad (1.27)$$

$$= \begin{pmatrix} \cos(\theta_{t-1}) \nabla x_t - \sin(\theta_{t-1}) \nabla y_t + x_{t-1} \\ \sin(\theta_{t-1}) \nabla x_t + \cos(\theta_{t-1}) \nabla y_t + y_{t-1} \\ \nabla \theta_t + \theta_{t-1} \end{pmatrix} \quad (1.28)$$

R désigne la matrice de rotation de la pose du robot à l'instant $(t-1)$:

$$R = \begin{pmatrix} \cos(\theta_{t-1}) & -\sin(\theta_{t-1}) \\ \sin(\theta_{t-1}) & \cos(\theta_{t-1}) \end{pmatrix} \quad (1.29)$$

c. Partitionnement en blocs fonctionnels

Un bloc fonctionnel peut être défini comme un ensemble d'instructions destinées à accomplir, en un temps fini, le traitement d'une tâche donnée de l'algorithme. Pour permettre une meilleure analyse et implantation de l'algorithme, celui-ci est partitionné en blocs fonctionnels. L'objectif est d'apporter des optimisations logicielles et/ou matérielles pour accélérer les traitements du GraphSLAM. Le partitionnement est donc effectué en prenant en considération les paramètres suivants :

- Minimisation du transfert de données : le but est de réduire autant que possible le transfert de données entre blocs fonctionnels. Nous favorisons, d'autre part, le transfert par bloc. Il est plus intéressant, par exemple, de transférer une matrice de 10Mo que d'envoyer ses éléments, un par un, à la demande d'un autre bloc fonctionnel. Par conséquent, d'un point de vue communication, nous regroupons les petites tâches calculatoires dans un même bloc fonctionnels si celles-ci requièrent un grand mouvement de données entre elles.

- Dépendance de données : en termes de parallélisation, il est important que les blocs soient homogènes. Le fait d'avoir une partie séquentielle et une autre parallèle dans un même bloc compliquerait l'optimisation de celui-ci sur des architectures parallèles.
- Type de calcul : il est judicieux de séparer les blocs selon le type d'opérations. Nous regroupons les séquences dont l'interaction avec la mémoire est très dense dans un même bloc fonctionnel (e.g. construction du système CCS). Un tel partitionnement permet d'investiguer des optimisations relatives à l'accès aux données en mémoire.

Nous allons nous focaliser sur l'optimisation de graphe (la partie Back-end du GraphSLAM). La figure 1.6 présente le découpage en blocs fonctionnels. Après la mise à jour de la structure du graphe dans FB1, on commence une étape d'optimisation de graphe (FB2 à FB8). Celle-ci peut nécessiter plusieurs itérations de l'algorithme Gauss-Newton. Nous décrivons, dans ce qui suit, les blocs fonctionnels.

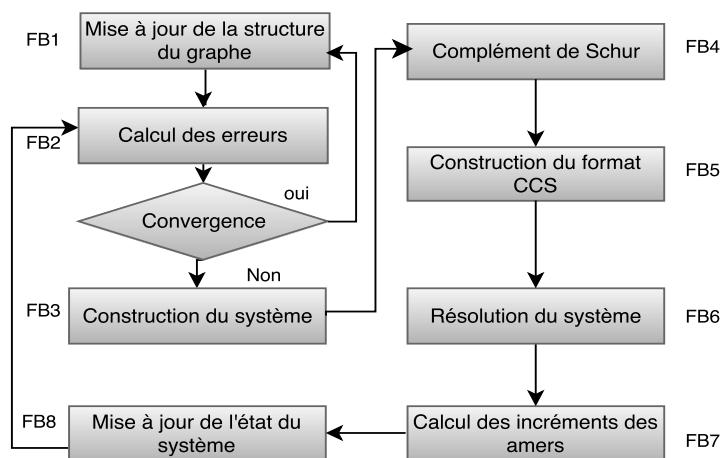


FIGURE 1.6 – Partitionnement en blocs fonctionnels.

1.4.5 Calcul des erreurs

Il s'agit dans ce bloc de calculer les erreurs sur les arêtes. Nous avons vu auparavant comment la fonction d'erreur est exploitée dans l'optimisation de graphe. Nous allons maintenant détailler les équations de cette fonction en prenant en considération le modèle de déplacement et d'observation ainsi que les paramétrisations correspondantes.

a. Erreur de mouvement

Soit l'arête odométrique (X_i, X_j) . Les poses courantes des deux nœuds dans le repère global sont respectivement (x_i, y_i, θ_i) et (x_j, y_j, θ_j) . La mesure proprioceptive correspondante est $Z_{ij} = (\Delta x_{ij}, \Delta y_{ij}, \Delta \theta_{ij})$. Pour calculer l'erreur, il faut exprimer la pose X_j dans le repère mobile dont l'origine est X_i . On calcule, ensuite, la différence entre cette position, que l'on appelle mesure prédictive, et la mesure donnée par le capteur Z_{ij} . Pour des raisons de simplicité, on pose :

$$X_i^T = (t_i^T, \theta_i)^T \quad (1.30)$$

$$Z_{ij}^T = (t_{ij}^T, \theta_{ij})^T \quad (1.31)$$

$$R_i = \begin{pmatrix} \cos \theta_i & -\sin \theta_i \\ \sin \theta_i & \cos \theta_i \end{pmatrix} \quad (1.32)$$

, où R_i désigne la matrice de rotation correspondante à la pose X_i . La fonction d'erreur de mouvement est donc définie par :

$$e_{ij} = T_{glob,mob}[x_j, y_j, \theta_j] - Z_{ij} \quad (1.33)$$

$$= \begin{pmatrix} R_{ij}^T(R_i^T(t_j - t_i) - t_{ij}) \\ \theta_j - \theta_i - \theta_{ij} \end{pmatrix} \quad (1.34)$$

Par ailleurs, soient A_{ij} et B_{ij} les Jacobiennes de la fonction d'erreur de mouvement par rapport à X_i et X_j (respectivement). Les fonctions analytiques des Jacobiennes sont données par :

$$A_{ij} = \frac{\partial e_{ij}(x_i, x_j)}{\partial x_i} = \begin{pmatrix} -R_{ij}^T R_i^T & R_{ij}^T \frac{\partial R_i^T}{\partial \theta_i} (t_j - t_i) \\ 0^T & -1 \end{pmatrix} \quad (1.35)$$

$$B_{ij} = \frac{\partial e_{ij}(x_i, x_j)}{\partial x_j} = \begin{pmatrix} R_{ij}^T R_i^T & 0 \\ 0^T & 1 \end{pmatrix} \quad (1.36)$$

b. Erreur d'observation

Soit l'arête odométrique (X_i, m_j) . Les poses courantes des deux nœuds dans le repère global sont respectivement (x_i, y_i, θ_i) et $(x_j, y_j, z_j)^T = (t_j, z_j)$. La mesure extéroceptive correspondante est $Z_{ij} = (x_{ij}, y_{ij}, z_{ij})^T = (t_{ij}, z_{ij})^T$. Pour calculer la mesure prédite, on effectue un changement de repère de R_{glob} vers R_{mob} pour la position de l'amer.

La fonction d'erreur d'observation est donnée par la formule suivante :

$$e_{ij} = T_{glob,mob}[x_j, y_j, \theta_j] - Z_{ij} \quad (1.37)$$

$$= \begin{pmatrix} R_i^T(t_j - t_i) - t_{ij} \\ z_j - z_{ij} \end{pmatrix} \quad (1.38)$$

Les évaluations effectuées dans ce travail ont porté sur des jeux de données 2D. La troisième dimension est donc nulle. Dans ce cas, les Jacobiennes correspondantes A_{ij} et B_{ij} sont calculées comme suit :

$$A_{ij} = \frac{\partial e_{ij}(x_i, m_j)}{\partial y_i} = \begin{pmatrix} -R_i^T & \frac{\partial R_i^T}{\partial \theta_i} (t_j - t_i) \\ 0^T & 1 \end{pmatrix} \quad (1.39)$$

$$B_{ij} = \frac{\partial e_{ij}(y_i, m_j)}{\partial y_j} = \begin{pmatrix} R_i^T & 0 \\ 0^T & 1 \end{pmatrix} \quad (1.40)$$

1.4.6 Construction du système linéaire

Ce bloc construit le système linéaire $H\Delta y = -b$. En d'autres termes, on calcule la matrice d'information H et le vecteur d'information b . H est une matrice $(p+n)x(p+n)$ où p est le nombre de poses et n est le nombre d'amers dans le graphe. Chaque élément de la matrice d'information est lui-même une matrice $d_i \times d_j$. d_i et d_j désignent les degrés de liberté des nœuds concernés (intersection ligne/colonne). Chaque bloc dans la matrice H correspond à un élément dans le graphe (nœud ou arête). En réordonnant les variables de telles manière à mettre les amers après les poses, la matrice H peut être partitionnée en quatre sous-matrices :

$$H = \begin{pmatrix} H_{pp} & H_{pl} \\ H_{lp} & H_{ll} \end{pmatrix} \quad (1.41)$$

La sous-matrice H_{pp} contient les blocs correspondant aux poses ainsi qu'aux arêtes de mouvement. H_{pl} comprend les blocs relatifs aux arêtes d'observation. H_{ll} , quant à elle, stocke les blocs relatifs aux amers. Si une pose et un amer sont respectivement représentés par (x, y, θ) et (x, y) dans un repère 2D, alors un bloc de H_{pp} est une matrice 3×3 . Un bloc de H_{ll} est une matrice 2×2 et un bloc de H_{pl} est une matrice 3×2 . D'autre part, le vecteur d'information b est un vecteur de $(p+n)$ blocs.

La construction du système s'effectue incrémentalement. En effet, chaque contrainte représentée par une arête contribue, d'une manière additive, au remplissage de la matrice H et du vecteur b . Les contraintes sont traitées une à une sans nécessairement tenir compte de l'ordre d'insertion dans le graphe. Initialement, la matrice H et le vecteur b sont à zéro. La figure 1.7 illustre le processus de remplissage. L'arête issue de l'observation de l'amer m_1 depuis x_1 entraîne la mise à jour des éléments $H(x_1, x_1), H(m_1, m_1), H(x_1, m_1), H(m_1, x_1)$. De même, le mouvement de x_1 à x_2 provoque la mise à jour des éléments $H(x_1, x_1), H(x_2, x_2), H(x_1, x_2)$ et $H(x_2, x_1)$. Après plusieurs étapes, on obtient la matrice de la figure 1.7c. En parallèle de la construction de H , pour chaque arête, les éléments correspondant aux nœuds en question sont mis à jour dans le vecteur b . A la fin, on obtient par construction un système à caractère épars. Les éléments hors diagonaux de H sont tous nuls avec deux exceptions :

- Entre deux poses consécutives x_{t-1} et x_t car il y a obligatoirement une contrainte de mouvement
- Entre un amer m_j et une pose x_t si l'amer a été observé depuis x_t

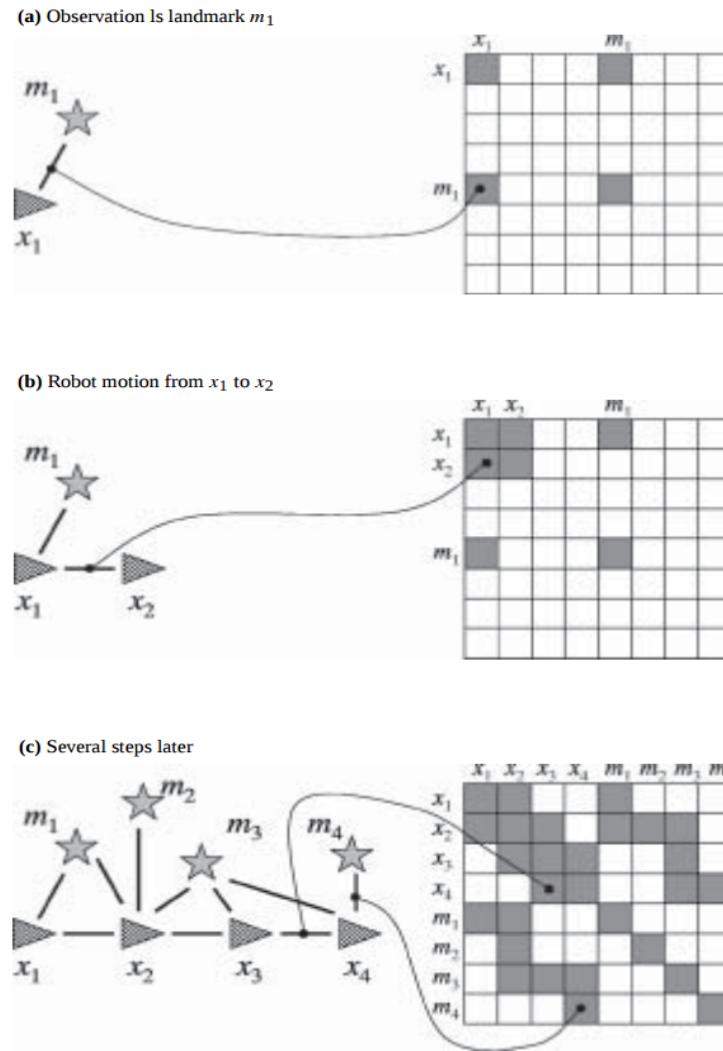


FIGURE 1.7 – Construction incrémentale de la matrice d'information [Thrun and Montemerlo, 2006].

Tous les éléments entre deux amers sont nuls. Cela est dû au fait qu'on suppose que les observations sont indépendantes les unes des autres. Par ailleurs, la matrice est, par construction, symétrique définie positive. Ces caractéristiques seront d'une immense importance lors de la résolution du système d'équation. La symétrie permet également de se limiter au calcul de la partie triangulaire supérieure (ou inférieure) de la matrice H .

1.4.7 Marginalisation des amers et complément de Schur

Dans [Frese, 2005] la marginalisation consistait à éliminer des entrées de la matrice de variances-covariances (inverse de la matrice d'information) dans le but de diminuer la densité du système (le rendre plus épars). Les auteurs prouvent théoriquement qu'une entrée correspondant à deux amers, dans la matrice inverse du système, décroît exponentiellement avec la distance parcourue entre les deux observations. Ils prouvent, donc, qu'il est possible d'approximer une matrice dense par une autre épars en éliminant les entrées hors-diagonales des amers distants.

La marginalisation appliquée dans ce travail est similaire à celles décrites dans [Thrun and Montemerlo, 2006, Dellaert and Kaess, 2006, Konolige and Agrawal, 2008, 2007, Sibley et al., 2007, Kummerle et al., 2011]. Le système linéaire construit est constitué de deux types de variables (Eq. 1.42) : poses (p) et amers (l). Le complément de Schur dans le GraphSLAM consiste à réduire le graphe en marginalisant les amers. La marginalisation d'un amer implique la connexion deux à deux les poses depuis lesquelles cet amer a été observé. Le graphe réduit peut être également obtenu par une mise en correspondance directe des faisceaux Laser entre deux poses donnant, ainsi, la distance relative entre ces poses. [Konolige and Agrawal, 2008] ont proposé un système de SLAM visuel appelé FrameSLAM. Celui-ci utilise une caméra stéréo pour estimer le mouvement du robot et reconstruire la carte de l'environnement. Les auteurs considèrent deux types de variables frames (images acquises par la caméra) et les amers détectés. Pour efficacement résoudre de larges systèmes, leur technique consiste à utiliser un squelette du graphe (skeleton graph). Celui-ci est composé d'un sous-ensemble de frames suite à la marginalisation des amers.

La figure 1.8 illustre la procédure de marginalisation des amers que l'on peut appeler aussi réduction de graphe. Celle-ci est réalisée en utilisant le complément de Schur [Thrun and Montemerlo, 2006]. Mathématiquement, le complément de Schur consiste à réduire la taille du système en utilisant l'équation 1.43. Il transforme le système construit en un nouveau système plus petit, et ne contenant que les poses.

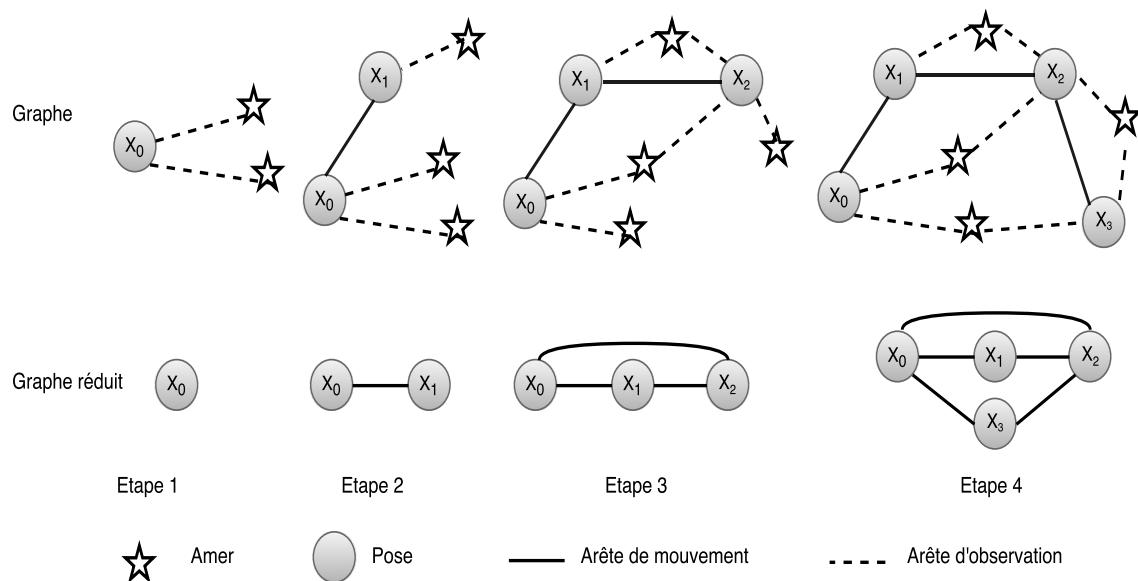
$$\begin{pmatrix} H_{pp} & H_{pl} \\ H_{lp} & H_{ll} \end{pmatrix} \begin{pmatrix} \Delta y_p \\ \Delta y_l \end{pmatrix} = \begin{pmatrix} -b_p \\ -b_l \end{pmatrix} \quad (1.42)$$

$$(H_{pp} - H_{pl}H_{ll}^{-1}H_{pl}^T)\Delta y_p = -b_p + H_{pl}H_{ll}^{-1}b_l \quad (1.43)$$

Il est à noter que le complément de Schur est appliqué, en général, pour réduire la taille du système, et ainsi son temps de résolution. Néanmoins, la marginalisation des amers génère de nouvelles contraintes entre les poses. Le nouveau système peut, donc, être moins épars, voire très dense par rapport au système initial. Le temps de résolution peut alors augmenter et devient très important. D'autre part, ce phénomène de remplissage peut être masqué par la forte réduction de la taille du système si le nombre d'amers est très grand par rapport au nombre de poses tel que dans [Konolige and Agrawal, 2008].

1.4.8 Construction du format CCS

Dans le cas de matrices creuses, il est intéressant de ne garder en mémoire que les valeurs non nulles. Pour ce faire, les matrices creuses ont des formats de stockage particuliers. Dans le GraphSLAM, on a affaire à un système linéaire dont la matrice est symétrique et souvent creuse. Pour préparer la matrice d'information pour un solveur épars utilisant Cholesky, nous avons opté pour le format CCS (Compressed Column Storage) [Duff et al., 1989]. Celui-ci ne nécessite aucune condition sur la matrice. Il

**FIGURE 1.8 – Exemple illustratif de la réduction de graphe.**

$\begin{pmatrix} 1 & 0 & 4 & 0 \\ 0 & 5 & 0 & 2 \\ 0 & 0 & 0 & 1 \\ 6 & 8 & 0 & 0 \end{pmatrix}$	<table border="1"> <thead> <tr> <th>Col.</th><th>0</th><th>2</th><th>4</th><th>5</th><th>7</th></tr> </thead> <tbody> <tr> <th>Lig.</th><td>0</td><td>3</td><td>1</td><td>3</td><td>0</td></tr> <tr> <th>Val.</th><td>1</td><td>6</td><td>5</td><td>8</td><td>4</td></tr> <tr> <td></td><td></td><td></td><td>1</td><td>2</td><td>1</td></tr> </tbody> </table>	Col.	0	2	4	5	7	Lig.	0	3	1	3	0	Val.	1	6	5	8	4				1	2	1
Col.	0	2	4	5	7																				
Lig.	0	3	1	3	0																				
Val.	1	6	5	8	4																				
			1	2	1																				

TABLE 1.3 – Exemple d'un format CCS.

a le mérite d'être le plus économique en termes d'encombrement mémoire. Cependant, les éléments de la matrice sont accédés via une indirection. La mémorisation se fait par colonne et nécessite trois tableaux : un pour les valeurs non nuls et les deux autres pour leurs indices correspondants.

Soit A la matrice donnée dans la figure 1.3. La matrice A est parcourue par colonne. Le tableau Val contient les éléments non nuls. Pour chaque élément de Val, le tableau Lig contient l'indice de ligne, dans A , de cet élément. Le tableau Col contient les indices, dans Val, du premier élément de chaque colonne.

1.4.9 Résolution du système

Il s'agit dans ce bloc de résoudre le système linéaire (Eq. 1.20). Pour ce faire, on peut intuitivement choisir d'inverser la matrice d'information comme dans [Lu and Milios, 1997]. Cette solution est très coûteuse en temps de calcul. Celui-ci croît cubiquement avec le nombre de noeuds. Dans la pratique, on peut avoir de très larges systèmes (pour des dizaines de milliers de noeuds). L'inversion est, par conséquent, très contraignante. [Gutmann and Konolige, 1999] ont proposé une méthode qui construit incrémentalement la carte d'environnement. La technique d'inversion de matrice est utilisée à des étapes prédéterminées.

Pour pallier au problème d'inversion, on peut recourir à d'autres techniques de l'algèbre linéaire. En effet, le système peut être résolu en utilisant des méthodes directes dites de factorisation telles que

QR, LU et Cholesky. On peut également utiliser des méthodes itératives telles que la relaxation de Gauss-Seidel [Duckett et al., 2000, Howard et al., 2001, Thrun et al., 2004, Thrun and Liu, 2005] et la méthode du Gradient Conjugué [Konolige, 2004]. Dans la pratique, la convergence dépend de la première initialisation (pré-conditionnement). Plus celle-ci est proche de la solution exacte, plus la convergence est rapide. Pour accélérer la convergence, [Konolige, 2004] utilisent une factorisation incomplète de Cholesky afin de pré-conditionner le système.

Les méthodes itératives ont l'avantage d'être moins coûteuses en termes d'espace mémoire, mais peuvent souffrir du problème de convergence. D'autre part, les méthodes directes donnent des solutions exactes au détriment d'une importante consommation d'espace mémoire. [Dellaert et al., 2010] ont proposé une méthode appelée SPCG (subgraph preconditioned conjugate gradients). Leur approche combine entre les méthodes directes et itératives. Les auteurs identifient une sous-partie du graphe (donc du système) qui peut être résolue par une méthode directe. Le reste est résolu en utilisant la méthode du Gradient Conjugué pré-conditionné.

La matrice du système étant symétrique définie positive, les travaux récents du GraphSLAM ont, en grande majorité, opté pour les méthodes directes de factorisation (QR et Cholesky). [Dellaert and Kaess, 2006] ont proposé une formulation du GraphSLAM, à savoir \sqrt{SLAM} . Bien que la factorisation QR soit plus précise et numériquement stable, les auteurs de \sqrt{SLAM} soulignent que la méthode de Cholesky est plus rapide que la méthode QR. La méthode de Cholesky exploite le caractère épars du système. Elle ne prend en compte que les valeurs non nulles. Cela réduit énormément le temps de calcul en comparaison avec la méthode QR. La méthode de Cholesky est la méthode la plus utilisée pour résoudre les systèmes linéaires liés au GraphSLAM. On la retrouve dans la plupart des travaux portant sur le GraphSLAM ou encore l'ajustement de faisceaux [Kaess et al., 2008, 2012, Dellaert and Kaess, 2006, Kummerle et al., 2011, Polok et al., 2013a,b].

Dans cette thèse, nous avons opté pour la méthode de Cholesky pour ses grands avantages :

- Elle est plus précise par rapport aux méthodes itératives qui souffrent du problème de convergence
- Elle permet d'exploiter le caractère épars du graphe. Elle ne stocke et ne prend en compte que les valeurs non nulles. Cela permet un gain considérable en termes de temps de calcul et d'espace mémoire.

Nous allons donner, dans ce qui suit, une brève présentation de la méthode de Cholesky. Le lecteur peut trouver une description plus détaillée dans l'annexe B.

a. Présentation de la méthode de Cholesky

La méthode de Cholesky consiste en la décomposition d'une matrice carrée symétrique définie positive A en un produit d'une matrice triangulaire inférieure L et sa transposée L^T . Soit A une matrice symétrique définie positive de taille $n \times n$. Alors, A peut être factorisée sous cette forme :

$$A = L L^T \quad (1.44)$$

Le système linéaire $A x = b$ peut donc s'écrire :

$$L L^T x = b \quad (1.45)$$

Le système de l'équation (Eq. 1.45) peut donc être traité par la résolution successive de deux systèmes triangulaires simples : $L y = b$ et $L^T x = y$. Il existe une variante de cette méthode qui tire profit du caractère épars du système et minimise le nombre de calculs. La résolution du système est effectuée en quatre étapes [Erhel et al., 1993] :

- Renumérotation (ré-ordonnancement) : L'élimination d'une variable au niveau d'une ligne, lors de la factorisation, peut provoquer un remplissage qui caractérise le fait qu'un élément nul dans A devient non nul dans L . Le facteur de Cholesky L peut donc devenir moins éparse que A , voire dense. Cette étape dans la factorisation permet de minimiser le remplissage dans L . Formellement, on cherche une matrice de permutation P et on applique la factorisation sur la nouvelle matrice PAP^T . Pour ce faire, il existe plusieurs algorithmes. Les plus populaires sont l'algorithme de degré minimum et l'algorithme de Dissection emboîtée.
- Factorisation symbolique : Cette étape construit un arbre d'élimination. Celui-ci définit la structure creuse du facteur de Cholesky L .
- Factorisation numérique : En se basant sur la structure de L , on calcule les éléments non nuls du facteur de Cholesky L .
- Résolution : Dans cette étape, on résout, d'abord, le système $L y = b$ (descente). L étant triangulaire inférieure, la résolution revient à calculer les éléments de la solution y un à un en substituant à chaque fois les éléments calculés par leurs valeurs. Une fois le premier système résolu, on effectue la même opération en remontée pour calculer la solution x du système initial.

1.4.10 Calcul des incrément des amers

La résolution du système linéaire (Eq. 1.43) donne les incrément Δy_p des poses p (trajectoire). Les incrément des amers Δy_l peuvent être alors calculés dans FB7 en utilisant l'équation 1.46.

$$H_{ll}\Delta y_l = -b_l - H_{pl}^T\Delta y_p \quad (1.46)$$

1.4.11 Mise à jour de l'état du système

Ce bloc consiste à mettre à jour l'état du système en ajoutant les incrément du système résolu à la configuration actuelle. Notons que l'ajout des incrément se fait via les modèles de mouvement et d'observation.

1.5 Approches de réduction de la complexité de calcul

La complexité de calcul du GraphSLAM est due à plusieurs paramètres :

- La convergence : l'utilisation d'une méthode itérative telle que Gauss-Newton peut nécessiter un grand nombre d'itérations pour converger. Par conséquent, il est difficile de borner le temps de calcul.
- La taille du graphe : le graphe s'accroît continuellement avec le déplacement du robot.
- La structure du système linéaire : elle impacte fortement le temps de résolution du système linéaire par Cholesky. Celle-ci donne de meilleurs temps si le système est creux.

Nous présentons, dans cette section, les principales approches utilisées pour atténuer la complexité de calcul du GraphSLAM.

1.5.1 Rapidité de convergence

Le GraphSLAM nécessite une approche itérative (Gauss-Newton, Levenberg-Marquardt (LM), Dog-Leg,...) pour optimiser le graphe. La convergence est l'un des principaux problèmes dans ce type d'approches. L'objectif est d'améliorer la convergence pour un gain en temps et en précision. Le temps de calcul d'une optimisation de graphe dépend du nombre d'itérations requises par l'algorithme itératif pour converger (trouver un minimum global). [Carlone, 2013] a étudié les propriétés de convergence de l'approche Gauss-Newton. Il montre que celle-ci est influencée par : le ratio entre le contenu de l'information sur le déplacement en orientation et le déplacement en position, les distances inter-nœuds ainsi que la structure du graphe.

Pour améliorer la rapidité de convergence vers un minimum global, plusieurs techniques ont été utilisées dans la littérature. La première technique consiste à trouver une bonne configuration initiale du graphe. Celle-ci peut avoir un grand impact sur la rapidité de convergence ainsi que sur la précision de localisation et de cartographie. Elle dépend des caractéristiques des capteurs ainsi que des modèles d'observation et de mouvement utilisés. La technique intuitive consiste à utiliser les données capteurs brutes et les modèles d'observation et de mouvement pour définir une configuration initiale. Cette technique n'est pas robuste à cause du bruit inhérent sur les mesures capteurs. Pour pallier à ce problème, on peut utiliser des techniques d'approximation [Carlone et al., 2012, Carlone and Censi, 2014, Carlone et al., 2014] et de fusion de plusieurs types de capteurs [Kneip et al., 2011, Martinelli, 2012, Dong-Si and Mourikis, 2012].

D'autre part, pour faire face au problème de mauvaises initialisations, [Agarwal et al., 2013, 2014] ont proposé une méthode appelée DCS (Dynamic covariance scaling) pour l'optimisation de graphe. DCS permet de gérer dynamiquement les covariances en présence de données aberrantes sans utiliser de techniques d'initialisation particulières. [Carlone et al., 2015] ont présenté une étude comparative des techniques d'initialisation de l'orientation 3D du robot au cours de la navigation. La motivation de ces techniques vient du fait que si les orientations étaient connues, l'optimisation de graphe serait ramenée à un problème de moindres carrés linéaire. Il est, donc, très important d'avoir une estimation précise de l'orientation pour améliorer la convergence.

Le travail effectué dans cette thèse n'a pas fait l'objet de l'utilisation d'une technique d'initialisation particulière. Nous nous intéressons pas non plus à la gestion des matrices de variances-covariances re-

latives aux modèles d'observation et de mouvement. Les modèles d'implantation que nous proposons sont indépendants du nombre d'itérations de Gauss-Newton.

1.5.2 Élagage

Dans les approches de lissage telles que le GraphSLAM, le graphe a tendance à devenir plus large et plus dense au fur et à mesure que le robot acquiert de nouvelles mesures capteurs. Pour permettre un processus de SLAM continue ou non limité en temps (longlife SLAM), il est impératif d'élaguer le graphe à cause des ressources de mémoire et de calcul qui finissent par devenir insuffisantes. Ce scénario est très courant pour les véhicules qui parcourront de grandes distances. L'élagage se fait à deux niveaux : la marginalisation et l'éparsification. La marginalisation consiste à réduire la taille du graphe en omettant un certain nombre de nœuds. L'éparsification cherche à éliminer des arêtes pour rendre le graphe encore plus épars. [Dissanayake et al., 2002] ont montré qu'il est possible de supprimer un grand pourcentage d'amers sans compromettre la consistance de la carte.

Pour élaguer le graphe, il existe plusieurs approches. Dans [Konolige and Agrawal, 2008], un nouveau nœud n'est ajouté au graphe que s'il n'est pas proche d'un autre nœud déjà existant. [Konolige and Bowman, 2009] ne mettent à jour la carte que si l'environnement change. Les auteurs s'appuient également sur la redondance des images au sein d'un voisinage pour réduire leur nombre. [Walcott-Bryant et al., 2012] exploitent le changement d'environnement pour supprimer les nœuds qui ne sont plus actifs. [Eade et al., 2010] ont présenté un système de SLAM monoculaire où la complexité est réduite en marginalisant des nœuds et leurs arêtes correspondantes. Quand le degré d'un nœud dépasse un certain seuil, les arêtes présentant de petites erreurs sont supprimées. En d'autres termes, les auteurs suppriment les arêtes qui collent le mieux avec l'état courant du système.

D'autres approches utilisent la théorie de l'information pour minimiser la perte d'information lors de l'élagage [Kretzschmar et al., 2010, Ila et al., 2010, Kretzschmar et al., 2011, Kretzschmar and Stachniss, 2012, Wang et al., 2013]. [Choudhary et al., 2015] ont développé un algorithme basé sur la théorie de l'information pour réduire le nombre de nœuds. Les auteurs proposent également une version incrémentale de cet algorithme pour le SLAM incrémental. Leur méthode utilise une fonction objectif basée sur la consommation mémoire et l'élimination des amers les moins informatifs.

Par ailleurs, [Carlevaris-Bianco et al., 2014] ont introduit la technique des contraintes linéaires génératrices pour approximer la contraintes générées par la marginalisation des nœuds. Cette technique a été améliorée dans les travaux de [Mazuran et al., 2014, 2016]. Les auteurs remplacent les contraintes par un ensemble de mesures virtuelles non linéaires. Ils utilisent, ensuite, une méthode d'optimisation numérique pour les mesures du bruit associées à chaque arête virtuelle.

Finalement, d'autres travaux de recherche ont porté sur le couplage/découplage de la marginalisation et l'éparsification [Paull et al., 2016].

1.5.3 Sous-cartographie et fenêtre glissante

Une autre approche pour réduire la complexité de calcul consiste à appliquer la stratégie de “diviser pour régner”. Il s’agit, concrètement, de diviser le graphe en petits sous-graphes ou sous-cartes. Chaque sous-graphe est, ensuite, optimisé indépendamment des autres. Les sous-graphes sont enfin joints pour reconstituer le graphe initial [Estrada et al., 2005, Clemente et al., 2007, Snavely et al., 2008, Pinies and Tardos, 2008, Ni and Dellaert, 2010, Zhao et al., 2011, Grisetti et al., 2012]. [Ni et al., 2007] ont proposé une approche de sous-cartographie basée sur la méthode de dissection emboîtée. Chaque sous-graphe a un nœud servant d’origine (base node). Le graphe global est obtenu en effectuant des transformations rigides des sous-graphes les uns par rapport aux autres. [Zhao et al., 2013] ont proposé une méthode dans laquelle les sous-cartes sont transformées dans le même repère avant l’étape de jointure. Celle-ci est réalisée via la résolution d’un problème de moindres carrés linéaire. Ce dernier travail a été étendu dans [Zhao et al., 2014]. Les sous-cartes construites en utilisant trois images monoculaires ont différentes échelles. Les auteurs proposent, donc, de nouvelles stratégies pour transformer les coordonnées et les échelles des sous-cartes. [Cheng et al., 2015] ont amélioré le travail de [Zhao et al., 2014] en traitant les fausses fermetures de boucles. Les auteurs proposent une optimisation retardée pour les sous-cartes contenant des fermetures de boucles aberrantes. Le complément de Schur est utilisé pour éliminer les blocs corrompus de la matrice d’information.

Par ailleurs, d’autres approches utilisent des fenêtres glissantes [Sibley et al., 2010, Huang et al., 2011] pour optimiser le graphe en un temps constant lors de la navigation. Le principe consiste à n’utiliser que les mesures capteurs relatives à la fenêtre courante. D’autre part, pour diminuer le nombre d’images à traiter et la taille du graphe résultant, on peut utiliser les approches basées images-clés (keyframe-based approaches) [Klein and Murray, 2007, Konolige and Agrawal, 2008, Konolige et al., 2010a, Strasdat et al., 2011]. Dans ces approches, un sous-ensemble d’images est heuristiquement sélectionné pour résoudre le problème de localisation ou de reconstruction d’environnement.

1.5.4 Incrémentalité

Pour optimiser le graphe, on distingue deux modes : le mode batch et le mode incrémental. Le premier mode revient à résoudre le problème de SLAM hors-ligne. En d’autres termes, on attend la construction de tout le graphe avant d’entamer une optimisation de graphe. Les méthodes itératives peuvent ne pas converger en cas de mauvaise initialisation de l’état du système par les mesures capteurs. Dans le second mode, le graphe est incrémentalement optimisé. La partie Front-end reste opérationnelle et met continuellement à jour le graphe. Pour réduire la complexité de calcul, l’optimisation de graphes est effectuée à des instants précis (toutes les n étapes ou à la détection d’une fermeture de boucle). On peut également optimiser le graphe à chaque nouvelle mesure. Le fait de corriger continuellement l’état du système permet d’accélérer la convergence.

Les Frameworks iSAM 1.0 [Kaess et al., 2008] et iSAM 2.0 [Kaess et al., 2012] exploitent le caractère incrémental du GraphSLAM. La particularité de iSAM réside dans la construction incrémentale du

système linéaire. L'idée vient du fait que l'insertion d'une contrainte (arête) dans le graphe n'affecte qu'un voisinage limité du graphe. Par conséquent, à l'insertion d'une nouvelle contrainte, iSAM modifie uniquement les blocs affectés par la contrainte. L'optimisation de graphe complète (la résolution du système) est périodiquement effectuée. Cela permet de réduire considérablement le temps de calcul. Cependant, iSAM suppose qu'entre deux optimisations de graphe, l'état du système satisfait les contraintes du graphe. Dans la pratique, cela n'est pas tout le temps vérifiable, particulièrement dans les environnements fortement bruités. De plus, la marginalisation d'un nœud ou d'une arête dans le graphe rend le système linéaire biaisé et fausse le processus de construction incrémentale.

Par ailleurs [Rosen et al., 2012, 2014] ont proposé une méthode appelée RISE (Robust Incremental least-Squares Estimation). Celle-ci utilise une variante incrémentale de l'algorithme de Dog-Leg pour améliorer la convergence.

1.5.5 Méthodes de résolution des NLS

Dans cet axe, on trouve tous les travaux concernant les méthodes de résolution des problèmes de moindres carrées et des systèmes linéaires associés (discutés auparavant dans respectivement section 1.4.3 et section 1.4.9). La résolution du système linéaire peut être très contraignante particulièrement si le graphe est dense. Pour réduire la taille du système linéaire, on peut appliquer le complément de Schur [Thrun and Montemerlo, 2006, Dellaert and Kaess, 2006, Konolige and Agrawal, 2008, 2007, Sibley et al., 2007, Kummerle et al., 2011]. Cependant, celui-ci peut dégrader les performances temporelles si le système résultant est dense.

Par ailleurs, le ré-ordonnancement des variables (renumérotation) dans Cholesky est d'une grande importance dans la résolution. On distingue deux niveaux de ré-ordonnancements : le ré-ordonnancement des nœuds et le ré-ordonnancement des colonnes de la matrice élémentaire. Dans la première, les nœuds sont, à chaque fois, réordonnancés de telle sorte à mettre les poses robot avant les amers. Cela donne un système (Eq. 1.42) souvent épars et facile à gérer en termes de calcul et de stockage mémoire. Le deuxième niveau de ré-ordonnancement est appliqué lors de la phase de ré-ordonnancement de la méthode de Cholesky. [Agarwal and Olson, 2012] ont étudié les différentes stratégies de ré-ordonnancement dans le contexte des algorithmes de SLAM. Les auteurs donnent des recommandations basées sur leurs résultats d'évaluation.

D'autre part, [Polok et al., 2013a] ont proposé une factorisation incrémentale de Cholesky pour les problèmes de moindres carrés non linéaires. Les auteurs ont évalué leur factorisation sur un algorithme de SLAM basé optimisation de graphe.

1.5.6 Optimisation logicielle et matérielle

Les ressources de calcul et de stockage sur les architectures embarquées rendent les techniques précédentes insuffisantes. Pour accélérer encore les traitements du GraphSLAM, il est intéressant de

repenser les implantations de celui-ci en fonction des architectures de calcul. Les optimisations logicielles visent, en général, à exploiter le caractère épars des graphes issus du GraphSLAM ou de l'ajustement de faisceaux. Cela implique particulièrement l'organisation et la manipulation des données en mémoire. Dans cette optique, le Framework g^2o [Kummerle et al., 2011] agit sur la granularité des données. Celles-ci sont divisées en petits blocs dont les tailles varient en fonction des éléments de graphe correspondants (pose, amer, arête d'observation,). Les opérations arithmétiques sont effectuées sur des blocs au lieu des valeurs élémentaires. Cela permet d'optimiser considérablement les opérations de chargement et de rangement mémoire. Pour optimiser encore, g^2o exploite l'unité SIMD (Single Instruction Multiple Data) des processeurs. D'autre part, les librairie iSAM 1.0 [Kaess et al., 2008] et iSAM 2.0 [Kaess et al., 2012] exploitent l'incrémentalité tel que expliqué précédemment. De plus, de la même manière que g^2o , les données sont divisées et manipulées en petits blocs.

Par ailleurs, l'une des principales techniques pour réduire la complexité de calcul est d'exploiter les architectures parallèles. g^2o parallélise la construction du système linéaire sur une architecture homogène (CPU). La résolution du système linéaire n'est pas parallélisée. Elle est réalisée via des librairies externes (CSparse et CHOLMOD). L'exploitation des processeurs graphiques et des architectures reconfigurables permet d'accélérer encore les traitements. Le lecteur pourra trouver dans le chapitre 2 une description des architectures parallèles (GPU et FPGA). Nous discuterons également des travaux de recherche ayant utilisé ce type d'architectures pour accélérer le GraphSLAM ou l'ajustement de faisceaux.

1.5.7 Notre approche

Pour réduire encore la complexité de calcul, notre approche s'appuie sur deux axes complémentaires : la représentation mémoire des données et l'implantation sur architectures hétérogènes embarquées. Dans le premier, très peu de travaux ont investigué l'impact de l'organisation des données. Nous investiguons à travers ce travail l'impact des structures de données sur les performances temporelles. Nous proposons une structure de données adéquate pour la représentation du graphe en mémoire et la résolution du problème de moindres carrés associé. La représentation mémoire est couplée à une construction de graphe incrémentale. La structure de données est ainsi mise à jour incrémentalement au fur et à mesure que le robot explore l'environnement.

Pour le second axe, nous explorons l'utilisation d'architectures hétérogènes embarquées pour accélérer le GraphSLAM. Les performances accrues de ce type d'architectures peuvent être exploitées pour concevoir des systèmes embarqués temps-réel orientés applications de vision ou de robotique. Parmi ces architectures, on trouve les SoCs à base de FPGAs et des coeurs ARM (Cyclone V, Zynq-7000), ainsi que les SoCs à base de GPU et des coeurs ARM (Tegra K1). Nous proposons, donc, une méthodologie d'implantation du GraphSLAM issue de l'expertise acquise sur des architectures hétérogènes embarquées.

1.6 Conclusion

Nous avons parlé dans ce chapitre des différentes méthodes de SLAM. Celles-ci peuvent être classifiées en approches probabilistes et ensemblistes. Le premier type s'appuie sur l'inférence Bayésienne pour corriger la position du robot et la carte des amers. L'approche ensembliste, quant à elle, utilise l'analyse par intervalle dans un objectif de garantir ou borner les résultats de localisation et de cartographie. Les méthodes probabilistes EKF-SLAM et FastSLAM ont été les premières à être utilisées pour résoudre le problème de SLAM. Elles souffrent, cependant, du problème de consistance. Au fil du temps, les résultats deviennent inconsistants et ne reflètent pas la réalité. Le GraphSLAM est une approche de lissage. Celle-ci raffine, à chaque correction, toute la trajectoire. Le GraphSLAM permet de faire face au problème de consistance présent dans l'EKF-SLAM et le FastSLAM.

De par sa consistance ainsi que sa faible consommation en mémoire en comparaison avec l'EKF-SLAM et le FastSLAM, le GraphSLAM s'impose de plus en plus, au sein de la communauté scientifique, comme un premier choix pour le SLAM [Mur-Artal et al., 2015]. Il peut être utilisé pour localiser le robot et reconstruire de larges environnements sur de longues périodes de temps (longlife SLAM).

Malgré ses avantages indéniables en comparaison à l'EKF-SLAM et le FastSLAM, il est important de réduire encore la complexité de calcul du GraphSLAM. Dans cette optique, plusieurs approches peuvent être utilisées. On peut agir sur la rapidité de convergence des méthodes itératives (Gauss-Newton) par le biais d'une bonne initialisation du graphe. On peut, par ailleurs, atténuer la croissance rapide du graphe en élaguant des parties de celui-ci. Il est aussi possible de diviser le graphe en sous-graphes où chacun est indépendamment optimisé des autres. Le principe d'incrémentalité a également toute son importance pour le SLAM à large-échelle.

Cependant, ces approches de réduction de complexité de calcul restent toujours contrariées par les capacités de calcul et de mémoire des calculateurs. Les ressources de ceux-ci finissent par devenir insuffisantes avec la croissance du graphe, particulièrement sur les architectures embarquées. Notre travail s'inscrit dans la catégorie optimisation logicielle et matérielle. Celle-ci a été très peu explorée. Il s'agit, dans ce travail, d'explorer l'apport de cette approche dans l'accélération des algorithmes de SLAM basé graphe. Notre approche vise à prendre avantage du caractère épars et incrémental du GraphSLAM. Nous nous intéressons, en particulier, à deux aspects : l'organisation des données en mémoire et la répartition adéquate des tâches sur des architectures parallèles hétérogènes.

Chapitre 2

Architectures et outils de synthèse de haut niveau

Sommaire

2.1	Introduction	42
2.2	Processeurs graphiques	43
2.2.1	Architecture du Tegra K1	44
2.2.2	Programmation des processeurs graphiques	46
2.2.3	Processeurs graphiques pour l'accélération du SLAM	47
2.3	Architectures reconfigurables	48
2.3.1	Vers des FPGAs intégrant des processeurs hardcores	49
2.3.2	Architecture système-sur-puce du Cyclone V	50
2.3.3	FPGA pour le SLAM	51
2.4	Flux de conception sur FPGA : Outils de synthèse de haut niveau	52
2.4.1	Description manuelle	52
2.4.2	Synthèse de haut niveau	53
2.4.3	Présentation de LegUp	55
2.4.4	Approche OpenCL	58
2.5	Conclusion	62

2.1 Introduction

La conception des systèmes embarqués s'adapte aux besoins du marché en tirant profit des dernières technologies matérielles et d'implantations logicielles. L'objectif est d'améliorer les performances globales et la fiabilité du système tout en minimisant le coût de conception et de production. Les performances s'articulent autour de trois axes principaux, à savoir les performances temporelles, la consommation d'énergie et la sûreté de fonctionnement. Limitée par les latences d'accès à la mémoire, l'augmentation de la fréquence des processeurs monocœurs ne suffit plus. Cela a donné lieu aux architectures multicœurs et reconfigurables. Celles-ci sont capables d'exécuter un grand nombre de tâches en parallèle en vue d'accélérer des flots de traitements. A la quête de meilleures performances, les calculateurs ont beaucoup évolué, depuis les premiers processeurs CPUs jusqu'aux architectures massivement parallèles telles que les GPUs et les FPGAs. On peut classifier ces calculateurs en trois catégories :

- Circuits dédiés : Un circuit dédié ou ASIC (Application-Specific Integrated Circuit) est un circuit personnalisé pour exécuter une tâche spécifique de manière câblée.
- Processeurs programmables : Ce sont des processeurs pouvant être reprogrammés pour exécuter une variété de tâches. Cette catégorie comprend essentiellement les processeurs généralistes CPU (Central Processing Unit), les processeurs graphiques GPU (Graphics Processing Unit) et les processeurs de traitement de signal DSP (Digital Signal Processor).
- Circuits reconfigurables : Ce sont des circuits pouvant concilier les deux précédents. Ils sont composés d'éléments logiques pouvant être librement reconfigurés pour implanter toute fonction logique. En termes de flexibilité de programmation, ce type de circuits se situe entre les circuits dédiés et les processeurs programmables (Fig. 2.1). Dans cette catégorie, on trouve essentiellement les FPGAs (Field-Programmable Gate Array) et les CPLDs (Complex Programmable Logic device).

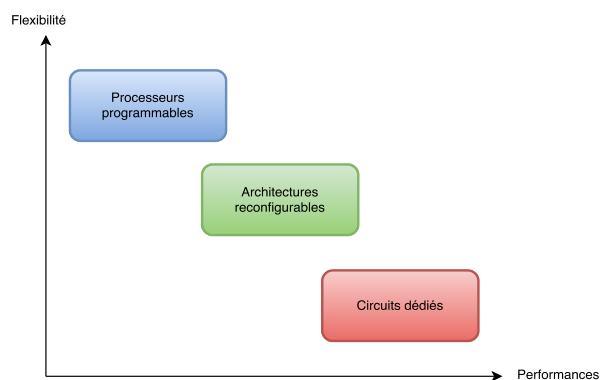


FIGURE 2.1 – Flexibilité vs performances.

En fonction des types de calculateurs, on distingue deux types de systèmes de calcul : systèmes homogènes et hétérogènes. Un système homogène comprend un seul type de calculateurs. Un système hétérogène combine, par contre, plusieurs calculateurs de différents types (CPU, GPU, DSP, FPGA,...). Un système hétérogène peut également disposer de circuits coprocesseurs matériels interfacés au processeur principal. Cela est le cas dans la conception d'appareils téléphoniques qui intègrent des unités de décodage vidéo sur la même puce que le processeur principal.

Par ailleurs, un système hétérogène est dit discret si celui-ci relie des calculateurs physiquement séparés. La communication est généralement assurée par une liaison PCI express (PCIe) tel que les ordinateurs classiques qui intègrent un coprocesseur graphique à coté du processeur généraliste. Ce type de liaisons peut être très contraignant, pour concevoir des systèmes embarqués temps-réel, à cause des latences de communications qui peuvent être élevées et du protocole d'arbitrage complexe. Partant de ce constat, de nouvelles architectures hétérogènes embarqués sont apparues dans lesquelles les calculateurs sont intégrés sur la même puce que le processeur principal donnant lieu à un système sur puce ou System-on-Chip (SoC). Ces calculateurs sont connectés entre eux via des bus spécifiques, et partagent un certain nombre de ressources telles que les mémoires. Le temps de communication entre les calculateurs (transfert de données) est grandement amélioré en comparaison avec les systèmes discrets. On retrouve ce type d'architectures dans les APU AMD, la série Tegra de Nividia (Tegra 2, K1, X1), Broadwell de Intel, Exynos 5422 de Samsung.

Chaque type de calculateurs présente des avantages et des inconvénients. L'utilisation d'un calculateur donné (CPU, DSP, GPU, FPGA,...) peut s'avérer plus convenable pour une tâche donnée que pour une autre. L'approche Adéquation Algorithme-Architecture (AAA) [Sorel, 1994, Boulos, 2012, Brillu et al., 2013] consiste à étudier simultanément les aspects algorithmiques et architecturaux en prenant en compte leurs interactions. L'adéquation algorithme-architecture permet de répartir les tâches d'un algorithme sur les différents calculateurs disponibles dans le but de minimiser, autant que possible, une fonction « objectif ». Celle-ci est susceptible d'être formalisée et fait intervenir, en général, le temps de calcul et la consommation d'énergie tout en prenant en considération le coût des différents calculateurs. Nous avons suivi cette approche pour explorer des implantations du GraphSLAM.

Dans cette thèse, nous nous intéressons particulièrement aux processeurs programmables du type CPU et GPU ainsi qu'aux architectures reconfigurables du type FPGA. Nous allons montrer, à travers ce chapitre, les caractéristiques et le flux de conception de chaque architecture. Nous mettrons en avant les architectures et les outils connexes utilisés dans ce travail de thèse.

2.2 Processeurs graphiques

Le terme GPU (Graphic Processing Unit) est apparu en 1999 lorsque Nvidia a créé le premier GPU au monde (GeForce256). Un GPU est constitué de plusieurs unités de calcul parallèles que l'on appelle multiprocesseurs de flux ou SM (Streaming Multiprocessor). Un SM est, à son tour, composé de plusieurs unités élémentaires appelées SP (Streaming Processor). Un SM est conçu pour exécuter de manière concurrente des milliers de threads.

La figure 2.2 présente l'architecture d'un GPU Fermi de Nvidia. Celle-ci comprend 16 multiprocesseurs de flux (SM). Chaque SM contient principalement 32 processeurs de flux (SP), 16 unités Load/Store, 64Ko de mémoire partagée et 32Ko de registres (32 bits). Les multiprocesseurs de flux sont entourés de 6 interfaces de mémoire DRAM, un cache L2, un Scheduler (ordonnanceur) Giga-Thread et une interface hôte.

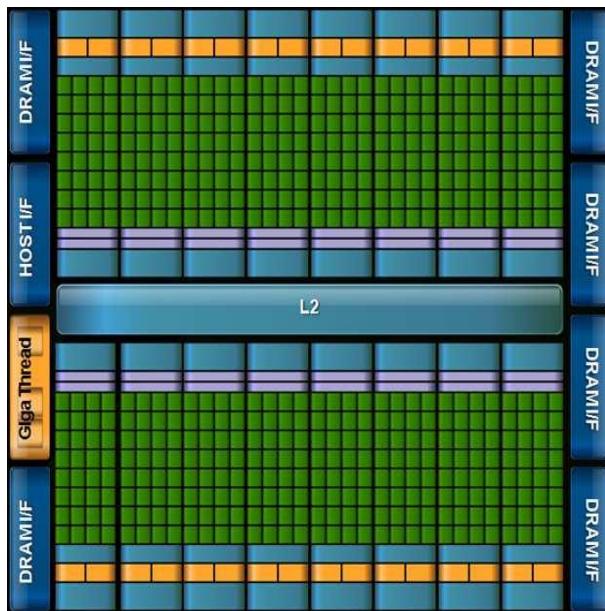


FIGURE 2.2 – Architecture Fermi de Nvidia (16 SM).

Nvidia et AMD sont les principaux vendeurs de processeurs graphiques. En termes d’architecture, la différence clé entre Nvidia et AMD réside dans le fait que AMD utilise des processeurs VLIW (Very Long Instruction Word) pour le calcul parallèle. Les performances de l’application dépendent, donc, du nombre d’instructions parallèles VLIW que le compilateur peut créer à partir du code source. Si celui-ci contient beaucoup de dépendances de données, l’approche VLIW ne sera pas performante. En effet, il est parfois difficile de trouver des instructions indépendantes pour les regrouper dans une seule instruction VLIW. D’autre part, Nvidia s’appuie sur une exécution SIMD (Single Instruction Multiple Data). Cette approche exploite le parallélisme de données. A l’exécution, le Scheduler/Dispatcher organise les threads dans des blocs de taille fixe appelés wraps. La taille d’un wrap dépend de l’architecture du GPU. Dans les architectures actuelles de Nvidia¹, la taille d’un wrap est de 32 [Lindholm et al., 2008]. Les threads d’un wrap sont lancés en parallèle et exécutent la même instruction, mais chacun sur une donnée différente.

L’approche SIMD est adaptée aux programmes contenant peu de divergences dans les chemins d’exécution (ou chemins de code). Une divergence entre threads peut survenir suite à l’évaluation d’une condition (branchement). Cela casse le parallélisme et dégrade les performances.

2.2.1 Architecture du Tegra K1

Le Tegra est un système-sur-puce hétérogène incluant un processeur ARM et un GPU Nvidia. Il est principalement destiné à la téléphonie mobile. Produit par Nvidia, depuis la première architecture Tegra lancée en 2008, le Tegra a connu une évolution très rapide. Nvidia a ensuite lancé le Tegra 2 (2009), le Tegra 3 (2011), le Tegra 4 (2013), le Tegra 4i (2013), le Tegra K1 (2014) et enfin le Tegra X1 (2015).

1. Pour les GPUs AMD, un wrap peut contenir jusqu’à 64 threads.

Pour une implantation hétérogène du GraphSLAM, nous avons utilisé l'architecture Tegra K1. Le nouveau Tegra X1 présente une version évoluée, en termes de performances, en comparaison avec le Tegra K1. Ce dernier existe en deux versions 32 bits et 64 bits. La version utilisée dans les travaux de cette thèse est 32 bits [Nvidia, 2013]. La figure 2.3 illustre l'architecture interne du Tegra K1. Il est composé essentiellement d'un processeur ARM 4-PLUS-1 Cortex-A15 « r3 » et un GPU Kepler. Il intègre également des unités de décodage vidéo/audio, un dual core ISP (Image Signal Processor) interfaçable avec une caméra (jusqu'à 100 Mégapixels), ainsi qu'une unité d'affichage 4K.

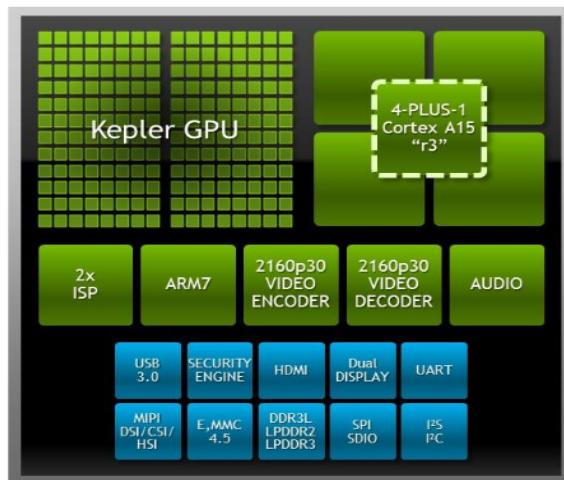


FIGURE 2.3 – Architecture du Tegra K1 [Nvidia, 2013].

Le GPU Kepler utilisé dans le Tegra K1 a été optimisé pour fournir de meilleures performances avec une basse consommation d'énergie. La figure 2.4 donne l'architecture interne du GPU Kepler. Le Kepler du Tegra K1 dispose d'un seul multiprocesseur de flux (SM) à 192 processeurs de flux (Streaming Processor). Notons que les GPUs Kepler destinés aux machines classiques peuvent avoir jusqu'à 2880 coeurs (8 SMs pour le GPU GeForce GTX 680). Par conséquent, ils consomment quelques centaines de watts en énergie. Le Tegra K1 consomme moins de 4 watts. Cela fait du Tegra K1 une architecture convenable pour les systèmes embarqués.

Table 2.1 – Specifications du Tegra K1.

CPU	ARM 4-PLUS-1 Cortex-A15 « r3 »
Fréquence du CPU	2.3 GHz
Famille du GPU	Kepler
Nombre de multiprocesseurs de flux (SM)	1
Nombre de coeurs du SM	192
Fréquence du GPU	0.85 GHz
Mémoire globale	1746 Mo
Mémoire partagée entre CPU et GPU	Oui

Le tableau 2.1 résume les spécifications les plus importantes du Tegra K1. Les caractéristiques du Tegra K1 font de lui une architecture puissante et intéressante pour les applications mobiles dont les

besoins en ressources grandissent continuellement. Il est donc intéressant d'explorer une implantation du GraphSLAM sur cette architecture.

2.2.2 Programmation des processeurs graphiques

Historiquement, les GPUs étaient programmés à l'aide de langages considérés de bas niveau tels que OpenGL. L'exploitation d'un GPU était donc fastidieuse et nécessitait une expertise sur l'architecture du GPU et le langage de développement pour avoir de bonnes performances. En 2007, Nvidia a introduit le premier modèle de programmation pour GPU, à savoir CUDA (Compute Unified Device Architecture). Celui-ci est destiné aux GPUs de Nvidia. Une année plus tard, Apple conçoit un autre langage appelé OpenCL (Open Computing Language). Dans le cadre d'une collaboration avec AMD, Intel et Nvidia, OpenCL devient ensuite un standard pour la programmation des systèmes hétérogènes. Il existe d'autres langages et APIs (Application Programming Interface) tels que OpenACC qui permet également de programmer des GPUs.

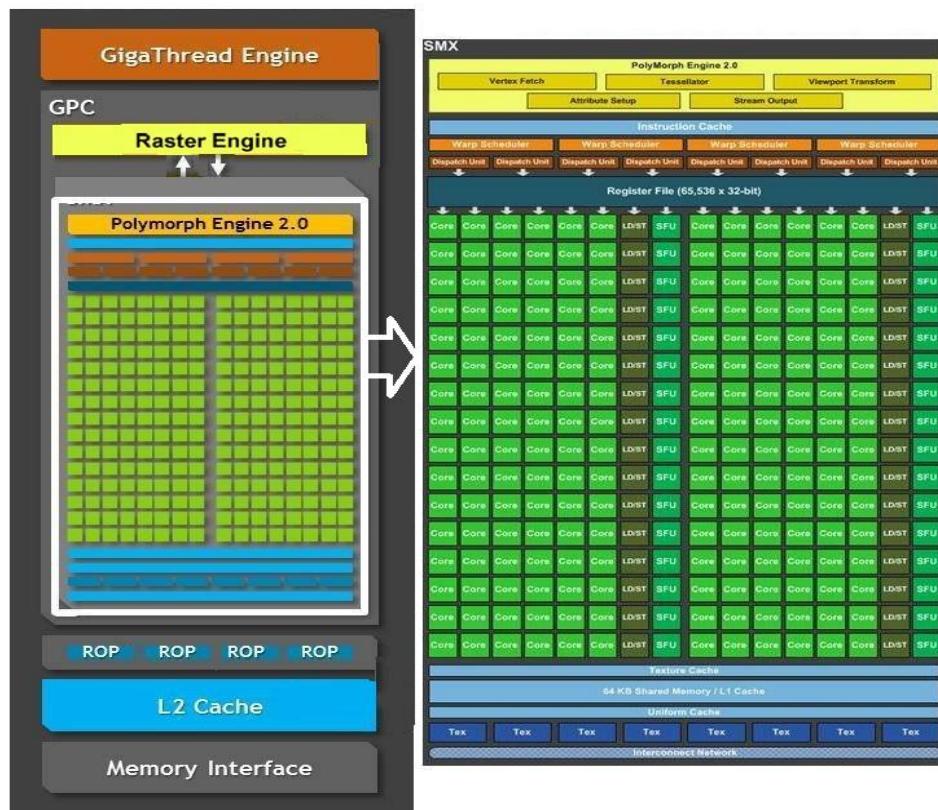


FIGURE 2.4 – Architecture Kepler du Tegra K1 [Nvidia, 2013].

OpenCL et CUDA restent les plus évolués et utilisés pour implanter des applications sur architectures hétérogènes. Bien que OpenCL puisse être utilisé pour programmer des GPUs Nvidia, les développeurs préfèrent CUDA. Celui-ci offre plus de flexibilité, en termes de développement, pour les architectures Nvidia. Pour notre étude sur le Tegra K1, nous avons utilisé CUDA. Notons que le Tegra K1

ne supporte pas encore OpenCL. Le modèle de programmation de OpenCL est flexible et supporte, grâce à sa standardisation au sein du groupe Chronos, plusieurs types de systèmes hétérogènes constitués de CPUs, GPUs, DSPs et récemment de FPGAs. Pour les GPUs, OpenCL est notamment utilisé par la communauté AMD. Par ailleurs, durant les dernières années, il y a eu une évolution rapide dans l'utilisation de OpenCL pour programmer des systèmes hétérogènes à base de FPGA. Altera a rejoint le groupe Chronos et a créé un outil de synthèse de haut niveau basé sur OpenCL. Nous donnerons plus de détails sur cette nouvelle technologie plus loin dans ce chapitre.

Les modèles de programmation de CUDA et OpenCL considèrent deux parties. Une partie hôte qui exécute le programme principal sur un processeur généraliste. La seconde partie est relative au coprocesseur (GPUs pour CUDA et GPUs, DSPs, FPGA pour OpenCL). Le processeur principal lance les calculs sur le coprocesseur et prend en charge les transferts de données.

Les tâches à exécuter sur le GPU sont organisées en kernels. Un kernel est, globalement, une séquence d'instructions destinée à traiter un élément élémentaire (par exemple le traitement relatif à un élément d'un vecteur). Pour de meilleures performances, le concepteur devrait réécrire l'algorithme de telle manière à maximiser le parallélisme de données. Un kernel est exécuté par plusieurs threads lancés en parallèle. Ces threads sont organisés en blocs (thread blocks). Les blocs de threads sont, à leur tour, organisés en grille. L'ordonnanceur (Wrap Scheduler) du multiprocesseur de flux (SM) divise les threads en wraps puis les affecte aux différents processeurs de flux (SP). Les threads d'un même bloc (CUDA Thread Block) peuvent coopérer et partager des données stockées dans la mémoire partagée du SM correspondant.

2.2.3 Processeurs graphiques pour l'accélération du SLAM

Les GPUs ont été largement utilisés dans des applications de robotique, particulièrement dans la vision par ordinateur. La communauté scientifique a également exploité les GPUs pour accélérer des algorithmes de reconstruction d'environnement ou de SLAM d'une manière générale. Les solutions proposées sont souvent hétérogènes où le CPU et le GPU coopèrent ensemble pour exécuter les tâches de l'algorithme à accélérer. [Michel et al., 2007] se sont servis d'un GPU pour accélérer le suivi d'objets 3D à l'aide de caméras en vue d'atteindre les performances temps réel lors du contrôle d'un robot humanoïde. [Zhang and Martin, 2013] ont proposé une méthode pour accélérer le filtre particulaire (FastSLAM) sur un GPU de Nvidia. Les auteurs ont déporté le calcul des poids des particules sur le GPU.

Quant aux algorithmes basés optimisation de graphe, des travaux ont porté sur l'accélération de certaines tâches de l'algorithme sur GPU. Ces travaux portent essentiellement sur la méthode d'ajustement de faisceaux ou BA (Bundle Adjustment) [Triggs et al., 1999, Lourakis and Argyros, 2005, Konolige and Agrawal, 2008] dans le domaine de la vision. L'ajustement de faisceaux consiste à minimiser l'erreur entre les observations réelles et les mesures prédictes (reprojection) de n amers observés par un ou plusieurs capteurs. La résolution de ce problème conduit à un problème d'optimisation de graphe. Celui-ci est globalement similaire à celui du GraphSLAM. L'ajustement de faisceaux, se

caractérise souvent par un très grand nombre d'itérations en vue de reconstruire principalement la carte de l'environnement exploré. Pour accélérer cette reconstruction, [Choudhary et al., 2010] ont proposé une approche hétérogène pour répartir les calculs sur le CPU et le GPU. L'algorithme utilisé s'appuie sur une variante épars de la méthode LM [Lourakis and Argyros, 2009]. Le Hessien (matrice d'information) et le complément de Schur sont construits sur le GPU. Dans leur algorithme, la résolution du système linéaire passe par une inversion matricielle. Celle-ci est réalisée via une factorisation Cholesky effectuée en utilisant la librairie MAGMA [Ltaief et al., 2010]. [Wu et al., 2011] ont présenté un partitionnement CPU/GPU pour l'ajustement de faisceaux. Les auteurs ont utilisé une approche itérative pour résoudre le problème de moindre carrés, à savoir le PCG (Preconditioned Conjugate Gradients). [Rodriguez-Losada et al., 2013] ont parallélisé un algorithme de construction de grilles d'occupation sur GPU. La résolution du système est assurée par une librairie externe. [Ratter et al., 2013] ont présenté un algorithme de GraphSLAM couplé à une grille d'occupation. Les auteurs raffinent la carte de l'environnement en utilisant un GPU.

En termes d'architecture de calcul, les travaux précédents se servent de calculateurs classiques (Intel I7, Tesla C1060, Nvidia 570GTX, etc) dans leurs systèmes d'évaluation. Le travail de cette thèse s'intéresse aux architectures embarquées dont les ressources de calcul et de mémoire sont souvent très faibles par rapport aux calculateurs utilisés dans les travaux précédents. Par ailleurs, durant la dernière décennie, les performances des architectures embarquées à base de GPU, destinées à la téléphonie mobile, ont connu une croissance très rapide. Cela promeut leur utilisation dans les systèmes de vision par ordinateur.

Récemment, des travaux de recherche ont essentiellement traité de l'optimisation et de l'évaluation de performances des applications de vision sur des architectures mobiles. [Nardi et al., 2015] ont proposé le SLAMBench. Celui-ci est un Framework qui permet de valider et d'évaluer les nouvelles implantations de l'algorithme de KinectFusion (KF) [Newcombe et al., 2011]. Le KF permet de reconstruire des scènes 3D moyennant une caméra à profondeur telle que la Kinect de Microsoft. Le SLAM-Bench vise à investiguer des compromis en performances temporelles, précision et en consommation d'énergie. Parmi les architectures utilisées par les auteurs, on trouve l'ODROID (XU3), Arndale et le Tegra K1. Les auteurs soulignent que le TK1 a atteint les performances temps-réel avec 22 images/s. [Zia et al., 2016] ont étendu le SLAMBench en rajoutant un algorithme de LSD-SLAM (Large-Scale Direct Monocular SLAM) [Engel et al., 2014]. Dans le même contexte, [Backes et al., 2015] ont présenté plusieurs optimisations concernant l'implantation du KF sur les architectures embarquées. Les évaluations ont été effectuées sur l'ODROID (XU3) et Arndale. En termes d'optimisation, nous présentons dans ce document des optimisations de parallélisation indépendantes de l'architecture utilisée tout en exploitant le caractère épars et incrémental du GraphSLAM [Dine et al., 2015b, 2016].

2.3 Architectures reconfigurables

A l'inverse des circuits dédiés qui sont figés et destinés à exécuter une tâche bien spécifique, les architectures reconfigurables offrent une grande flexibilité dans la conception des systèmes à base

de circuits logiques. Dans cette catégorie, on trouve essentiellement les PALs (Programmable Array Logic), les CPLDs (Complex Programmable Logic Device) et les FPGAs (Field Programmable Gate Array). Les FPGAs constituent la technologie la plus évoluée et utilisée de ce type d'architectures. Les principaux fabricants de FPGAs, aujourd'hui, sont Xilinx, Altera et Actel.

La figure 2.5 illustre la structure interne d'un FPGA. Cette dernière est une grille 2D de cellules logiques reconfigurables ou CLB (Configurable Logic Blocks). Une CLB est généralement constituée d'un circuit de mémorisation contrôlé par un signal d'horloge, d'une LUT (Look-Up Table) et d'éléments logiques simples. Les LUTs sont de type SRAM (Static Random Access Memory) et donc volatiles. Elles sont utilisées pour implanter des fonctions logiques grâce à leurs tables de vérité. Les cellules logiques peuvent être interconnectées via un réseau de routage programmable par le concepteur. Un réseau de routage est constitué de fils verticaux et horizontaux. L'interconnexion est assurée grâce à des blocs d'interconnexion (Connection Blocks) et de routage (Switch Blocks). La grille de CLBs est entourée de blocs d'entrées/sorties ou IOB (Input-Output Block). Ces derniers sont reliés au réseau de routage ainsi qu'aux broches physiques afin de permettre la communication entre le FPGA et les composants extérieurs du système. Les CLBs peuvent également être utilisées selon le besoin pour réaliser des blocs de mémoire souvent de petite taille pour rapprocher les données aux accélérateurs. De plus, un FPGA dispose souvent de blocs spécifiques de type mémoires dédiés (RAM blocks) ou bien multiplicateurs câblés. Ceux-ci sont également connectés au réseau de routage.

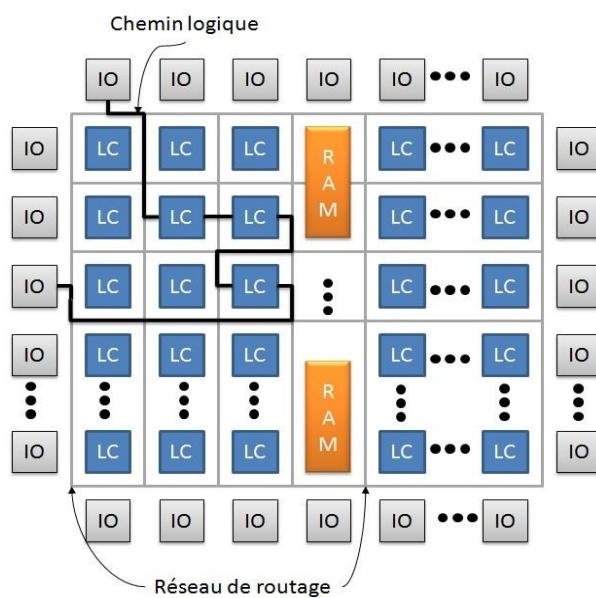


FIGURE 2.5 – Structure interne d'un FPGA [Njiki, 2013].

2.3.1 Vers des FPGAs intégrant des processeurs hardcores

La densité d'intégration d'éléments logiques et de blocs de RAMs dans un FPGA ne cesse de croître. Les FPGAs modernes comptent plus d'un milliard de portes logiques. Par ailleurs, bien que les fréquences d'horloges aient été améliorées, les FPGAs opèrent à des fréquences très basses par rapport

aux processeurs programmables. D'autre part, la fréquence de fonctionnement est toujours en dessous de la fréquence constructeur. Le facteur limitant est les communications vers l'extérieur et vers l'accélérateur qui se limitent, en général, à 200MHz. Cependant, les FPGA peuvent surpasser les processeurs en implantant plusieurs briques de calcul en parallèles, personnalisées et pipelinées [Thomas et al., 2009, Koch and Torresen, 2011]. D'autre part, pour élargir le spectre d'utilisation des FPGAs et augmenter leurs performances, la nouvelle génération des FPGAs ont tendance à inclure de plus en plus des composants de calcul de différents types sur la même puce. Des FPGAs comme Cyclone V, Arria V et X de Altera ainsi que Virtex-4 et Virtex-5 de Xilinx intègrent des blocs DSP dédiés pour effectuer des opérations mathématiques telles que l'addition/soustraction ou la multiplication/accumulation.

Altera et Xilinx intègrent aussi des processeurs sur le FPGA. On distingue deux types de processeurs embarqués : processeurs softcores et processeurs hardcores. Les processeurs softcores sont des blocs IP. Ils sont implantés en utilisant des cellules logiques du FPGA, à partir d'une description de haut niveau (VHDL ou VERILOG). Parmi les processeurs softcores, on trouve le NIOS-II de Altera ainsi que PicoBlaze et Microblaze de Xilinx. Il existe également plusieurs processeurs open-source tels que le Tiger MIPS de l'université de Cambridge. Les performances d'un processeur softcore restent limitées par rapport à un processeur hardcore (ARM, PowerPC, Intel,...). Ces derniers peuvent aussi être implantés en dur sur le FPGA. Xilinx embarque des processeurs ARM (Cortex-A9) dans le Zynq-700 et des processeurs PowerPC (PowerPC440 et PowerPC405). Altera utilise des processeurs ARM (Cortex-A9) dans les circuits Cyclone V, Stratix V, Arria V et Arria X. Il est également possible d'instancier plusieurs processeurs softcores à côté du processeur hardcore ; l'ensemble pouvant exploiter les performances des accélérateurs câblés dans le FPGA.

2.3.2 Architecture système-sur-puce du Cyclone V

Parmi les architectures hétérogènes à base de FPGA et intégrant un processeur hardcore, on citera le Cyclone V de Altera. La figure 2.6 présente le diagramme de blocs du Cyclone V. Le processeur hardcore ou HPS (Hard Processor System) est en réalité implanté en dur sur la fabrique FPGA. Dans le Cyclone V, le HPS est un Cortex-A9 dual core. Le FPGA peut disposer d'un ou plusieurs contrôleurs mémoire ainsi que d'une liaison PCI-express.

Le FPGA et le HPS peuvent opérer indépendamment l'un de l'autre. Cette configuration semble peu intéressante en termes de performances, mais elle est très utile en prototypage. La coopération entre le HPS et le FPGA, pour tirer profit des avantages de chacun, constitue l'objectif de ce type d'architectures. Pour ce faire, le HPS et le FPGA sont étroitement interconnectés via des interfaces AXI de 32, 64 et 128 bits. Ces interfaces sont aussi appelées bridges (ponts). Le FPGA peut accéder aux bus esclaves du HPS via l'interface FPGA-to-HPS. De la même manière, le HPS a accès aux bus esclaves du FPGA via l'interface HPS-to-FPGA. Le HPS peut, par ailleurs, programmer le FPGA via un port de configuration de 32 bits.

Dans ce travail de thèse, nous avons utilisé comme plateforme d'évaluation le DE1-SoC qui intègre un Cyclone V. Respectant l'architecture générale de la figure 2.6 , le FPGA de cette plateforme implante

un Dual-core ARM Cortex-A9 opérant à 800 MHz. Pour ce qui est de la mémoire externe, cette plateforme dispose de deux mémoires : une SDRAM de 64 Mo pour le FPGA ainsi qu'une DDR3 de 1 Go pour le HPS. Le bus de données de la mémoire FPGA est 16 bits, alors que celui de la mémoire HPS est 32 bits. Le FPGA a également deux contrôleurs mémoire. Les principales spécifications du FPGA sont résumées dans le tableau 2.2.

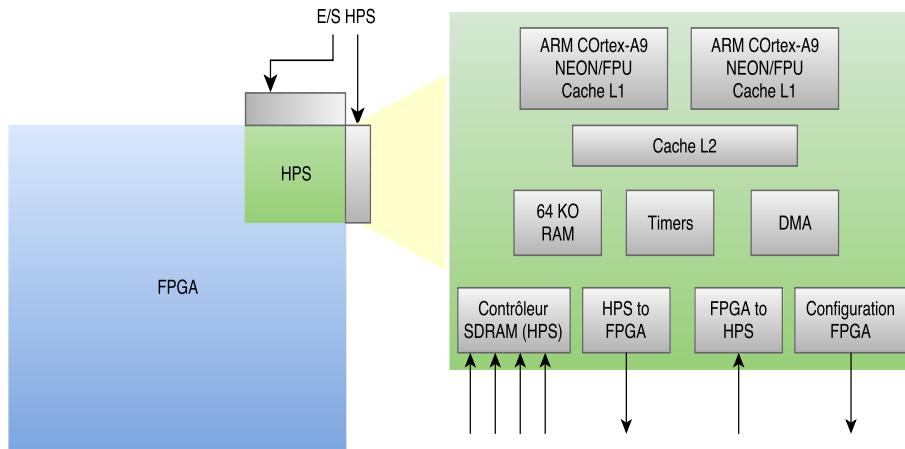


FIGURE 2.6 – Diagramme de blocs du Cyclone V.

TABLE 2.2 – Spécifications du FPGA de la plateforme DE1-SoC.

Spécifications	
FPGA	Cyclone® V SE 5CSEMA5F31C6N
Élément logiques	85K
Horloge	50 MHz
Mémoire sur puce (Onchip-memory)	4450 Kbits
Mémoire externe	64 Mo (32M x 16) SDRAM

L'objectif de se contraindre au niveau de l'architecture était pour nous obliger à mieux exploiter l'existant de sorte à faire émerger des méthodes et techniques de réflexions sur l'algorithmie et son efficacité sur la cible matérielle.

2.3.3 FPGA pour le SLAM

Le FPGA offre une grande flexibilité dans la conception des systèmes embarqués, avec une faible consommation d'énergie par rapport aux processeurs programmables. Son spectre d'utilisation est très large. On le retrouve pratiquement dans tous les domaines. Le FPGA a également attiré l'attention de la communauté scientifique quant à l'accélération et la conception de systèmes de SLAM embarqués. Dans la plupart des cas, le FPGA est utilisé pour accélérer la détection, la mise en correspondance des amers ou encore les calculs matriciels. [Bonato et al., 2006] ont conçu un système de SLAM basé sur l'EKF-SLAM en se servant d'un Stratix (FPGA EP1S10F780C6). L'algorithme de SLAM est exécuté sur un NIOS II instancié sur le FPGA. Les auteurs annoncent un système capable de

traiter 30 images/s en couleur et 60 images/s en niveau de gris. [Mingas et al., 2012] ont introduit le SMG-SLAM (Scan-Matching Genetic SLAM). La mise en correspondance entre les faisceaux laser fournis par un capteur laser (Laser Range Finder) est réalisée via un algorithme génétique. Celui-ci est implanté en matériel sur un FPGA. [Cruz et al., 2013] ont implanté la phase de mise à jour de l'EKF-SLAM sur un FPGA. [Tertei et al., 2016] ont présenté un système de SLAM visuel 3D basé sur l'EKF-SLAM. L'algorithme est entièrement implanté sur une plateforme Zynq-7020 (ARM+FPGA). Pour accélérer les traitements, les auteurs déportent le calcul matriciel sur le FPGA. Les auteurs affirment que leur système est capable de maintenir et corriger, à une fréquence de 30Hz, une carte de 20 amers avec une paramétrisation AHP (Anchored Homogeneous Point). [Gu et al., 2015] ont proposé un système d'odométrie visuelle basé caméra stéréo. L'algorithme est implanté sur un Stratix III EP3SL340 en utilisant un NIOS II comme processeur maître. Le FPGA se charge essentiellement du calcul matriciel. La fréquence de traitement du système atteint les 31 images/s avec une carte de 30000 amers. [Nikolic et al., 2014] ont proposé un système d'odométrie visuelle destiné pour les MAVs (Micro Air Vehicle). Le système embarque une centrale inertie avec quatre caméras interfacées à une plateforme Zynq-7020 (ARM+FPGA). Pour améliorer les performances temporelles, le traitement d'image (e.g. la détection de points d'intérêts) est assuré par le FPGA. D'autre part, [Sileshi et al., 2015, 2016b,a] ont effectué des travaux pour accélérer le SLAM basé filtre particulaire (FastSLAM) sur FPGA. Dans une approche logicielle/matérielle, les auteurs répartissent les tâches de l'algorithme sur un processeur embarqué (Microblaze) et un accélérateur FPGA.

Beaucoup d'expériences de prototypages sur cibles simples sont effectuées dans la communauté scientifique. L'objectif est d'arriver à faire émerger de l'innovation et de l'amélioration algorithmique à partir d'une architecture FPGA bas coût (ou non complexe).

2.4 Flux de conception sur FPGA : Outils de synthèse de haut niveau

Bien que les FPGAs présentent des avantages apparents quant au parallélisme et la consommation d'énergie, la description des architectures à base de FPGA est relativement complexe par rapport aux processeurs programmables. Depuis l'apparition des FPGAs dans les années 80, les approches et les outils de conception ont connu une grande évolution dans l'objectif de faciliter la description matérielle et la rapprocher des langages de haut niveau. On distingue deux types de descriptions matérielles sur FPGA : la description manuelle et la description automatisée ou synthèse de haut niveau.

2.4.1 Description manuelle

En 1971, [Bell and Newell, 1971] ont introduit , pour la première fois, le concept de description RTL (Register-Transfer Level). Pour décrire le comportement du circuit à réaliser, le concepteur spécifie des registres, décrit les transferts possibles entre ces registres ainsi que les opérations logiques effectuées sur les signaux d'entrées/sorties. Le concepteur peut également définir des contrôles sur les

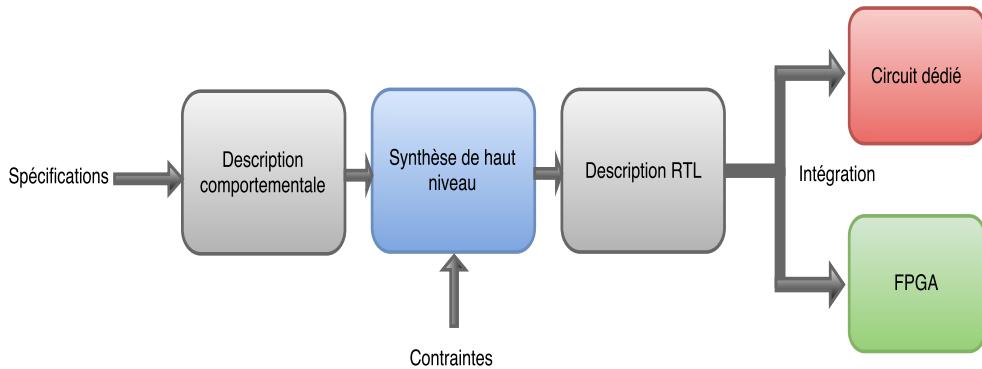


FIGURE 2.7 – Flux de synthèse de haut niveau.

transferts et les opérations logiques. Le concept RTL constitue le fondement de tous les langages dits de description RTL. Les langages HDL (Hardware Description Languages) permettent de décrire le comportement et la structure d'un circuit à un niveau abstrait ressemblant au langage C. Contrairement aux langages comportementaux tels que C et Matlab, les langages HDL permettent de décrire un comportement concurrentiel au niveau instruction, et d'exprimer le temps.

En 1983, le langage ABEL (Advanced Boolean Expression Language) a été créé [Lee et al., 1985]. Plus tard dans les années 80, il y a eu la création des deux langages les plus utilisés aujourd'hui, à savoir VHDL (Very High Speed Integrated Circuit, Hardware Description Language) et Verilog. Ces derniers doivent leur succès, entre autres, à la standardisation par l'Institute of Electrical and Electronics Engineers (IEEE).

La description manuelle s'appuie sur l'expertise technique qui mène à des architectures fortement optimisées ou encore très personnalisées. D'autre part, bien qu'elle soit intéressante sur le plan maîtrise générale de l'architecture et des performances, la description manuelle présente des inconvénients. Elle nécessite souvent un temps relativement important pour le développement, la vérification et surtout la maintenabilité. La description HDL présente ainsi un inconvénient majeur quant au TTM (Time-To-Market). Pour accélérer le développement et rendre accessible la description matérielle aux non experts, des outils de synthèse de haut niveau ont été créés permettant de générer automatiquement une description HDL.

2.4.2 Synthèse de haut niveau

La synthèse de haut niveau ou HLS (High-Level Synthesis) [Casseau and Le Gal, 2009, Cong et al., 2011a, Coussy and Takach, 2008, Coussy et al., 2009, Martin and Smith, 2009] consiste en la génération automatisée de descriptions HDL à partir d'une description comportementale effectuée au moyen d'un langage haut niveau tel que C, SystemC, C++, Matlab. L'interprétation de l'application et la génération de la description matérielle sont guidées par un ensemble de contraintes introduites par le concepteur.

Le flux de conception par synthèse de haut niveau est schématisé dans la figure 2.7. Partant du cahier des charges du circuit à réaliser, le concepteur établit l'algorithme de fonctionnement, puis le décrit à

l'aide d'un langage comportemental (C, SystemC, C++, Matlab, ...). A ce stade, on peut déjà vérifier et valider le comportement du système. Cela est un atout précieux en comparaison avec les langages HDL. Bien que ces derniers permettent de vérifier le comportement par simulation, celle-ci est parfois très contraignante en temps. D'autre part, le concepteur peut spécifier les contraintes de génération portant sur les ressources, la communication, les librairies à utiliser, etc.

Une fois que l'application et les contraintes de génération sont prêtes, arrive l'étape de synthèse de haut niveau. Les outils de synthèse procèdent premièrement par une analyse de syntaxe et de sémantique afin de valider que le code d'entrée est synthétisable. Les synthétiseurs modernes sont couplés à des compilateurs comme gcc et LLVM [Lattner and Adve, 2004, Lattner, 2008]. Ce dernier est un compilateur open-source utilisé autant par les industriels que par la communauté scientifique. Le parseur (analyseur) clang de LLVM transforme le code en entrée en une représentation intermédiaire ou IR (intermediate representation). Celle-ci est un code assembleur indépendant de la machine et composé d'instructions simples (addition, décalage, multiplication, branchement,...). Pour ce faire, le parseur effectue plusieurs passes pour optimiser le code en entrée. Il peut, par exemple, éliminer les portions de code inaccessibles et dérouler les boucles. Même si ces passes sont à l'origine destinées à l'optimisation des codes logiciels, elles représentent une phase essentielle avant le début de la synthèse. Celle-ci peut être divisée en trois parties [Coussy et al., 2009, Gajski et al., 2012, Elliott, 2012] : allocation, ordonnancement et assignation.

Allocation : Cette étape détermine puis alloue les ressources matérielles nécessaires (unités fonctionnelles, bus, etc). Le nombre de ressources est déterminé en fonction des contraintes introduites par le concepteur ainsi que des latences des ressources elles-même.

Ordonnancement : Cette étape consiste à associer à chaque opération une date d'exécution. Les dépendances de données entre opérations, le nombre et le type des ressources déterminent l'ordre dans lequel les opérations doivent être exécutées.

Assignation : Cette étape affecte chaque opération à une unité fonctionnelle. Les signaux d'entrées/sorties ainsi que les variables temporaires sont assignés à des registres. Un registre peut être partagé par deux variables différentes si leurs utilisations ne se chevauchent pas dans le temps. Une fois les unités fonctionnelles et les registres de mémorisation sont alloués, on interconnecte l'ensemble des ressources en utilisant des ressources d'interconnexion (multiplexeur, bus trois-états, etc).

A la fin des étapes précédentes, on peut procéder à la génération d'une architecture RTL comprenant un chemin de données et une machine à états finis ou FSM (Finite-State Machine). Cette dernière cadence le flux de données dans le chemin de données via des signaux de contrôle.

Pour ce qui est des outils de synthèse de haut niveau existants, ils sont nombreux et partagés entre l'industrie et le monde académique². Ces outils sont, en grande majorité, basés sur le langage C [Edwards, 2006]. Dans le monde académique, plusieurs travaux sur la synthèse HLS ont été effectués dont certains ont donné lieu à des outils HLS souvent open-source. Parmi ces derniers, on trouve Gaut [Coussy et al., 2008] de l'Université de Bretagne Sud, GraphLab [Le Gal and Casseau, 2011] de

2. Le lecteur pourra trouver plus de détails sur les outils de synthèse de haut niveau dans [Cong et al., 2011b].

l’Université de Bordeaux, SPARK [Gupta et al., 2003] de l’Université de Californie, Trident Compiler [Tripp et al., 2007], ROCC [Guo et al., 2008, Villarreal et al., 2010] de l’Université de Californie, et LegUp [Canis et al., 2011] de l’Université de Toronto. D’autre part, les principaux outils industriels sont Catapult C [Bollaert, 2008], Vivado [Feist, 2012], AutoPilot [Zhang et al., 2008], Symphony C Compiler [synopsys, 2016], Handel-C [Solutions, 2008], C-to-Silicon [Cadence, 2011], C2H [Corp., 2009].

Par ailleurs, pour standardiser la conception sur les architectures hétérogènes, Altera et Xilinx ont adopté le standard OpenCL pour la génération d’architectures RTL. Dans la suite, nous allons donner un aperçu sur LegUp ainsi que l’approche OpenCL que nous avons utilisés dans notre étude.

2.4.3 Présentation de LegUp

LegUp [Canis et al., 2011] est un outil de synthèse de haut niveau développé par l’Université de Toronto. En plus de son avantage d’être open-source, il supporte, depuis la version 4.0 [Toronto, 2015], les architectures hétérogènes type Cyclone V (DE1-SoC, SoCkit) de Altera. Le Framework de LegUp se compose de deux parties. La première consiste en un compilateur C. La seconde partie est un synthétiseur de haut niveau. En termes de conception logicielle/matérielle, LegUp propose trois flux de conception :

Conception logicielle : Ce flux de conception est important pour vérifier et valider le comportement de l’application en utilisant le code binaire. LegUp utilise, pour la partie logicielle, le processeur Tiger MIPS qui a été créé par l’université de Cambridge. Celui-ci est implanté comme un processeur softcore sur le FPGA lors de la synthèse. La version 4.0 de LegUp prend également en charge le processeur hardcore (ou HPS) des architectures hétérogènes (SoCKit et DE1-SoC).

Conception purement matérielle : Dans ce flux, la totalité du code est synthétisé sur FPGA.

Conception hybride : Dans ce flux, le concepteur partitionne le code en plusieurs segments. Il peut ensuite choisir quels segments de l’application à planter sur le FPGA en vue d’améliorer les performances (temps de calcul, consommation d’énergie). Le reste du code est exécuté par le processeur embarqué (ARM ou Tiger MIPS).

Nous nous intéressons particulièrement à la conception hybride (hétérogène). La figure 2.8 illustre le flux de cette conception. Celui-ci comprend six étapes :

- Étape 1 : LegUp utilise le compilateur LLVM pour générer un code binaire exécutable à partir du code C en entrée. Ce dernier doit être écrit selon la norme ANSI C.
- Étape 2 : Exécution et profilage du code binaire sur le Tiger MIPS ou le HPS (ARM).
- Étape 3 : LegUp effectue un partitionnement logiciel/matériel du code binaire. Pour ce faire, LegUp consulte un fichier texte introduit par le concepteur. Le partitionnement se fait au niveau des fonctions C. Des directives sont à la disposition du concepteur pour indiquer les fonctions à planter comme accélérateurs sur FPGA. LegUp génère une description RTL (Verilog) de tout le système comprenant le processeur embarqué et les accélérateurs implantés sur FPGA.

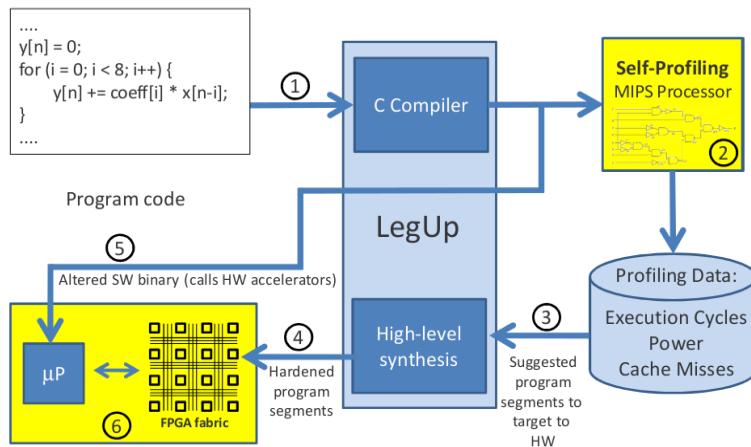


FIGURE 2.8 – Flux de conception de LegUp [Canis et al., 2011].

- Étape 4 : A cette étape, la description RTL est synthétisée en utilisant un outil commercial. Nous avons utilisé Quartus de Altera. QSys³ est utilisé pour générer l’architecture du système. Il est à noter qu’il est possible avant la synthèse d’intervenir sur la description Verilog pour ajouter des librairies, voire modifier la description. D’autre part, LegUp donne la possibilité de décrire entièrement un module Verilog. LegUp se contente, dans ce cas, de générer les interfaces de communication nécessaires. Les modules à décrire manuellement peuvent être indiqués par le concepteur au même titre que le partitionnement.
- Étape 5 : Pour assurer la communication entre le FPGA et le processeur embarqué, les fonctions implantées sur FPGA sont remplacées par des adaptateurs (wrappers). Ceux-ci se chargent de lancer les accélérateurs. Le code modifié est à nouveau compilé en un code binaire exécutable.
- Étape 6 : Le système peut être lancé. Notons que dans le cas où l’on utilise le HPS, LegUp ne permet pas, dans sa version actuelle, d’utiliser un système d’exploitation. Le HPS est donc utilisé comme un puissant microcontrôleur.

L’architecture générale d’un système hétérogène généré par LegUp est illustrée dans la figure 2.9. Elle comprend principalement des accélérateurs implantés sur FPGA, un processeur embarqué et des composants mémoire. La communication entre le processeur et les accélérateurs est assurée par une logique d’interconnexion AVALON. Celle-ci est automatiquement générée par QSys. L’accès à la mémoire est géré par un contrôleur mémoire. Le cache implanté sur FPGA (On-chip Cache) est partagé entre les accélérateurs et le processeur MIPS s’il est utilisé.

Concernant l’implantation des accélérateurs. Si l’on veut synthétiser une fonction et que celle-ci fait appel à d’autres fonctions, alors ces dernières sont également synthétisées en accélérateurs. La figure 2.10 donne un exemple d’instanciation d’accélérateurs. Les fonctions a et b sont synthétisées en deux accélérateurs visibles et contrôlés par le processeur embarqué. La routine c, appelée depuis a et b, est instanciée deux fois. En d’autres termes, elle est instanciée dans tous les accélérateurs qui lui font appel. La duplication peut améliorer les performances temporelles du système. Elle peut, néanmoins, augmenter la consommation de ressources logiques si les routines dupliquées en requièrent beaucoup.

3. QSys est un outil d’intégration système d’Altéra.

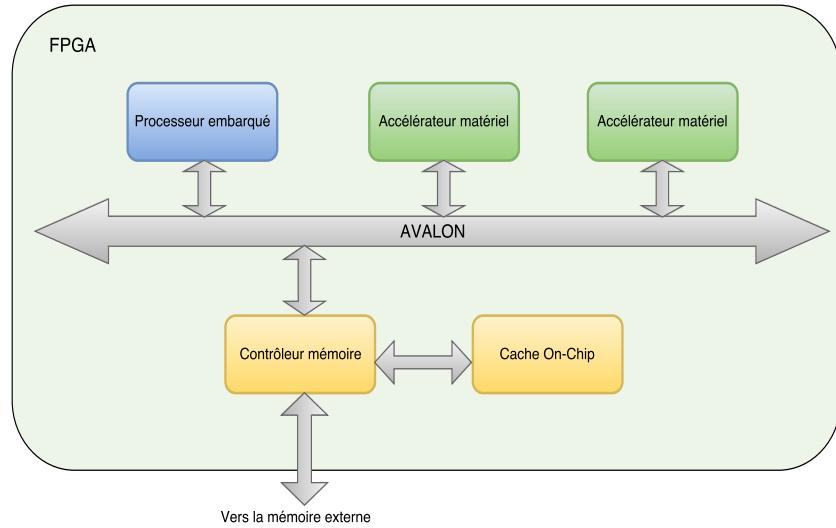


FIGURE 2.9 – Architecture d'un système hétérogène généré par LegUp.

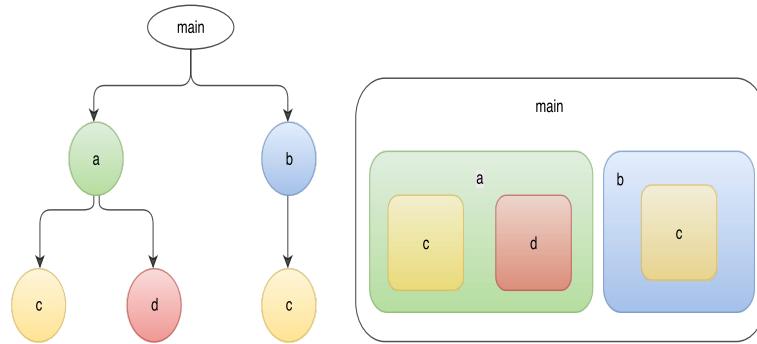


FIGURE 2.10 – Hiérarchie d'instanciation des accélérateurs avec LegUp. A gauche l'arbre d'appels ; à droite l'architecture instanciée.

Il est intéressant de faire un compromis entre la fréquence maximale et les ressources concernant la duplication.

Pour l'architecture mémoire, LegUp utilise quatre niveaux de mémoire : mémoire locale (registre), mémoire globale, mémoire cache et mémoire externe. Les deux dernières sont partagées par le processeur embarqué et les accélérateurs FPGA.

- Mémoire locale : Les mémoires locales sont utilisées pour stocker les variables locales à la fonction synthétisée. Elles sont implantées à l'intérieur des accélérateurs. Un accélérateur peut avoir plusieurs mémoires locales. Celles-ci sont implantées en utilisant des blocs RAM séparés et donc peuvent être accédées en parallèles.
- Mémoire globale : On y met les variables partagées par plusieurs fonctions C (accélérateurs).
- Mémoire cache : Cette mémoire est implantée sur FPGA et elle est partagée entre le processeur embarqué et les accélérateurs. LegUp implante un mécanisme d'accès à cette mémoire pour garantir l'intégrité des données.
- Mémoire externe : LegUp permet aux accélérateurs et au processeur embarqué d'accéder aux mémoires externes. Pour le DE1-SoC, la version 4.0 de LegUp ne prend en considération que la mémoire HPS pour une conception hybride. Nous nous sommes donc limités à la mémoire

HPS.

Finalement, LegUp est un outil de synthèse qui est en pleine expansion en termes d'utilisation par la communauté scientifique. Dans sa version actuelle, il a des limitations concernant plus particulièrement la consommation de ressources logiques. LegUp n'intègre pas encore de systèmes d'exploitation pour le HPS. En attendant le support d'autres plateformes (Arria V), LegUp constitue d'ores et déjà un outil capable de prototyper des systèmes dont les performances temporelles sont compétitives avec les outils commerciaux [Canis et al., 2011]. Notre expérience avec LegUp dans le contexte du GraphSLAM sera discutée dans le chapitre 5.

2.4.4 Approche OpenCL

Dans la synthèse de haut niveau basée langage C, le concepteur nécessite de réécrire le code selon l'outil utilisé. Le niveau d'abstraction de haut niveau nécessite une expertise sur l'outil et l'architecture cible pour maximiser les performances. L'idée de l'approche OpenCL est de permettre à l'utilisateur de concevoir une architecture hétérogène en se basant sur le standard OpenCL. Dans ce type d'architectures, on distingue deux parties : l'hôte et l'accélérateur FPGA. Ce dernier joue alors le rôle d'un coprocesseur au même titre qu'un GPU.

Il y a eu ces dernières années plusieurs travaux, menés par des industriels et des académiques, pour concevoir des outils de synthèse de haut niveau basés OpenCL ou encore CUDA. FCUDA [Papakonstantinou et al., 2009] est un outil de synthèse qui prend en entrée un code CUDA. Ce dernier est transformé en un code C synthétisable par AutoPilot [Zhang et al., 2008]. Celui-ci génère une architecture personnalisée, pour le code en entrée, basée sur une machine à états finis (FSM) et un chemin de données. Le circuit créé comprend plusieurs coeurs dont chacun est conçu pour un seul thread. OpenRCL [Lin et al., 2010] est un outil qui, à partir d'une application OpenCL, extrait un parallélisme et synthétise une architecture sur FPGA. Le calcul parallèle est réalisé à travers l'instanciation de plusieurs processeur MIPS sur le FPGA. [Jaaskelainen et al., 2010] ont proposé un flux de synthèse basé sur OpenCL en utilisant des architectures Transport-Triggered (Transport-Triggered Architectures). L'architecture comprend un processeur VLIW qui implante un parallélisme au niveau instruction. [Owaida et al., 2011] ont proposé un Framework appelé SOpenCL. Celui-ci utilise une machine à états finis avec un chemin de données adéquat à l'exécution d'un kernel OpenCL. Ce chemin de données est figé (défini au préalable). Un chemin de données figé limite les performances globales. En effet, on a moins de souplesse, par exemple, quant au choix des tailles de données.

Altera a été le premier vendeur à fournir un Framework complet pour la synthèse de haut niveau basée OpenCL [Czajkowski et al., 2012]. L'outil de compilation et de synthèse est appelé AOCL (Altera Offline CompiLer). La particularité de AOCL réside dans la stratégie avec laquelle le parallélisme est traité sur le FPGA. Dans cette approche, le chemin de données n'est pas figé et il est créé en fonction de chaque kernel. AOCL ne synthétise pas non plus des processeurs softcores (NIOS II, Tiger MIPS, Microblaze,...). Un kernel est implanté en une brique calculatoire dotée d'un pipeline très profond en vue d'augmenter le débit de calcul. La stratégie d'exécution des threads est, donc, différente de celle des processeurs graphiques. Dans la même optique, Xilinx s'est également lancée dans cette

nouvelle technologie en créant un outil appelé Sdaccel [Xilinx, 2014]. Celui-ci a été intégré dans Vivado. Récemment, [Richter-Gottfried et al., 2015] ont présenté OCLAcc qui est un générateur basé OpenCL. Les auteurs discutent l’implantation de certaines fonctionnalités de OpenCL 2.0.

Quant à l’utilisation de la technologie OpenCL, des travaux de recherche ont été menés pour principalement évaluer son apport vis-à-vis des langages de description HDL. [Warne et al., 2014] présente une comparaison entre AOCL, Vivado et une description manuelle en VHDL d’un algorithme d’algèbre linéaire. La comparaison a porté sur les performances en termes d’utilisation de ressources, de facilité de développement et de précision de calcul. Les résultats on montré l’avantage de Vivado. Cependant, les résultats pour Vivado et la description VHDL ont été extraits à partir d’une simulation ; alors que AOCL a été testé sur une plateforme Bittware S5PH-Q avec une liaison PCIe entre le FPGA et l’hôte. [Hill et al., 2015] ont réalisé une étude d’implantation de certains opérateurs de traitement d’image (Sobel, Canny edge et Surf) en utilisant une description manuelle en VHDL ainsi que AOCL comme outil de synthèse de haut niveau. Les kernels ont été implantés sur un FPGA Stratix-V. Pour les plateformes (hôte + FPGA), les auteurs ont utilisé Gidel ProceV, PCIe385-D5 de Nallatech et S5PH-Q de Bittware. Pour OpenCL, le même code a pu être porté sur les trois plateformes. Alors que les résultats en termes de performances (images/s) sont très comparables (2 à 10% de différence), la description manuelle est avantageuse en termes d’utilisation de ressources. En effet, AOCL consomme jusqu’à 70% plus de ressources par rapport à la description VHDL. Cependant, en termes de développement, OpenCL est 6 fois plus rapide comparé à la description VHDL.

a. Altera Offline Compiler (AOCL)

Dans ce travail, nous avons utilisé AOCL pour générer une architecture hétérogène sur la plateforme DE1-SoC. Nous allons, dans la suite, présenter la stratégie d’implantation et d’exécution des kernels dans AOCL.

De la même manière que sur GPU, la conception par AOCL est divisée en deux parties parallèles : la partie hôte et la partie FPGA (device). Le code à exécuter sur l’hôte est compilé en fonction de l’architecture cible (Intel X86, ARM,...). AOCL comprend aussi un synthétiseur. Celui-ci génère comme tout outil de HLS une description RTL (Verilog) avant de la synthétiser en une architecture. Pour ce faire, le concepteur regroupe les kernels dans un seul fichier qui sera utilisé par AOCL. L’ensemble des kernels est synthétisé en des briques personnalisées et fortement pipelinées. AOCL génère ensuite un fichier binaire pour la programmation du FPGA. L’utilisateur peut alors lancer l’accélérateur (les kernels) à partir du code hôte. Celui-ci jouera le rôle d’un coordinateur entre le kernels FPGA et le processeur hôte. AOCL peut être utilisé pour les systèmes discrets (FPGA et machine hôte physiquement séparés) comme pour les plateformes intégrant un processeur embarqué et un FPGA sur la même puce (Cyclone V, Arria 5, Stratix V,...).

b. Modèle d'architecture et notion de parallélisme

Pour les GPUs, les kernels sont compilés en une séquence d'instructions exécutées sur une architecture figée. Celle-ci est constituée de plusieurs cœurs de calcul. Pour les FPGAs, AOCL implante chaque kernel en un circuit personnalisé pour efficacement effectuer le calcul associé au kernel. L'implantation d'un kernel utilise les différentes ressources logiques (ALMs, DSP, blocs de mémoire, etc). La figure 2.11 donne un exemple de kernels OpenCL. Il s'agit d'implanter sur FPGA l'addition de deux vecteurs a et b de n éléments. A la différence des GPUs, un kernel est transformé en un pipeline à plusieurs étages. Le concepteur peut dupliquer le même pipeline plusieurs fois. AOCL génère également toute la logique d'interconnexion entre les kernels au sein du FPGA et avec le processeur hôte.

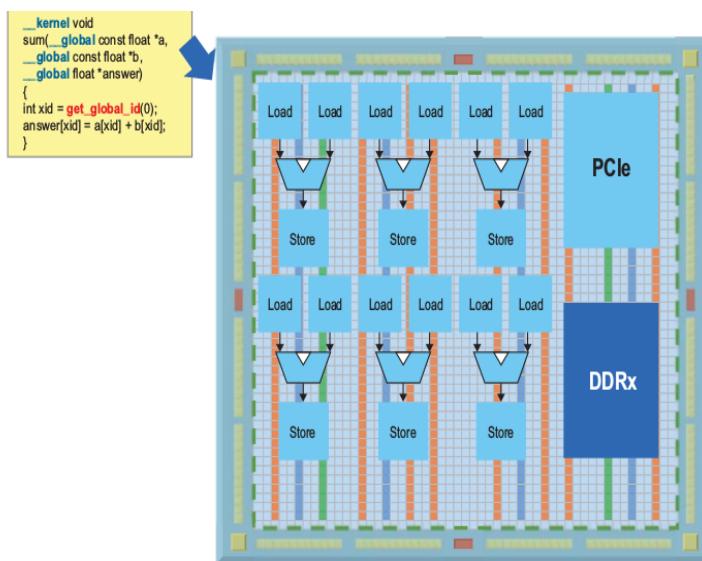


FIGURE 2.11 – Exemple d'architectures générées par AOCL.

Pour l'exemple d'addition de vecteurs, le pipeline synthétisé comprend trois étages. Le chargement de a et b donne lieu à un étage de chargement (Load unit). Un circuit load contient toute la logique nécessaire pour accéder à la mémoire externe et récupérer les données. Celles-ci sont ensuite passées à un deuxième étage (additionneur) qui va effectuer l'opération d'addition flottante des deux éléments. Enfin, le résultat est rangé en mémoire externe à l'aide d'un dernier étage du pipeline qui sert d'une unité de rangement (Store unit).

La notion de parallélisme dans AOCL est différente de celle des GPUs. Les GPUs sont des processeurs SIMD (single-instruction multiple-data) où un ensemble de threads (généralement 32 threads) exécutent la même instruction sur l'ensemble des données correspondantes. Par ailleurs, AOCL implante un parallélisme de pipeline où les threads exploitent concurremment le pipeline. Pour illustrer ce mécanisme, nous reprenons le même kernel effectuant l'addition de vecteurs. Comme illustré dans la figure 2.12, le pipeline est constitué de trois étages. Au premier cycle, le thread 0 occupe l'étage de chargement (Load). Les éléments récupérés sont stockés dans des registres. Au deuxième cycle, le thread 0 passe à l'additionneur et libère l'étage de chargement au thread 1. Au troisième cycle,

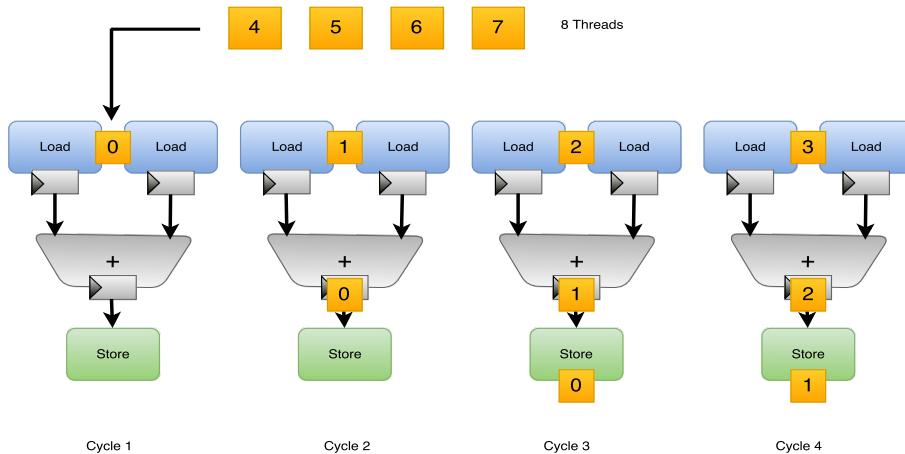


FIGURE 2.12 – Parallélisme pipeliné dans AOCL.

thread 0 sera au troisième étage en train de ranger le résultat d'addition en mémoire externe. Thread 1 occupe l'additionneur et thread 2 vient prendre l'étage de chargement. De cette manière, les threads passeront un après l'autre sur les trois étages. A partir du troisième cycle, le débit de sortie est égal à un élément par cycle. Cela revient à dire qu'en ayant un seul pipeline, on a besoin de $(n + 2)$ cycles pour calculer les n éléments du vecteur de sortie.

Cette notion de parallélisme constitue un des principaux avantages de AOCL. La stratégie de AOCL a l'avantage de pallier au problème de branchement lors de l'exécution SIMD sur GPU. L'unité SIMD du GPU opère sur une seule instruction à la fois. A la rencontre d'une condition à N options, l'unité SIMD est contrainte d'exécuter tous les chemins de code possibles selon l'évaluation de la condition par les threads. Par exemple, supposons que sur 32 threads, cinq d'entre eux, ont chacun choisi un chemin différent du reste des threads. Dans ce cas, les six chemins sont exécutés l'un après l'autre en désactivant, à chaque fois, les threads qui ne satisfont pas la condition. Cela casse le parallélisme et dégrade donc les performances. Sur FPGA, les branchements ne posent pas tant de problèmes du moment que tous les chemins possibles sont synthétisés et peuvent s'exécuter en parallèle. D'autre part, le pipeline masque considérablement les latences d'accès à la mémoire permettant ainsi d'augmenter le débit par cycle.

L'approche OpenCL présente une technologie prometteuse dans la conception des systèmes hétérogènes à base de FPGA. En étant un standard, les kernels de OpenCL sont portables (en code) et peuvent être compilés pour différentes plateformes sans modification [Hill et al., 2015]. D'autre part, le parallélisme pipeliné de AOCL permet d'augmenter les performances en améliorant le débit de sortie. Néanmoins, l'implantation d'un pipeline nécessite l'utilisation de beaucoup de ressources de mémorisation pour traiter les aléas de contrôle et de données entre les différents étages. Cela augmente la consommation de ressources par rapport à une description HDL où le concepteur peut en optimiser l'utilisation. Finalement, il est évident que l'approche OpenCL permet de réduire considérablement le temps de développement grâce à son haut niveau d'abstraction.

2.5 Conclusion

Nous avons présenté dans ce chapitre les architectures de calcul que nous avons utilisées dans notre étude d’implantation du GraphSLAM. Les architectures parallèles constituent une technologie clé dans la conception des systèmes embarqués d’aujourd’hui. L’utilisation d’un seul type de calculateurs pourrait limiter les performances du système. Il s’avère souvent qu’une tâche calculatoire est adaptée à un calculateur donné plutôt qu’un autre. La combinaison de plusieurs calculateurs de différents types dans un même système hétérogène permet de concevoir des systèmes performants dans une approche Adéquation Algorithme-Architecture (AAA). Celle-ci a pour objectif de répartir les tâches d’un algorithme donné sur les calculateurs disponibles dans le but de minimiser une fonction objectif incluant, entre autres, le temps de calcul et la consommation d’énergie.

Nous avons également présenté les approches de conception sur FPGA en mettant en avant les atouts de la synthèse de haut niveau par rapport à la description manuelle. Celle-ci a l’avantage de fournir des solutions hautement optimisées, particulièrement, en termes d’utilisation de ressources logiques. Cependant, elle présente l’inconvénient d’être contraignante en temps de développement, particulièrement, si l’on veut réaliser rapidement des prototypes ou encore vérifier l’adéquation du FPGA avant une mise en œuvre manuelle.

L’avènement des systèmes sur puce (SoCs) hétérogènes devrait mener à une grande avancée dans la conception des systèmes embarqués destinés à la robotique mobile. Il s’agit, dans notre travail, d’explorer dans un premier temps l’apport des processeurs graphiques dans l’atténuation de la complexité de calcul du GraphSLAM. Dans un second temps, nous exploitons les outils de synthèse de haut niveau pour étudier l’adéquation des FPGAs vis-à-vis du GraphSLAM et des problèmes de moindres carrés. Le choix des outils de synthèse de haut niveau nous permettra d’investiguer deux approches. La première est axée sur la parallélisation où une même brique est dupliquée plusieurs fois en vue d’accélérer les flots de traitements. La deuxième approche exploite la technologie OpenCL de Altera (AOCL) dont le paradigme de parallélisme dominant est le pipeline.

Chapitre 3

Représentation mémoire pour le GraphSLAM incrémental

Sommaire

3.1	Introduction	64
3.2	Représentation mémoire du GraphSLAM	64
3.3	Structure de données à index (IDS)	66
3.4	Structure de données compacte et incrémentale	68
3.4.1	Représentation des éléments du graphe	68
3.4.2	Connectivité : connexions entre nœuds	69
3.4.3	Complément de Schur	70
3.4.4	Système linéaire	72
3.4.5	Construction incrémentale du graphe	73
3.5	Évaluations et résultats temporels	73
3.5.1	Présentation des jeux de données	75
3.5.2	Modalités d'évaluation	75
3.5.3	Résultats fonctionnels	77
3.5.4	Présentation des architectures d'évaluation	78
3.5.5	Métrique d'évaluation : Temps de Traitement par Élément de graphe (TPE)	79
3.5.6	Charges temporelles des blocs fonctionnels	79
3.5.7	Évaluations temporelles sur un PC Intel	81
3.5.8	Évaluations temporelles sur architectures embarquées	84
3.5.9	Complément de Schur	86
3.5.10	Évolution du temps de traitement : TPE	87
3.6	Conclusion	91

3.1 Introduction

Nous avons vu dans le chapitre 1 une variante détaillée de l'algorithme GraphSLAM en mettant en évidence sa complexité algorithmique. Celle-ci est due à plusieurs paramètres inhérents à l'algorithme lui-même, mais également à la caractéristique de lissage. Comparativement aux ordinateurs classiques, les architectures embarquées disposent de ressources de calcul et de mémoire limitées.

Il est primordial dans un premier temps d'investiguer une représentation mémoire permettant de réduire les accès mémoire et donc d'exploiter efficacement celle-ci. Le problème de structure de données pour le GraphSLAM constitue la pièce maîtresse de ce chapitre. Nous avons proposé deux structures de données pour représenter le graphe et résoudre le problème de moindres carrés associé au GraphSLAM. Notre approche exploite le caractère épars et incrémental du problème. L'objectif est d'avoir une représentation mémoire compacte qui évite le plus possible le stockage de valeurs nulles en mémoire. Cela est motivé par le fait que les graphes et les systèmes linéaires résultant sont, en général, épars. La structure de données devrait aussi permettre un accès direct aux données pour réduire les accès mémoire. L'organisation des données devrait, par ailleurs, faciliter la parallélisation et le portage de l'algorithme sur des architectures parallèles hétérogènes.

La première structure de données proposée a pour objectif de représenter efficacement les connexions dans le graphe en effectuant un compromis entre, d'une part, l'utilisation d'espace mémoire, et d'autre part le coût de recherche et d'accès aux données. La deuxième représentation mémoire, quant à elle, est basée sur la première. Cependant, l'originalité est que notre approche n'est plus limitée uniquement à l'algorithme en tant qu'une succession de blocs fonctionnels. Nous allons plus loin en considérant à la fois l'algorithme lui-même et son exécution au fil du temps, c'est-à-dire l'historique des optimisations de graphe. En d'autres termes, les données fournies par la partie en-amont (observations) sont utilisées pour garder trace des connexions dans le graphe. On garde également trace des informations requises par les blocs calculatoires telles que les emplacements des éléments non nuls dans le graphe, et ainsi dans le système linéaire. On ajoute à cela une mise à jour incrémentale de la structure de donnée. Cela permet d'éviter de reconstituer à chaque étape et itération du NLS (processus de GN, LM, DOG-LEG,...) le voisinage d'un nœud dans le graphe.

Nous allons décrire, dans la suite de ce chapitre les deux structures de données proposées avec un retour d'expérience sur les structures de données utilisées dans les implantations de l'état de l'art (g^2o [Kummerle et al., 2011], iSAM [Kaess et al., 2008, 2012]).

3.2 Représentation mémoire du GraphSLAM

Dans le GraphSLAM, nous avons affaire en général à des graphes épars ou creux qui conduisent à des systèmes linéaires épars par blocs. La figure 3.1 montre un exemple de graphe épars accompagné de la structure par blocs de la matrice du système résultant. Un bloc non nul dans la matrice correspond à un élément dans le graphe. Celui-ci peut être un nœud (pose robot, amer) ou une arête (mouvement,

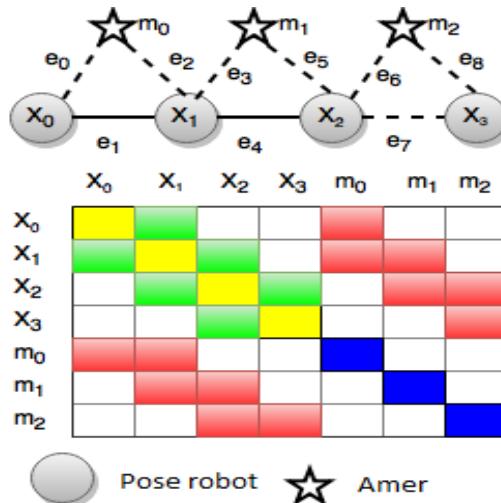


FIGURE 3.1 – Un exemple de graphe avec la structure épars par blocs de la matrice d’information associée.

observation). Une structure de données pour le GraphSLAM devrait permettre un accès rapide à toute donnée relative aux différents éléments du graphe ainsi qu'à la résolution du problème NLS. Par ailleurs, un paramètre très important dans une structure de données pour le GraphSLAM est la manière dont les voisinages des nœuds sont représentés et récupérés. Pour un nœud donné dans le graphe, nous avons besoin de garder trace de tous ses nœuds adjacents. La rapidité de récupération des voisins des nœuds impactera directement la vitesse de construction du système linéaire (FB3) et le format CCS (FB5).

Il est clair qu'une structure de données naïve sous forme d'une matrice bidimensionnelle (2D) permettrait un accès naturel à toutes les données dans le graphe. En effet, chaque entrée de la matrice peut contenir les données nécessaires associées au nœud (ou arête) correspondant telles que la position, les mesures capteur, l'erreur, le bloc correspondant dans la matrice d'information, etc. Une telle représentation mémoire n'exploite pas le caractère épars du problème. Par conséquent, elle est très contraignante en termes d'espace mémoire. Celui-ci croît quadratiquement avec le rajout de nouveaux noeuds robot ou amers. De plus, un espace mémoire très large peut affecter dramatiquement les performances en termes de calcul. Le GraphSLAM manipule et explore une grande masse de données à chaque itération du processus de Gauss-Newton. D'où, la mémoire cache peut ne pas être efficacement exploitée. La complexité de stockage mémoire, dans ce cas, est $O(N^2)$ comme dans l'EKF-SLAM.

Pour obtenir de meilleures performances en termes de temps de calcul, la structure de données utilisée devrait avant tout limiter autant que possible l'espace mémoire utilisé. L'idée est de ne stocker que les éléments non nuls du graphe et du système linéaire. Dans un second temps, une stratégie efficace est requise pour référencer et accéder à ces éléments.

Le problème de représentation mémoire, pour les problèmes à base d'optimisation de graphe et plus généralement les problèmes de moindres carrés, a été abordé dans les implantations de l'état de l'art. G^2o utilise une structure de données à base de blocs. En d'autres termes, les données sont organisées

en blocs de 3×3 , 2×2 et plus généralement $d_i \times d_j$, suivant les degrés de libertés des nœuds correspondants. L'objectif est d'accélérer les opérations de chargement et de rangement des données en mémoire. De la même manière, la matrice d'information (H) a une structure en blocs. Pour référencer les blocs non nuls, g^2o utilise une structure de données en arbre. Ainsi, un bloc dans la matrice d'information ou un élément dans le graphe est indirectement accessible car leur emplacement doit être d'abord retrouvé. Par ailleurs, iSAM utilise un format spécial de stockage épars pour les blocs la matrice d'information. En effet, une matrice creuse, dans iSAM, est représentée par un vecteur dense à la taille du nombre de colonnes où chaque élément est un vecteur épars contenant les blocs de la ligne correspondante. Pour accélérer la résolution du système linéaire, [Polok et al., 2013b] a proposé une organisation mémoire pour implanter les opérations matricielles par blocs dans les problèmes de moindres carrées non linéaires. Les blocs des matrices sont stockés dans des listes triées où chaque élément contient un index vers la ligne associée, ainsi qu'un pointeur vers le bloc de données. Pour accéder aux données, [Polok et al., 2013b] utilisent une fonction de mappage entre les colonnes et les lignes calculée avant les opérations arithmétiques avec un coût de $O(n)$ où n est le nombre de blocs par ligne (ou par colonne). L'inconvénient de cette organisation mémoire est le coût élevé engendré par l'insertion de nouveaux blocs dans la liste triée.

Il est à noter que toutes ces structures de données utilisent des formats de stockage épars qui sont compliqués à mettre à jour dans le GraphSLAM incrémental. En effet, l'accès à un bloc de données est toujours précédé par la recherche de son emplacement. Notons aussi que ces structures de données sont conçues et optimisées pour accélérer la résolution du système linéaire plutôt que sa construction. Notre objectif principal est de définir une structure de données dont la complexité de stockage mémoire sera linéaire en nombre de nœuds. Elle devrait aussi permettre de référencer efficacement la structure du graphe de telle manière à accélérer l'accès aux données et éviter la recherche de leurs emplacements en mémoire.

3.3 Structure de données à index (IDS)

Pour pallier aux problèmes de référencement des blocs non nuls et limiter la consommation d'espace mémoire, nous avons proposé une première structure de données offrant un compromis entre l'espace mémoire et le coût d'accès aux données. L'idée est d'organiser les données de telle manière à maximiser la localité temporelle et spatiale pour un meilleur bénéfice de la mémoire cache. Le graphe est représenté moyennant quatre vecteurs séparés tel que illustré dans la figure 3.2. L'utilisation de vecteurs séparés permet un accès direct aux données relatives aux éléments du graphe. Chaque entrée des vecteurs contient les données nécessaires de l'élément correspondant du graphe. De par le fait que chaque bloc dans le système linéaire est associé à un et un seul élément du graphe, et du fait de la symétrie de la matrice d'information, les blocs de cette dernière peuvent alors être stockés dans ces quatre vecteurs. Les blocs de la diagonale sont mis dans les vecteurs associés aux nœuds robot et amers. Quant aux blocs hors-diagonaux, ils sont placés dans les vecteurs d'arêtes de mouvement et d'observation.

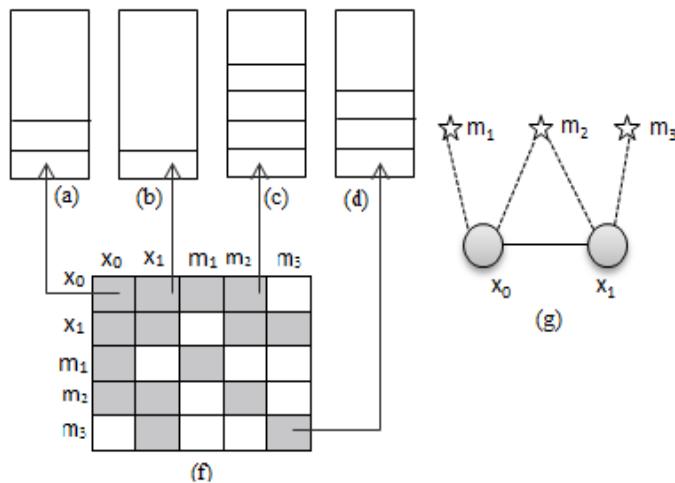


FIGURE 3.2 – Représentation mémoire du graphe dans IDS. (a) Poses, (b) Arêtes de mouvement, (c) Arêtes d'observation, (d) Amers, (f) Table d'index ; valeurs non nulles en gris. (g) Graphe correspondant.

Cette organisation de données réduit considérablement l'espace mémoire de la matrice d'information. On ne stocke que les blocs non nuls de la partie triangulaire supérieure (ou inférieure) de la matrice d'information. Cependant, une telle organisation ne donne aucune idée sur la connectivité dans le graphe et le voisinage de chaque nœud. Pour pouvoir récupérer les voisnages et éviter la recherche de leurs emplacements, nous utilisons une table d'index à deux dimensions $(p+n) \times (p+n)$, où p est le nombre de poses robot et n est le nombre d'amers. Chaque élément de cette table d'index pointe vers le bloc de données (de la matrice d'information) dans le vecteur correspondant. Cette solution remplace la totalité des données associées à un bloc dans la matrice de représentation mémoire, illustrée dans la figure 3.1, par un simple pointeur. Dans le cas où les nœuds i et j ne sont pas reliés par une arête, l'entrée d'intersection correspondante est remplacée par un pointeur nul. Cette structure de donnée à l'avantage de permettre un accès direct aux données relatives au graphe et au système linéaire. D'autre part, la recherche d'un voisinage d'un nœud donné dans le graphe, revient tout simplement à parcourir la ligne ou la colonne correspondante dans la table d'index.

La table d'index est initialement remplie de valeurs nulles. Elle est ensuite mise à jour lors de l'insertion de nouveaux éléments de graphe. L'ajout d'un noeud ou d'une arête dans le graphe résulte en l'affectation du pointeur du bloc (dans la matrice d'information) à l'entrée correspondante dans la table d'index. Par ailleurs, la table d'index étant symétrique et contenant deux types de variables, on peut la partitionner en trois sous-matrices, à savoir H_{pp} , H_{ll} et H_{pl} . H_{pp} pointe vers le sous graphe composé de noeuds robots. H_{pl} indexe les arêtes d'observation. H_{ll} quant à elle, est une table unidimensionnelle et pointe vers les blocs diagonaux relatifs aux amers.

La table d'index présente des avantages, pour les petits et moyens graphes [Dine et al., 2014], quant à l'utilisation d'espace mémoire, l'accès direct et la recherche de voisnages. Toutefois, elle peut être très contraignante en termes d'espace mémoire si le graphe est très large. En effet, même si elle diminue considérablement l'espace mémoire alloué pour la matrice d'information, elle stocke toujours des valeurs nulles en mémoire. L'espace mémoire alloué pour la table d'index s'accroît, donc, quadratiquement avec l'avancement du robot dans l'environnement.

3.4 Structure de données compacte et incrémentale

La première représentation mémoire proposée, malgré ses avantages, présentait des limitations quant à la consommation d'espace mémoire, pour les graphes très larges, et l'adéquation à une mise à jour incrémentale. Partant de ce constat, nous avons proposé une deuxième structure de données que l'on va appeler LightSLAM ou LS (SLAM léger). Celle-ci a pour but de pallier aux problèmes de référencement des éléments et blocs non nuls, ainsi que de permettre une mise à jour incrémentale des structures d'index.

La nouvelle structure et organisation mémoire est différente de celles utilisées dans les implantations de l'état de l'art (g^2o , iSAM et IDS) comme suit :

- Elle fournit une organisation mémoire très compacte pour limiter la consommation d'espace mémoire.
- Elle fournit une méthode efficace pour garder trace des connexions dans le graphe sans pour autant utiliser de table d'index.
- Elle n'utilise pas de format de stockage compressé ni pour référencer les éléments non nuls dans le graphe (nœuds et arêtes) ni pour construire le système linéaire. Il est à noter, par contre, que nous avons utilisé la librairie CSparse qui ne prend pas de système par blocs en entrée. D'où, l'utilisation d'un format CCS, pour le solveur épars, s'avère inévitable. Tel est aussi le cas pour g^2o et iSAM. La construction du format CCS prend elle-même avantage de cette structure de données. L'algorithme de construction du format CCS sera détaillé plus loin dans ce chapitre.
- Elle est couplée à une mise à jour incrémentale, effectuée à chaque fois que l'on insère une nouvelle pose ou une nouvelle observation.
- Elle est optimisée pour bénéficier autant que possible de la mémoire cache au travers les blocs fonctionnels de l'algorithme.
- Elle est conçue pour servir de base pour les optimisations logicielles et matérielles telles que la parallélisation.

La figure 3.3 illustre la nouvelle organisation mémoire. Cette organisation peut être divisée en quatre parties relatives à :

- Structure du graphe (éléments du graphe)
- Connectivité
- Complément de Schur
- Système linéaire

3.4.1 Représentation des éléments du graphe

Nous reprenons, pour cette partie, la même idée que dans la première structure de donnée IDS. En effet, nous utilisons un vecteur pour chaque type d'élément de graphe. De même, une entrée des vecteurs stocke toutes les données en rapport avec l'élément en question telles que la position, les mesures capteur, l'erreur, etc. Par contre, les blocs de la matrice d'information ne sont plus mis dans ces vecteurs.

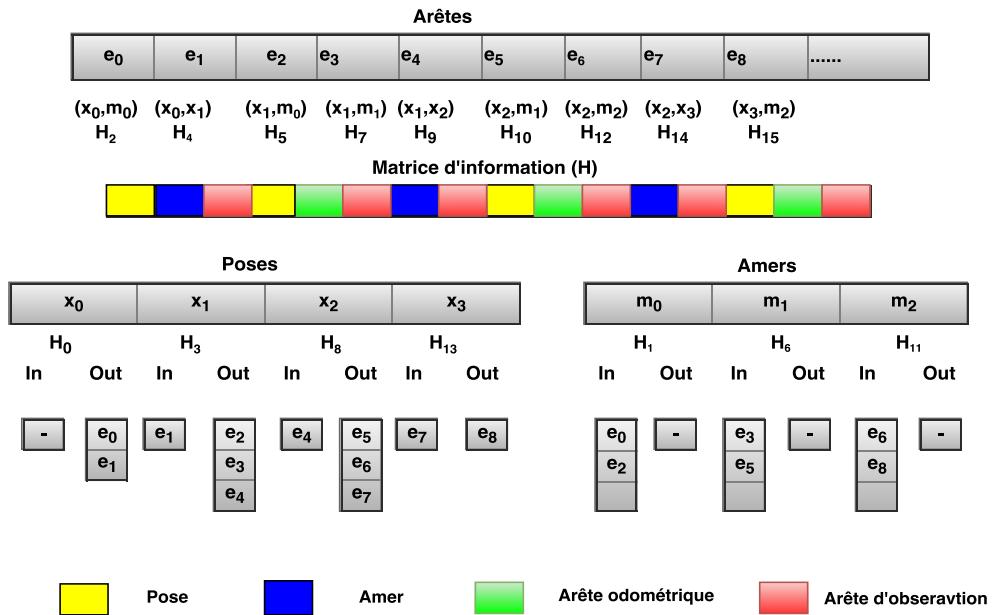


FIGURE 3.3 – Illustration de la structure de données de LS.

3.4.2 Connectivité : connexions entre nœuds

La représentation du graphe, seule, n'est pas suffisante. En effet, nous avons besoin de garder trace des connexions dans le graphe. Celles-ci permettent de retrouver les voisnages de chaque nœud. Les voisnages sont importants pour construire le système linéaire et offrir un grand degré de parallélisme de données. Afin de limiter l'usage de l'espace mémoire, notre idée consiste à éviter l'utilisation d'une matrice bidimensionnelle du moment que l'on a souvent affaire à des graphes épars.

Pour ce faire, indépendamment du type d'arêtes (mouvement ou observation), nous distinguons deux types : arête-entrantes et arêtes-sortantes. Une arête e_{ij} entre les noeuds i et j est à la fois une arête-sortante pour le noeud i , et une arête-entrant pour le noeud j . Tel que illustré dans la figure 3.3, nous référençons les arêtes par rapport aux noeuds. A chaque noeud, on associe deux listes qui lui indiquent son voisinage. La première liste contient les arêtes entrantes. La deuxième liste indexe les arêtes sortantes. Notons que l'indexation des arêtes permet également de retrouver les nœuds voisins.

L'illustration de la figure 3.3 montre la représentation mémoire d'un graphe à amers. Une arête-sortante d'un nœud robot est tout simplement une arête de mouvement. Les arêtes-entrantes peuvent inclure une arête de mouvement si le nœud robot n'est pas le dernier, et éventuellement un ensemble d'arêtes d'observation si le nœud est relié à des amers. Ceux-ci peuvent avoir uniquement des arêtes-entrantes.

Par ailleurs, la même organisation de données peut être appliquée dans le cas des graphes à pose, qui sont des graphes particuliers composés uniquement de noeuds robot. Dans ce cas, un nœud robot peut avoir plusieurs arêtes-sortantes et entrantes.

Cette manière de référencer les connexions limite considérablement l'occupation mémoire. Nous n'avons plus besoin de référencer l'absence d'arêtes entre nœuds comme dans IDS. La complexité

de stockage mémoire de LS est linéaire en nombre d’arêtes et de nœuds. D’autre part, l’utilisation de cette organisation mémoire permet de récupérer directement le voisinage d’un nœud donné dans le graphe. Cependant, le test de présence d’une arête entre deux nœuds i et j a un coût de $O(s)$, où s est le maximum entre le nombre d’arêtes sortantes de i et le nombre d’arêtes entrantes de j . Pour pallier à cela, les blocs fonctionnels peuvent être réécrits de manière à récupérer directement les voisnages sans effectuer de tests, et ainsi tirer profit de la structure de données. Le format CCS sera plus simple à construire du moment qu’une colonne dans la matrice d’information correspond à l’ensemble des arêtes entrantes et sortantes indexées à l’intérieur du nœud.

3.4.3 Complément de Schur

Le complément de Schur se ramène essentiellement à des produits de matrices et de vecteurs épars. Pour une meilleure performance, une implantation du complément de Schur devrait tirer avantage de la sparsité des opérandes.

Pour les multiplication matricielles éparses, de nombreux algorithmes et implantations existent dans la littérature [Bell and Garland, 2009, 2008]. Ces algorithmes utilisent des formats de stockage compressés. Il est à noter que g^2o utilise des formats de stockage compressés ainsi que la librairie Eigen pour effectuer ce type de calculs. Ces algorithmes restent généralistes. Il serait par contre très intéressant d’exploiter le caractère épars et incrémental, mais aussi la structure spécifique des matrices et vecteurs liés au GraphSLAM. Pour cela, nous nous appuyons toujours sur la structure de donnée incrémentale décrite dans la figure 3.3. Les blocs non nuls sont récupérés via les nœuds et leurs voisnages. Donc, nous n’utilisons pas de format de stockage compressés, pour le complément de Schur, tel que dans l’état de l’art.

Cependant, telle que la structure de données est décrite, elle ne donne aucune information à priori sur l’emplacement des blocs non nuls dans la matrice d’information résultante après marginalisation des amers. De ce fait, toutes les entrées de la matrice résultante doivent être calculées. Dans le cas où les matrices sont creuses, cela génère des calculs et des accès mémoire inutiles. Pour remédier à cela, nous étendons notre structure de données en ajoutant une table (table SchurNonZero) indexant tous les blocs non nuls de la nouvelle matrice d’information. Chaque élément de cette table contient les indices des deux nœuds liés (ou du noeud lui-même pour les blocs diagonaux).

La création de la table SchurNonZero s’effectue d’une manière incrémentale en même temps que la mise à jour de la structure du graphe. Nous décrivons dans la figure 3.4 les étapes les plus importantes à effectuer afin de mettre à jour la table SchurNonZero. Le graphe est initialement constitué d’une seule pose (non représentée dans l’illustration). A la première étape, la marginalisation des deux amers ne génère aucune nouvelle arête. Les indices de la pose sont insérés dans la table SchurNonZero. A l’étape 2, en plus des indices de x_1 , nous rajoutons $(0, 1)$ à la table SchurNonZero du fait de la présence d’une arête odométrique entre x_0 et x_1 . A l’étape 3, x_0 et x_2 ne sont pas reliés par une arête odométrique, mais ils ont un amer en commun. Par conséquent, une nouvelle arête est créée entre les deux nœuds. A l’étape 4, on applique la même procédure que sur les étapes précédentes. La table

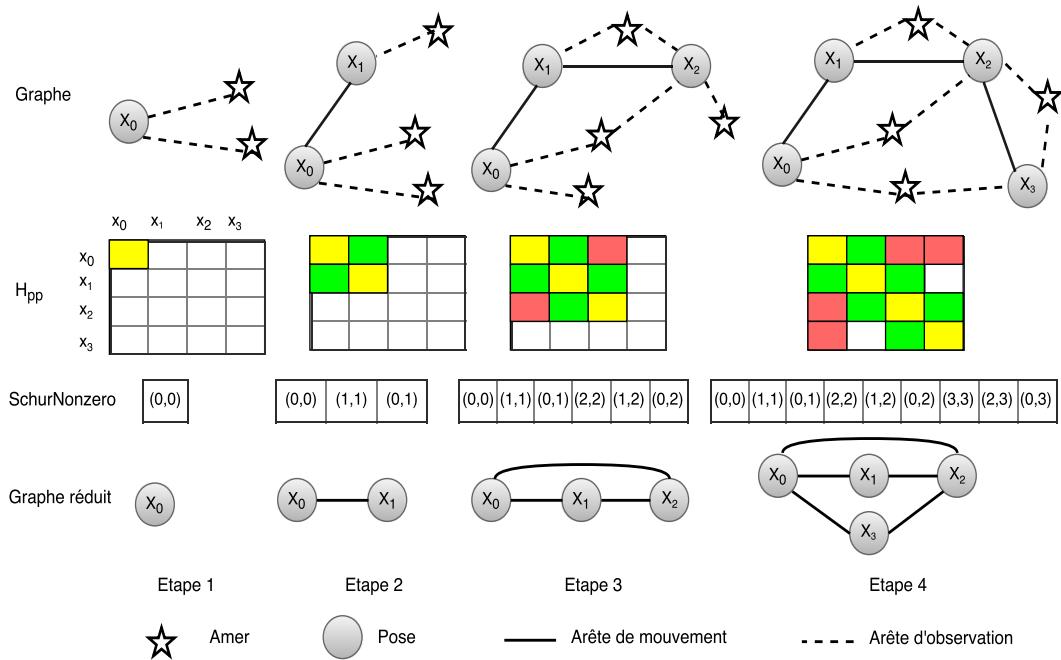


FIGURE 3.4 – Construction et mise à jour de la table SchurNonZero.

H_{pp} est uniquement donnée à titre illustratif et montre l'évolution de la matrice d'information suite à l'application du complément de Schur.

La table SchurNonZero et la capacité de récupérer les voisnages des nœuds donnent lieu à une nouvelle réécriture du complément de Schur. L'algorithme 2 présente les détails de cette nouvelle variante. Nous commençons le calcul par inverser la sous-matrice H_{ll} . Du moment que celle-ci est diagonale par blocs, nous avons simplement à inverser les blocs diagonaux. Nous calculons par la suite la nouvelle matrice d'information H_{pp} et le nouveau vecteur d'information b_p . Notons que H_{pp} et b_p peuvent être calculés en parallèle du moment qu'il n'y a pas de dépendance de données. La partie la plus importante en temps de calcul est H_{pp} . Avec la table SchurNonZero, nous ne nous intéressons plus qu'aux entrées qui seront non nulles dans la nouvelle matrice H_{pp} . Pour n'intégrer que les blocs non nuls dans les matrices originelles, nous exploitons les voisnages relatifs aux nœuds robots.

Pour représenter les connexions dans le graphe réduit, nous utilisons le même principe qu'avec le graphe original. Nous utilisons, cette fois-ci, pour chaque pose, deux listes indexant les nœuds adjacents : nœuds-prédécesseurs et nœuds-sucesseurs. La différence avec les listes d'arêtes-entrantes et sortantes est que, maintenant, on ne garde que les indices des nœuds voisins plutôt que les indices des arêtes. La mise à jour de ces deux listes s'effectue en même temps que la mise à jour de la structure du graphe. Notons que la construction du système s'appuie plutôt sur les listes des arêtes. L'utilisation des deux autres listes permet de garder simultanément trace du graphe original et du graphe réduit.

La table SchurNonZero ainsi que les voisnages, permettent de ne faire intervenir que les blocs non nuls dans le calcul. On va également éviter d'utiliser des formats de stockage compressés de H_{pp} et H_{pl} , car ceux-ci augmenteraient le temps de calcul. En effet, ces formats doivent être mis à jour à

chaque nouvelle optimisation de graphe car leur construction est basée sur une règle de précédence stricte (parcours par colonne de haut en bas).

Algorithme 2 : Complément de Schur optimisé

```

1 //  $H_{ll}$  inversing :
2 foreach landmark  $k \in Landmarks$  do
3   |  $Hll_{kk} \leftarrow Hll_{kk}^{-1};$ 
4 end
5 //  $b_p$  computing :
6 foreach pose  $i \in RobotPoses$  do
7   | foreach landmark  $k \in V_i$  do
8     | |  $bp_i \leftarrow bp_i - Hpl_{ik} Hll_{kk} bl_k;$ 
9   | end
10 end
11 //  $H_{pp}$  computing :
12 foreach edge  $(i, j) \in SchurNonzero$  do
13   | foreach landmark  $k \in V_i$  do
14     | | if  $exist(e_{jk})$  then
15       | | |  $Hpp_{ij} \leftarrow Hpp_{ij} - Hpl_{ik} Hll_{kk} Hlp_{kj};$ 
16     | | end
17   | end
18 end
```

3.4.4 Système linéaire

La matrice d'information étant symétrique, comme dans la première structure de données, nous ne stockons en mémoire que la partie triangulaire inférieure (ou supérieure). Chaque bloc non nul correspond à un nœud ou une arête dans le graphe. Il est donc intéressant de regrouper les blocs non nuls dans un vecteur dense, où chaque bloc peut être accessible à travers son nœud correspondant.

L'utilisation d'un solveur épars tel que CSparse requiert un format compressé de la matrice d'information. La construction de ce format peut être contraignante en temps d'exécution notamment si la matrice parcourue est très large. Pour réduire ce temps, nous exploitons la représentation mémoire des voisinages afin de ne tester que les blocs non nuls. L'algorithme 3 présente la procédure par laquelle le format CCS est construit. Si l'algorithme de construction originel consiste à parcourir la matrice d'information par colonne, le nouvel algorithme effectue un parcours par nœud. Cela revient au fait qu'une colonne correspond, en réalité, à un nœud dans le graphe. Les blocs non nuls sont donnés par les intersections avec les nœuds adjacents. Ceux-ci sont récupérés via les deux listes : arêtes-entrantes et arêtes-sortantes (comme expliqué précédemment). Pour respecter la règle de précédence dans la construction, on commence par les blocs associés aux arêtes entrantes, puis le nœud lui-même, et enfin les arêtes sortantes. Dans cet algorithme, les tests de valeurs nulles ne sont plus effectués qu'à l'intérieur des blocs non nuls.

Algorithme 3 : Construction du format CCS**Input :**

G : graph
 $InEdges_i$: list of in-edges of node i
 $outEdges_i$: list of out-edges of node i
 $(x \times y)$: degrees of freedom of node i

Output :

H_{CCS} : CCS format of the information matrix H

```

1 foreach node  $i \in G$  do
2   foreach degree  $k \in 1..y$  do
3     foreach neighbor node  $j \in InEdges_i \cup \{node_i\} \cup OutEdges_i$  do
4       foreach degree  $t \in 1..x$  do
5         | AddElement( $H_{CCS}, i, j, k, t$ ); //if not zero
6       end
7     end
8   end
9 end
```

3.4.5 Construction incrémentale du graphe

L’organisation mémoire proposée est couplée à une méthode de construction de graphe incrémentale. L’algorithme 4 résume les étapes les plus importantes à effectuer afin de mettre à jour la structure du graphe, et par conséquent la structure de données associée. La structure du graphe (nœuds et arêtes) est incrémentalement mise à jour au fur et à mesure que le robot navigue dans son environnement. En même temps, on met à jour l’organisation mémoire relative aux connexions, au complément de Schur, et au système linéaire. La construction incrémentale permet d’éviter la recherche de voisnages à chaque étape ou itération du solveur NLS. Cela pourrait être très coûteux en termes d’accès mémoire, notamment pour les graphes très larges. Il est à noter que la mise à jour des structures SchurNonZero, InNodes et OutNodes est relative à l’utilisation du complément de Schur. Pour tester si une arête, entre deux noeuds i et j a déjà été créée, on utilise astucieusement une petite table temporaire qui garde trace uniquement des arêtes relatives à l’étape en cours. Cela nous évite d’utiliser une matrice bidimensionnelle de la taille du nombre de poses telle que IDS.

3.5 Évaluations et résultats temporels

Nous présentons les évaluations temporelles des implantations basées sur les deux structures de données proposées. Les résultats seront comparés à deux implantations de l’état de l’art, à savoir g^2o et iSAM. Pour g^2o , nous avons utilisé la révision svn 54¹. Quant à iSAM, nous avons utilisé la version 1.7 de la révision svn 10².

1. <http://openslam.org/>
2. <https://svn.csail.mit.edu/isam/>

Algorithme 4 : Mise à jour incrémentale de la structure du graphe

Input : p_i : new robot pose at step i

$e_{(i-1)i}$: new odometry edge

V_i : set of visible landmarks at step i

$InEdges_i$: list of in-edges of node i

$outEdges_i$: list of out-edges of node i

$InNodes_i$: list of in-nodes of node i in the reduced-graph

$OutNodes_i$: list of out-nodes of node i in the reduced-graph

L : set of landmarks in the map.

Output : Updated graph data structure

```

1 Add  $P_i$  to the graph;
2 Insert the odometry edge  $e_{(i-1)i}$  into the graph;
3 Add  $(i, i)$  and  $(i - 1, i)$  to SchurNonzero;
4 Add the index of  $e_{(i-1)i}$  to  $OutEdges_{i-1}$ ;
5 Add the index of  $e_{(i-1)i}$  to  $InEdges_i$ ;
6 Add the index of node  $P_i$  to  $OutNodes_{i-1}$ ;
7 foreach landmark  $k \in V_i$  do
8   if  $k \notin L$  then
9     | Add the landmark  $k$  to  $L$ ;
10  end
11  foreach pose  $j \in InEdges_k$  do
12    if !exist( $e_{ij}$ ) then
13      | // if not already added;
14      | Add  $(i, j)$  to SchurNonzero;
15      | Add the index of node  $P_j$  to  $InNodes_i$ ;
16      | Add the index of node  $P_i$  to  $OutNodes_j$ ;
17    end
18  end
19  Insert the observation edge  $e_{ik}$  into the graph;
20  Add the index of  $e_{ik}$  to  $InEdges_k$ ;
21  Add the index of  $e_{ik}$  to  $OutEdges_i$ ;
22 end
23 Add the index of node  $(i - 1)$  to  $InNodes_i$ ;
```

3.5.1 Présentation des jeux de données

Pour l'évaluation des différentes implantations, nous avons utilisé des jeux de données communément utilisés par la communauté scientifique³. Figure 3.5 illustre la trajectoire et la carte, de chaque jeu de données, obtenues en utilisant notre implantation du GraphSLAM. Cet ensemble de jeu de données comporte quatre jeux de données simulées : Manhattan, 10k, City10k et CityTrees10k, et quatre jeux de données réelles : Intel, Freiburg, Freiburg University Hospital et Victoria Park.

Le tableau 3.1 présente la structure de graphe des jeux de données en termes de nombre de noeuds et d'arêtes. Deux types de graphes sont évalués, graphes à poses et graphes à amers. Le premier type contient uniquement des poses, tandis que les graphes à amers sont constitués de poses et d'amers. Ces jeux de données ont été choisis de telle sorte à avoir des graphes variés en termes de structure, mais aussi en termes de connectivité.

TABLE 3.1 – Structures de graphe des jeux de données.

Jeu de données	Poses	Amers	Arêtes
Intel	943	-	1837
Victoria Park	6969	151	10608
Freiburg 079	989	-	1217
Freiburg Hospital	1316	-	2820
CityTrees10k	10000	100	14442
Manhattan	3500	-	5453
10k	10000	-	64311
City10k	10000	-	20687

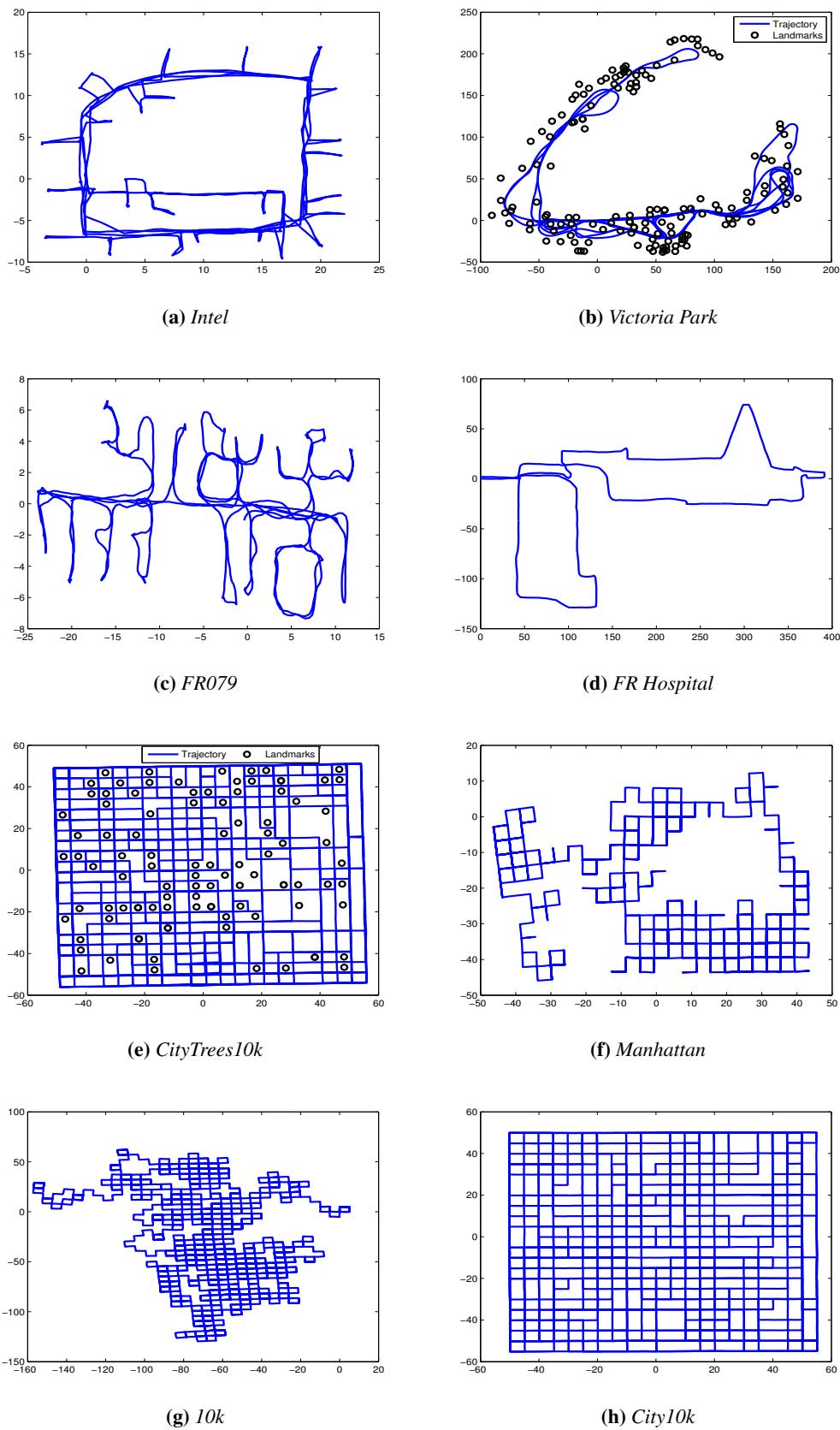
3.5.2 Modalités d'évaluation

Pour l'évaluation temporelle, nous considérons deux modes : batch et incrémental. Dans le mode batch, l'optimisation de graphe n'est effectuée qu'une fois tout le graphe est construit. Dans le mode incrémental, le graphe est incrémentalement optimisé. En d'autres termes, l'optimisation de graphe est effectuée toutes les x étapes. Dans cette étude, la période x est en général choisie pour des raisons d'évaluation temporelle.

D'autre part, il est à rappeler que le complément de Schur ne peut s'appliquer qu'aux graphes à amers. De plus, il n'améliore pas forcément le temps de calcul. En effet, il y a plusieurs paramètres d'influence, en particulier, le ratio du nombre d'amers par rapport au nombre de poses, la sparsité du nouveau système ainsi que l'efficacité de l'implantation du bloc fonctionnel lui-même. Pour cette raison, nous présentons, dans un premier lieu, les évaluations sans utiliser le complément de Schur. Nous détaillerons, dans un second lieu, l'effet du complément de Schur sur les performances. Cela permet d'avoir une visibilité plus objective concernant les performances des autres blocs fonctionnels.

G^2o et iSAM utilisent plusieurs algorithmes pour la résolution du NLS (GN, LM, DOG-LEG). Pour une comparaison équitable, toutes les implantations ont été configurées pour utiliser le processus

3. <https://sourceforge.net/projects/slam-plus-plus/files/data/>

**FIGURE 3.5 – Jeux de données utilisés dans l'évaluation.**

de Gauss-Newton avec un même nombre d’itérations par étape d’optimisation. De même pour les méthodes de résolution de systèmes linéaires, nous avons utilisé la librairie CSparse pour toutes les implantations. Par ailleurs, nous avons opté pour un calcul numérique des Jacobiennes afin de linéariser les erreurs correspondant aux arêtes.

Finalement, toutes les évaluations ont été effectuées sans parallélisation en utilisant un seul cœur (évaluation mono-cœur). Le but est d’évaluer l’impact de la structure de données et les optimisations liées à celle-ci sur les performances temporelles.

3.5.3 Résultats fonctionnels

Il est important de savoir que la réécriture des blocs fonctionnels et les optimisations mémoire ont été apportées de telle sorte à accélérer le traitement sans pour autant déprécier la précision de calcul sur la localisation et la cartographie. Pour évaluer formellement la précision de localisation et de cartographie, nous utilisons l’erreur euclidienne comme métrique. L’erreur euclidienne entre deux positions $p_0 = (x_1, y_1)$ et $p_1 = (x_2, y_2)$ est calculée comme suit :

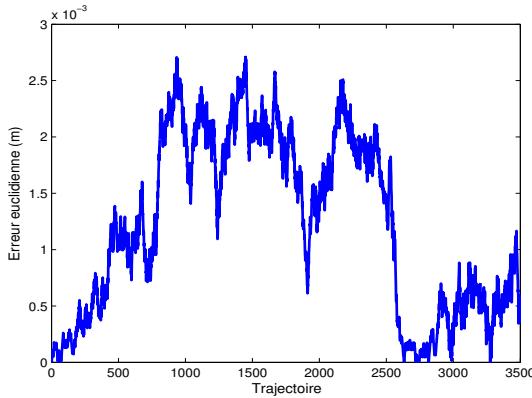
$$\text{Erreur euclidienne } (p_0, p_1) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (3.1)$$

Nous avons utilisé trois jeux de données simulées : Manhattan3500, City10k et CityTrees10k. Dans ces simulations, l’optimisation de graphe a été effectuée une fois tout le graphe construit. Le tableau 3.2 donne, pour chaque graphe, l’erreur euclidienne maximale et moyenne par rapport à la vérité de terrain. Du moment que les trois implantations (LS, g^2o , iSAM) utilisent globalement la même méthode, les erreurs sont comparables les unes aux autres. Notons que le nombre d’itérations de Gauss-Newton a été le même pour toutes les implantations. La petite différence en termes d’erreur euclidienne, entre g^2o et LS d’un côté et iSAM de l’autre côté, revient principalement à l’utilisation de constantes différentes dans le calcul numérique des Jacobiennes.

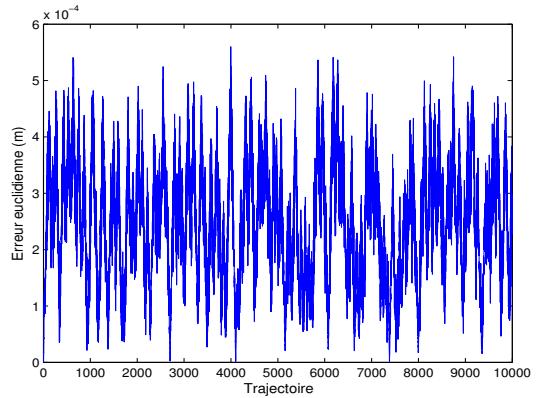
Par ailleurs, comme mentionné dans la chapitre 2, notre algorithme est basé plus particulièrement sur les travaux de [Grisetti et al., 2010]. Cette variante algorithmique est aussi utilisée dans g^2o . La figure 3.6 donne, pour chaque graphe, l’erreur euclidienne entre l’estimation donnée par LS et celle fournie par g^2o . Les figures 3.6a et 3.6b représentent l’erreur euclidienne sur les trajectoires correspondantes. Le graphe de CityTrees10K est un graphe à amers. La figure 3.6c donne l’erreur sur la trajectoire. L’erreur sur les amers est représentée sur la figure 3.6d. Nous constatons que l’erreur euclidienne maximale ne dépasse pas 1cm. Cela est négligeable par rapport à la taille de l’environnement. Cette légère différence est due aux arrondis de calcul dans les deux implantations.

TABLE 3.2 – Erreur euclidienne par rapport à la vérité du terrain.

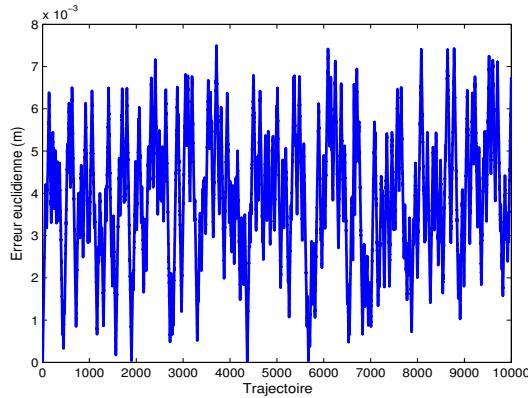
Erreur (m)	LS		g^2o		iSAM	
	Maximale	Moyenne	Maximale	Moyenne	Maximale	Moyenne
Manhattan	5.1757	1.0453	5.1670	1.0436	4.9588	1.0380
City10K	0.4118	0.1617	0.4116	0.1615	0.5896	0.2969
CityTrees10k	2.2703	1.1198	2.2769	1.1236	3.9952	1.9206



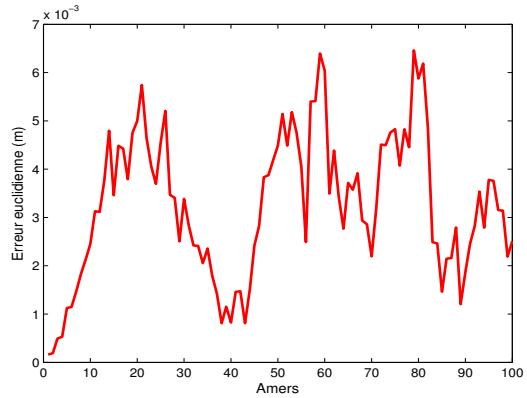
(a) Erreur euclidienne sur la trajectoire de Manhattan.



(b) Erreur euclidienne sur la trajectoire de City10k.



(c) Erreur euclidienne sur la trajectoire de CityTrees10k.



(d) Erreur euclidienne sur la carte de CityTrees10k.

FIGURE 3.6 – Erreurs euclidiennes : LS vs g^2o .

3.5.4 Présentation des architectures d'évaluation

Nous décrivons, dans ce qui suit, les architectures utilisées dans les évaluations de performances. Nous considérons deux niveaux de performances : un PC Intel et deux architectures embarquées, à savoir le Tegra K1 de la plateforme Jetson et l'Exynos 5422 de la plateforme ODROID XU4. Le tableau 3.3 donne les principales caractéristiques de ces architectures. Nous rappelons que dans ces évaluations, les processeurs graphiques ne sont pas utilisés.

TABLE 3.3 – Architectures d'évaluation.

Architecture	PC Intel	Tegra k1	Exynos 5422 (ODROID XU4)
Type	PC	Embarquée	Embarquée
CPU	Intel(R) i7-4700HQ	4 (Cortex-A15) + 1	4 (Cortex-A15) + 4 (Cortex-A7)
Fréquence CPU (GHz)	2.4	2.3	2 (Cortex-A15) et 1.4 (Cortex-A7)
Mémoire (Go)	6	1.7	2
Cache CPU (Mo)	6 (L3)	2 (L2)	2 (L2)

3.5.5 Métrique d'évaluation : Temps de Traitement par Élément de graphe (TPE)

Dans la majorité des travaux existant dans la littérature, nous avons constaté que “le temps de traitement par noeud” est utilisé comme référence de métrique d'évaluation et de comparaison [Kummerle et al., 2011, Polok et al., 2013a]. Or, d'un point de vue algorithmique, le temps de calcul dépend essentiellement du nombre d'arêtes. La construction du système linéaire s'effectue en traitant les arêtes du graphe une à une. De plus, une arête dans le graphe donne lieu à un bloc non nul dans le système linéaire. Il est clair qu'en utilisant la méthode de Cholesky, le temps de résolution est beaucoup plus dominé par le nombre de blocs non nuls que par les dimensions du système lui-même. Le nombre d'arêtes peut croître indépendamment du nombre de noeuds. Cela arrive quand le robot ré-observe les mêmes amers plusieurs fois au cours de la navigation. Cela est souvent le cas dans les problèmes où le nombre d'amers est largement supérieur au nombre de poses, comme dans la méthode d'ajustement de faisceaux [Triggs et al., 2000].

D'autre part, les optimisations algorithmiques et logicielles apportées dans ce travail ont pour but d'atténuer la complexité de calcul du GraphSLAM. Pour la majorité des blocs (sauf FB6), les transformations algorithmiques visent à garder le temps de traitement le plus linéaire possible en nombre d'éléments de graphe traités (arêtes ou noeuds). Pour ces raisons, dans ce travail, nous avons utilisé le temps de traitement par élément de graphe ou “Time Per graph Element” (TPE) comme métrique de comparaison. Le TPE est le temps nécessaire pour traiter un élément de graphe dans un bloc fonctionnel donné. Cela implique, en général, que le traitement se fait par noeud ou par arête. Pour les blocs FB7 et FB8, un élément de graphe est, en fait, une pose ou un amer. Le TPE est calculé par la formule suivante :

$$TPE = T/n \quad (3.2)$$

, où T est le temps global d'un ou plusieurs blocs fonctionnels, et n est le nombre d'éléments impliqués dans le calcul. Finalement, il est à noter que, pour les évaluations des temps totaux, nous utilisons le nombre d'arêtes plutôt que la somme du nombre d'arêtes et du nombre de noeuds.

3.5.6 Charges temporelles des blocs fonctionnels

La répartition du temps de calcul, sur les blocs fonctionnels, dépend de l'environnement exploré et de l'architecture de calcul utilisée. La représentation mémoire du graphe influence la manière dont les blocs fonctionnels sont implantés. Pour analyser la charge de chaque bloc fonctionnel, nous avons

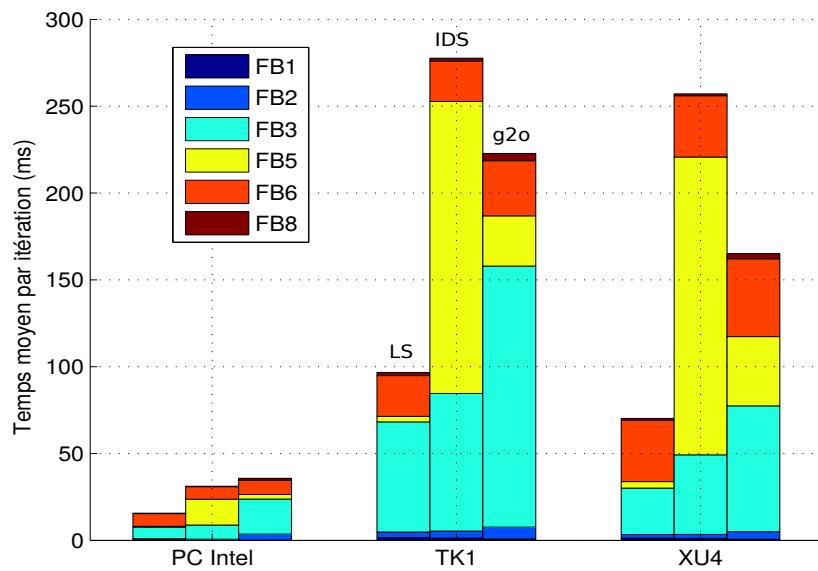


FIGURE 3.7 – Répartition des charges, sans complément de Schur, pour le graphe de Victoria Park.

évalué les trois implantations (LS, IDS et g^2o) sur le jeu de données de Victoria Park. Pour iSAM, vu sa construction incrémentale, il nous a été difficile de relever puis comparer les temps à chaque étape d'optimisation. Contrairement à iSAM, l'implantation de g^2o est assez proche de notre découpage fonctionnel. L'optimisation de graphe est effectuée toutes les 10 étapes. Le nombre d'itérations du processus Gauss-Newton a été fixé à une itération. Pour chaque itération, nous prélevons le temps de calcul de chaque bloc fonctionnel.

Le graphique de la figure 3.7 représente le temps d'optimisation de graphe moyen (697 itérations GN pour 6968 étapes), ainsi que les charges en termes de temps de calcul. Le complément de Schur n'est pas utilisé dans ce premier cas. On constate que la charge d'un bloc fonctionnel diffère d'une architecture à l'autre. En effet, pour LS, la construction du système linéaire (FB3) représente approximativement 45% du temps total sur le PC Intel et l'Exynos 5422. Elle a, par contre, une charge de plus de 60% sur le TK1. Par ailleurs, on remarque que la construction du format CCS (FB5) a une charge importante pour IDS et g^2o , particulièrement sur les architectures embarquées (TK1 et Exynos 5422). FB5 est caractérisé par un grand nombre d'opérations mémoire (chargement et rangement) avec très peu d'opérations arithmétiques. Il représente une charge moyenne de 60% pour IDS. D'autre part, la charge de FB5 est relativement négligeable dans LS. Cela est dû à la représentation mémoire des voisinages, mais également à la gestion incrémentale de cette représentation.

La figure 3.8 donne les temps moyens et les charges correspondantes sur chaque architecture dans la cas où l'on utilise le complément de Schur. On constate que le bloc de résolution (FB6) devient majoritairement le plus important pour nos implantations LS et IDS. Cela est expliqué par le fait que le complément de Schur peut rendre le graphe réduit moins épars. Le système linéaire résultant contient, ainsi, plus de blocs non nuls et pourrait nécessiter plus de temps pour le résoudre. Cela a une grande incidence sur la répartition des charges sur les blocs fonctionnels. Par ailleurs, l'algorithme

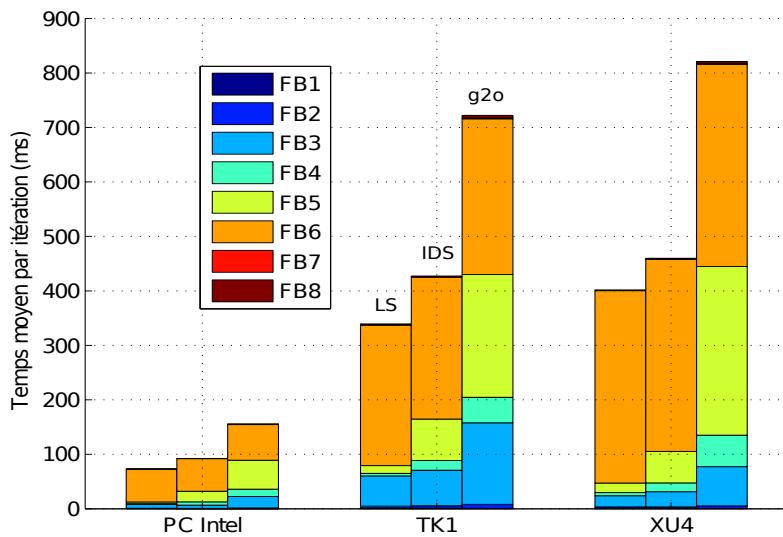


FIGURE 3.8 – Répartition des charges, avec complément de Schur, pour le graphe de Victoria Park.

optimisé du complément de Schur (FB4) dans LS est en moyenne 4 fois plus rapide que celui de IDS, et 8 fois plus rapide que celui de g^2o . Cela revient principalement au fait que, dans LS, en plus des voisinages des nœuds, les emplacements des blocs nuls de la nouvelle matrice sont connus à priori grâce à la construction incrémentale. Notons aussi que le temps de FB5 devient plus important dans g^2o du fait de la largeur du graphe.

En général, plus le graphe est dense, plus le temps de résolution du système linéaire (FB6) sera important par rapport aux autres blocs. L'utilisation du complément de Schur peut, au lieu de réduire, augmenter le temps de calcul si le graphe résultant devient dense après marginalisation des amers. Cela est aussi le cas pour les graphes à poses où la connectivité est plus dense. Le complément de Schur peut, cependant, améliorer le temps de calcul si le ratio *nombre d'amers/nombre de poses* est très grand. De ce fait, le complément de Schur s'avère très utile dans le cas de la méthode d'ajustement de faisceaux.

Enfin, pour un graphe à amers, ce qui est le cas le plus fréquent dans la pratique, la construction du système (y compris FB5) peut être très gourmande en temps de calcul ; soit en moyenne plus de 80% du temps global (IDS et g^2o). Moyennant une représentation mémoire efficace et des transformations algorithmiques, LS permet de réduire considérablement le temps de construction. Cependant, pour une meilleure performance, il est important de paralléliser la construction du système.

3.5.7 Évaluations temporelles sur un PC Intel

Il est intéressant de commencer par voir les performances de notre organisation mémoire sur une architecture classique type PC dont les ressources de calcul et de mémoire sont relativement élevées. Ce type d'architectures est le plus utilisé dans la littérature pour l'évaluation fonctionnelle ou temporelle

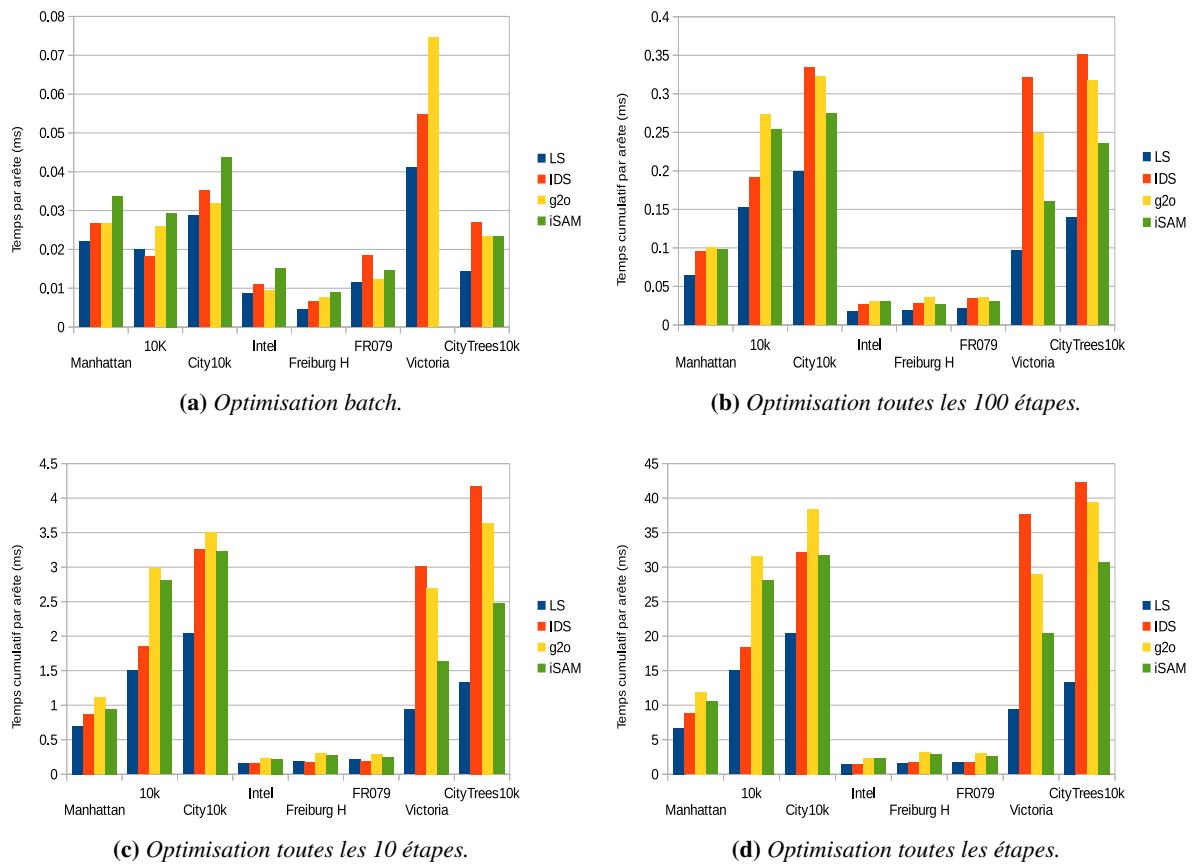


FIGURE 3.9 – Temps de traitement par arête sur le PC Intel.

des applications basées SLAM. Nous nous donnons ainsi une référence initiale de comparaison avec g^2o et iSAM en gardant le même type d’architectures utilisé dans leurs évaluations. Nous détaillons, dans ce qui suit, les résultats d’évaluation dans les deux modes, batch et incrémental.

a. Mode batch

Notons que dans ce mode, le graphe de Victoria Park n’a pas réussi à converger en utilisant iSAM. Cela est aussi souligné dans le travail de [Polok et al., 2013a]. Par ailleurs, pour avoir des temps comparables entre les jeux de données, nous utilisons la métrique TPE (Temps de traitement Par Élément) expliquée précédemment. Pour les temps globaux, nous avons opté pour le nombre total d’arêtes pour calculer le TPE.

Le tableau 3.4 présente les temps de calcul globaux en mode batch. La figure 3.9a donne les temps d’exécution par arête. Alors que g^2o est légèrement plus rapide que iSAM, IDS offre des TPEs comparables à g^2o sur les petits graphes, mais moins rapide sur les graphes larges. LS, quant à elle, surpassé toutes les implantations sur tous les graphes. LS réalise une accélération moyenne de 1.33 par rapport à IDS, 1.36 pour g^2o , et 1.57 pour iSAM. Ce gain est principalement dû à l’efficacité de notre implantation qui s’appuie sur la représentation mémoire proposée.

TABLE 3.4 – Temps d’optimisation en mode batch sur le PC Intel (s).

	Manhattan	10k	City10k	Intel	FR H	FR079	Victoria	CityTrees10K
g^2o	0.146	1.660	0.661	0.017	0.021	0.015	0.790	0.337
iSAM	0.183	1.889	0.903	0.027	0.025	0.017	NC	0.335
IDS	0.144	1.169	0.729	0.020	0.018	0.022	0.579	0.511
LS	0.123	1.281	0.595	0.016	0.013	0.014	0.436	0.208
Itérations	5	5	5	2	1	3	15	5

b. Mode incrémental

En mode incrémental, l’optimisation de graphe est réalisée toutes les étapes, 10 étapes et 100 étapes. Les périodes d’optimisation de graphe (1, 10, 100) sont choisies uniquement pour des raisons d’évaluation de performances temporelles. Pour converger, les graphes peuvent requérir moins d’optimisations de graphe ou d’itérations de GN que l’on utilise dans nos expérimentations. iSAM a été conçue pour mettre à jour incrémentalement le système linéaire, tandis que l’optimisation de graphe est faite périodiquement. Cela permet d’accélérer considérablement le traitement comparativement à g^2o . Mais, notons encore une fois que toutes les implantations ont été configurées pour avoir la même période d’optimisation de graphe. De même, le nombre d’itérations de GN a été fixé à 1 pour chaque étape.

Le tableau 3.5 donne le temps cumulatif de la totalité des étapes. Les figures 3.9b, 3.9c et 3.9d donnent les temps par arête (TPE). iSAM accomplit une légère accélération comparativement à g^2o sur les petits graphes à poses, et les graphes à amers. Cependant, sur les graphes relativement larges, à savoir 10k et City10k où la matrice d’information du système linéaire est plus dense, iSAM a montré un important ralentissement. Cela est dû à la large consommation d’espace mémoire quand on utilise CSparse. Par conséquent, pour ces deux graphes, nous avons utilisé la librairie CHOLMOD pour iSAM.

Par ailleurs, LS est toujours plus performante que IDS, g^2o et iSAM sur tous les graphes. De plus, le facteur d’accélération est plus important qu’il ne l’était en mode batch. Cette accélération est expliquée par le fait que notre structure de données est incrémentale. En effet, l’indexation de la connectivité dans le graphe, et ainsi dans le système linéaire s’effectue incrémentalement ; c-à-d à chaque étape. Il n’y a pas besoin de rechercher les blocs non nuls pour préparer à chaque étape les formats CCS et les tables d’index. Cela permet aussi un usage efficace de la mémoire cache. D’autre part, on constate que plus la période d’optimisation de graphe est petite, plus le gain apporté par LS est élevé. En effet, LS est jusqu’à 3 fois plus rapide que IDS et g^2o , et jusqu’à 2 fois plus rapide que iSAM, quand l’optimisation de graphe est appliquée à chaque étape. Notons que IDS donne relativement de bonnes performances par rapport à g^2o et iSAM sur les graphes à poses. Sur les graphes à amers, IDS subit un grand ralentissement à cause de l’initialisation et la gestion de la table d’index qui deviennent prohibitives quand le graphe est très large.

TABLE 3.5 – Temps de traitement total en mode incrémental sur le PC Intel.

	Manhattan	10k	City10k	Intel	FR H	FR079	Victoria	CityTrees10K
Optimisation de graphe toutes les étapes (s)								
g^2o	64.924	2035.154	794.643	4.346	8.820	3.632	308.013	570.021
iSAM	57.535	1807.782	656.332	4.145	7.975	3.160	216.929	444.322
IDS	47.951	1180.158	666.560	2.531	4.848	2.177	399.295	610.867
LS	36.483	964.863	422.775	2.677	4.459	2.032	99.351	193.364
Optimisation de graphe toutes les 10 étapes (s)								
g^2o	6.052	191.617	72.622	0.428	0.841	0.351	28.621	52.598
iSAM	5.155	180.621	64.953	0.404	0.762	0.307	17.385	35.715
IDS	4.742	119.045	67.314	0.281	0.503	0.229	31.967	60.282
LS	3.816	96.774	42.168	0.294	0.521	0.264	9.973	19.216
Optimisation de graphe toutes les 100 étapes (s)								
g^2o	0.541	17.559	6.665	0.055	0.101	0.044	2.639	4.584
iSAM	0.539	16.301	5.687	0.054	0.073	0.037	1.693	3.397
IDS	0.549	17.557	6.664	0.055	0.101	0.044	2.630	4.584
LS	0.350	9.818	4.121	0.033	0.054	0.026	1.034	2.021

3.5.8 Évaluations temporelles sur architectures embarquées

Les architectures embarquées et plus particulièrement les architectures destinées à la téléphonie mobile sont de plus en plus performantes. Dans la suite, nous allons présenter une évaluation temporelle de nos implantations, g^2o et iSAM sur le Tegra K1 et l'Exynos 5422. De la même manière que sur le PC Intel, toutes les expérimentations ont été effectuées en utilisant un seul cœur du processeur. Les coprocesseurs graphiques (GPU) ne sont pas non plus exploités. L'idée est toujours de quantifier le gain apporté par la représentation mémoire indépendamment de l'exécution parallèle qui sera abordée dans les chapitres suivants.

a. Évaluation sur le Tegra K1

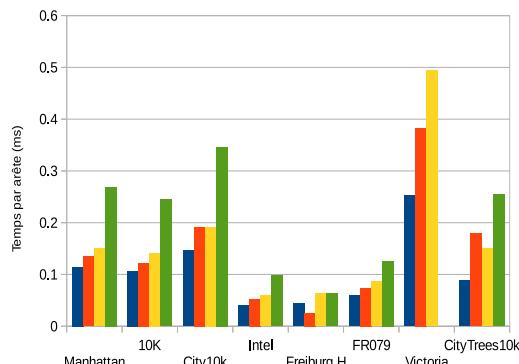
Le tableau 3.6 présente les temps d'exécution cumulatifs, sur le Tegra K1, dans les deux modes : batch et incrémental. La figure 3.10 donne les temps par arête. Dans le mode batch, nous constatons que le gain fourni par LS est plus important que celui obtenu sur le PC Intel en comparaison avec les autres implantations. Pour iSAM, on passe d'un facteur d'accélération de 1.57 sur le PC à un facteur moyen de 2.3 sur les architectures embarquées. De même, pour g^2o , LS est 1.5 fois plus rapide, contre 1.36 fois sur l'architecture PC. En revanche, on ne constate pas de différence en terme d'accélération entre LS et IDS. LS reste 1.29 fois plus rapide sur IDS.

Cette accélération est encore plus importante dans le mode incrémental. LS réalise une accélération moyenne de 2 fois comparativement à g^2o et iSAM, et 1.62 par rapport à IDS. LS est même jusqu'à 3 fois plus rapide que les autres implantations sur les graphes à amers. On remarque également que, comparée à IDS et g^2o , iSAM présente maintenant un important ralentissement dans le traitement vis-à-vis de l'évaluation sur PC. Les ressources limitées en mémoire, dans les architectures embarquées,

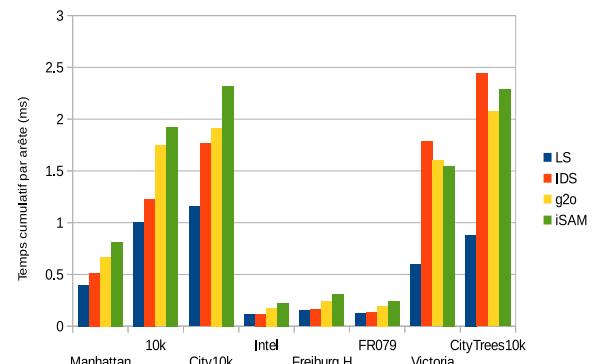
peuvent influencer dramatiquement le temps de calcul. En outre, iSAM utilise des librairies pour la résolution du système linéaire qui ne sont pas optimisées pour les architectures type ARM.

TABLE 3.6 – Temps de traitement total sur le TK1.

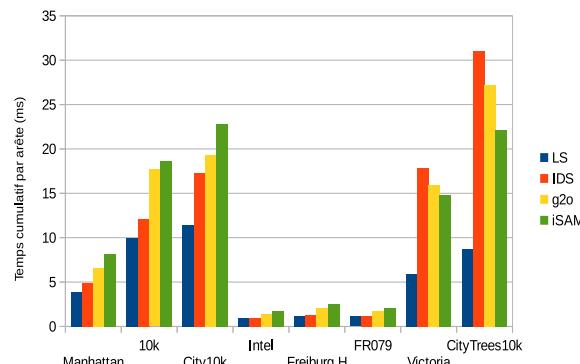
	Manhattan	10k	City10k	Intel	FR H	FR079	Victoria	CityTrees10k
Optimisation batch (s)								
g^2o	0.829	9.089	3.972	0.108	0.173	0.101	5.235	2.172
iSAM	1.467	15.760	7.154	0.187	0.224	0.155	NC	3.676
IDS	0.735	7.833	3.966	0.096	0.072	0.089	4.059	3.583
LS	0.628	6.881	3.025	0.073	0.122	0.078	2.687	1.284
Optimisation de graphe toutes les 10 étapes (s)								
g^2o	35.945	1137.575	399.647	2.631	5.901	2.179	168.820	392.37
iSAM	44.465	1235.631	472.168	3.225	7.117	2.514	156.545	319.719
IDS	26.621	779.750	358.738	1.738	3.762	1.422	189.908	448.139
LS	20.963	642.962	235.993	1.678	3.395	1.368	62.247	126.45
Optimisation de graphe toutes les 100 étapes (s)								
g^2o	3.654	112.116	39.622	0.313	0.681	0.152	16.981	29.924
iSAM	4.440	123.843	48.006	0.415	0.813	0.234	16.305	33.186
IDS	2.794	78.903	36.550	0.218	0.455	0.162	18.908	35.269
LS	2.162	64.852	23.871	0.212	0.439	0.297	6.331	12.735



(a) Optimisation batch.



(b) Optimisation de graphe toutes les 100 étapes (s).



(c) Optimisation de graphe toutes les 10 étapes (s).

FIGURE 3.10 – Temps de traitement par arête sur le TK1.

TABLE 3.7 – Temps de traitement total sur l’Exynos 5422.

	Manhattan	10k	City10k	Intel	FR H	FR079	Victoria	CityTrees10k
Optimisation batch (s)								
g^2o	0.565	5.823	3.091	0.072	0.064	0.066	1.426	3.126
iSAM	2.265	28.417	10.688	0.293	0.290	0.254	3.895	5.659
IDS	0.598	5.679	3.521	0.067	0.059	0.064	3.277	3.410
LS	0.487	4.906	2.904	0.051	0.037	0.048	2.092	1.018
Optimisation de graphe toutes les 10 étapes (s)								
g^2o	28.011	909.695	350.757	1.729	3.732	1.443	130.096	250.483
iSAM	66.509	1960.713	706.365	4.700	10.503	3.700	248.587	502.749
IDS	18.249	515.539	248.047	1.129	2.288	0.927	195.987	335.725
LS	15.337	448.719	198.063	0.962	1.979	0.755	46.604	97.155
Optimisation de graphe toutes les 100 étapes (s)								
g^2o	2.807	89.069	34.521	0.215	0.448	0.165	12.971	24.532
iSAM	6.946	218.673	72.637	0.617	1.315	0.447	25.749	51.621
IDS	1.953	52.245	25.409	0.155	0.293	0.116	16.552	32.169
LS	1.582	45.230	20.073	0.123	0.238	0.086	4.763	9.880

b. Évaluation sur l’Exynos 5422

Les résultats d’évaluation sur l’Exynos 5422 sont résumés dans le tableau 3.7. La figure 3.11 présente les temps par arête. On constate que les temps de LS, IDS et g^2o ont pratiquement les mêmes proportions entre eux que sur le Tegra K1. Les temps iSAM sont par contre très importants en comparaison avec le PC Intel et le Tegra K1. L’Exynos 5422 est équipé d’un quad-core ARM Cortex-A15 en big.LITTLE, couplé à un quad-core Cortex-A7. Nous soupçonnons que, pour iSAM, l’ordonnanceur sélectionne un processeur de faible consommation (LITTLE) Cortex-A7. Cela expliquerait l’augmentation significative en temps pour iSAM. Un travail d’investigation reste à faire pour expliquer ce comportement vis-à-vis de iSAM.

3.5.9 Complément de Schur

Le complément de Schur n’est appliqué qu’aux graphes à amers. La marginalisation des amers entraîne plus de remplissage dans la matrice d’information. Par conséquent, le complément de Schur peut augmenter le temps de calcul au lieu de le réduire. Les graphes de Victoria Park et CityTrees10k contiennent un nombre d’amers relativement petit par rapport au nombre d’arêtes d’observation et de poses. Cela donne des graphes réduits moins épars requérant plus de temps pour être optimisés. Les résultats obtenus en appliquant le complément de Schur sur ces deux graphes sont résumés dans le tableau 3.8 et la figure 3.12. En comparaison avec les optimisations de graphe sans complément de Schur, les temps de calcul ont considérablement augmenté à cause du remplissage engendré par la marginalisation des amers.

Par ailleurs, LS offre toujours la meilleure performance. Cependant, on constate une dégradation global du gain par rapport à g^2o et IDS. Cela revient au fait que la résolution du système linéaire (FB6) domine largement le temps de calcul à cause du phénomène de remplissage.

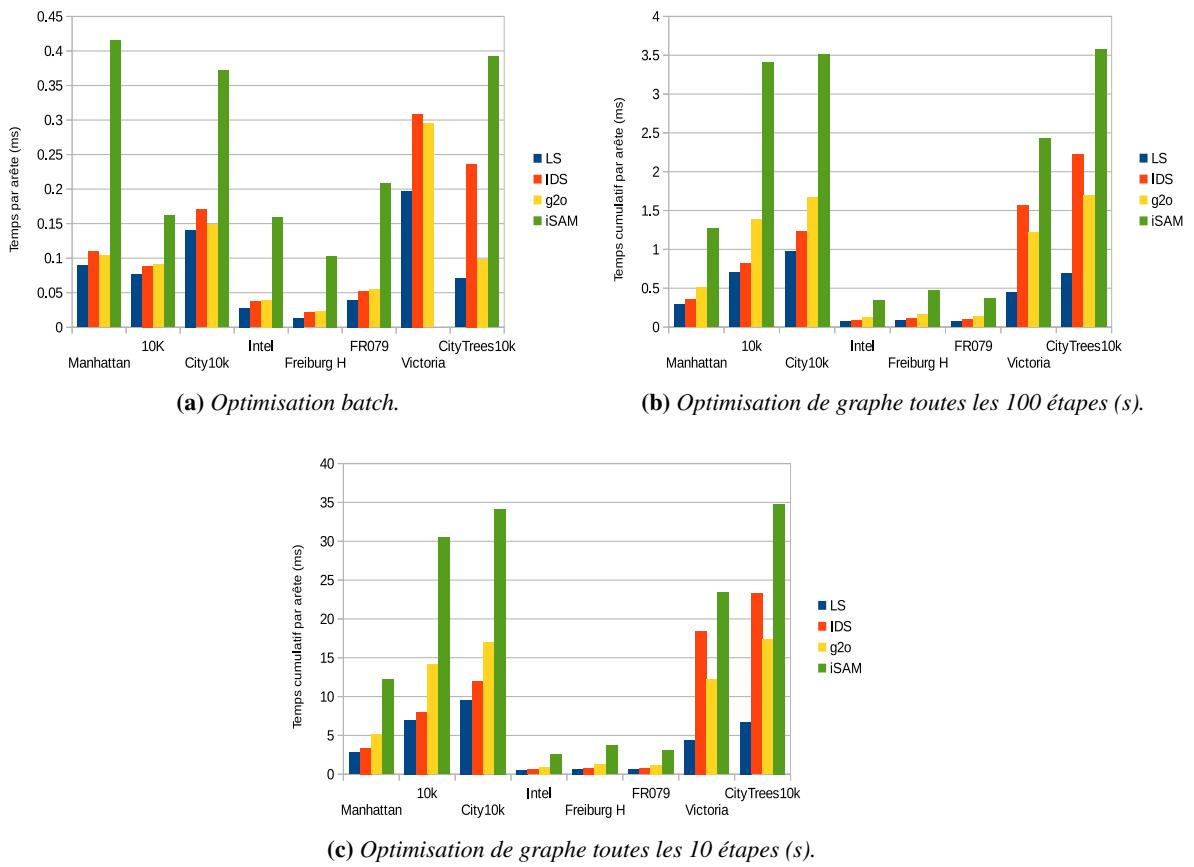


FIGURE 3.11 – Temps de traitement par arête sur l'Exynos 5422.

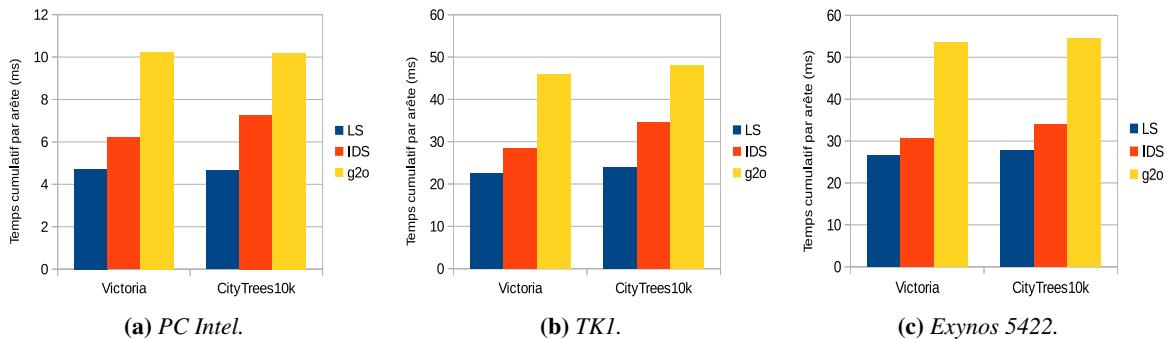


FIGURE 3.12 – Optimisation de graphe, avec complément de Schur, toutes 10 les étapes (s).

3.5.10 Évolution du temps de traitement : TPE

Le temps d'optimisation de graphe dépend du nombre d'arêtes et leur répartition dans le graphe, et donc dans la matrice d'information. Indépendamment des dimensions du système linéaire, et par rapport aux explications données dans le chapitre précédent, le temps de résolution est étroitement lié à la répartition des blocs non nuls dans la matrice d'information. Dans cette section, nous analysons

TABLE 3.8 – Évaluation avec complément de Schur.

Intel		TK1		Exynos 5422	
Victoria	CityTress10k	Victoria	CityTress10k	Victoria	CityTress10k
Optimisation en mode batch (s)					
g^2o	3.326	1.220	16.649	6.611	17.9111
IDS	3.675	1.334	15.916	6.454	24.193
LS	3.166	1.073	14.627	5.725	18.041
Optimisation de graphe toutes les 10 étapes (s)					
g^2o	108.584	147.115	486.697	694.136	569.631
IDS	65.898	104.854	302.805	500.154	324.126
LS	50.148	67.245	238.850	347.253	281.905
Optimisation de graphe toutes les 100 étapes (s)					
g^2o	10.379	6.824	48.444	69.134	56.586
IDS	6.945	10.648	32.052	54.239	33.172
LS	5.106	14.664	24.803	35.602	28.829

l'évolution du TPE en mode incrémental. Idéalement, le temps de traitement d'une arête ne doit pas croître. En d'autres termes, il ne doit pas être impacté par le nombre d'arêtes, de nœuds ou de la connectivité dans le graphe. Les figures 3.13 et 3.14 illustrent respectivement l'évolution du TPE sans et avec complément de Schur. Nous avons choisi les jeux de données de Victoria Park et City10k pour le cas sans complément de Schur, et Victoria Park et CityTrees10k dans le second cas (avec complément de Schur). La librairie iSAM n'utilise pas le complément de Schur. D'où, les évaluations du complément de Schur concernent uniquement trois implantations, à savoir LS, IDS et g^2o .

Les évaluations montrent que les différences en termes de performances entre les implantations sont variables d'une architecture à l'autre. En effet, on remarque notamment une dépréciation des performances de iSAM sur le Tegra K1 et l'Exynos 5422, en comparaison avec les autres implantations. En optimisant le graphe toutes les 10 étapes et plus particulièrement toutes les étapes, nous avons constaté une large consommation d'espace mémoire par iSAM. Cela se répercute sur les performances globales.

a. Sans complément de Schur

Pour le graphe de Victoria Park, la construction du système et sa résolution ont approximativement la même importance en termes de temps de calcul. On constate que IDS donne, au début, de meilleures performances par rapport à g^2o . Cependant, plus le graphe devient large, plus le TPE de IDS augmente avec un comportement quasi-linéaire. Par conséquent, au bout d'un certain nombre d'étapes, IDS subit un important ralentissement en comparaison avec g^2o . Cela est expliqué par le fait que dans le cas de graphes à amers, IDS initialise puis parcourt toute la table d'index pour construire le système linéaire et le format CCS. D'autre part, l'accès à la table d'index est accompagné par l'accès à d'autres zones mémoire. Cela rend le principe de localité spatiale et temporelle difficile à respecter, notamment quand le graphe est très large. Cela dépend aussi de l'architecture utilisée. IDS est plus rapidement dépassée par g^2o sur les architectures embarquées que sur le PC Intel qui dispose d'une hiérarchie

mémoire plus performante. Par ailleurs, LS fournit le plus faible TPE sur toutes les architectures. Le TPE de LS tend à être relativement stable malgré le fait que le nombre d'arêtes et de nœuds augmente avec le temps.

Le graphe de City10k est moins épars que celui de Victoria. De ce fait, la résolution du système représente le bloc le plus consommateur en temps de calcul. Cela impacte le comportement du TPE. En effet, pour toutes les implantations et sur toutes les architectures, le TPE croît avec le temps. Cependant, LS donne toujours le meilleur TPE par rapport aux autres implantations et le gain augmente avec le temps.

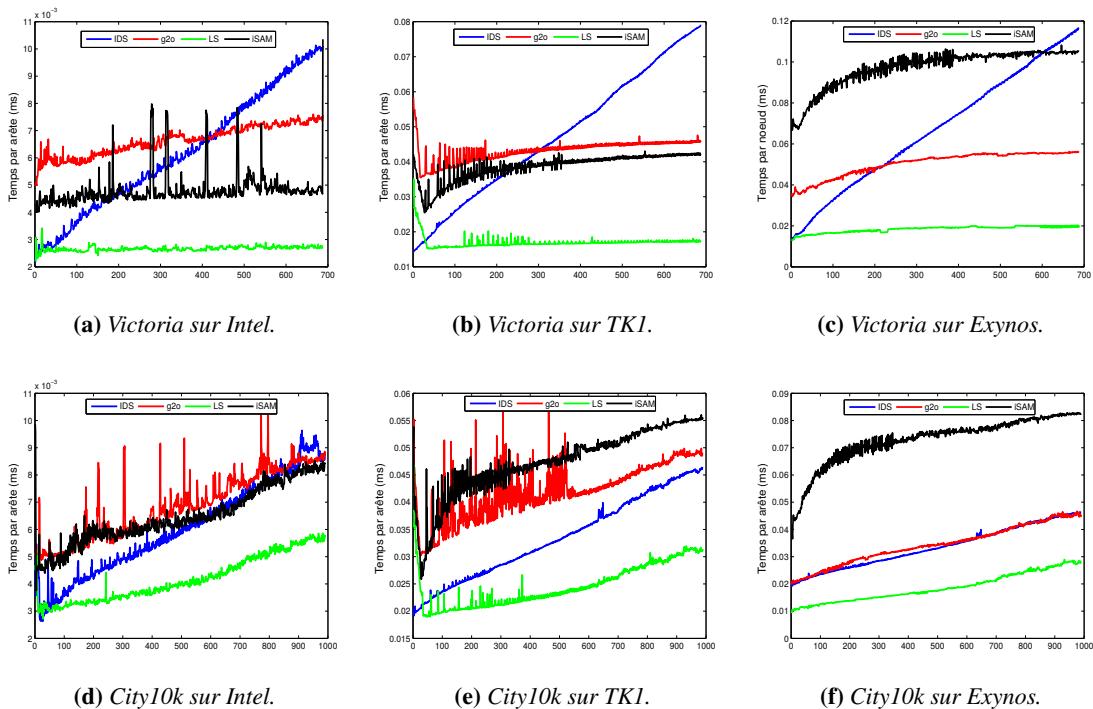


FIGURE 3.13 – Évolution du TPE sans complément de Schur.

b. Avec complément de Schur

La marginalisation des amers des graphes de Victoria Park et de CityTress10k résulte en des graphes moins épars et comportant plus d'arêtes. Par conséquent, le temps d'optimisation de graphe, dans ce cas, est dominé par le temps de résolution du système linéaire. Ainsi, plus le graphe va grandir, et plus les TPEs des implantations augmentent. Le comportement des TPEs est similaire sur toutes les architectures où LS est la plus rapide. Notons que le gain de LS par rapport à IDS est moins important du fait de l'importance du bloc de résolution.

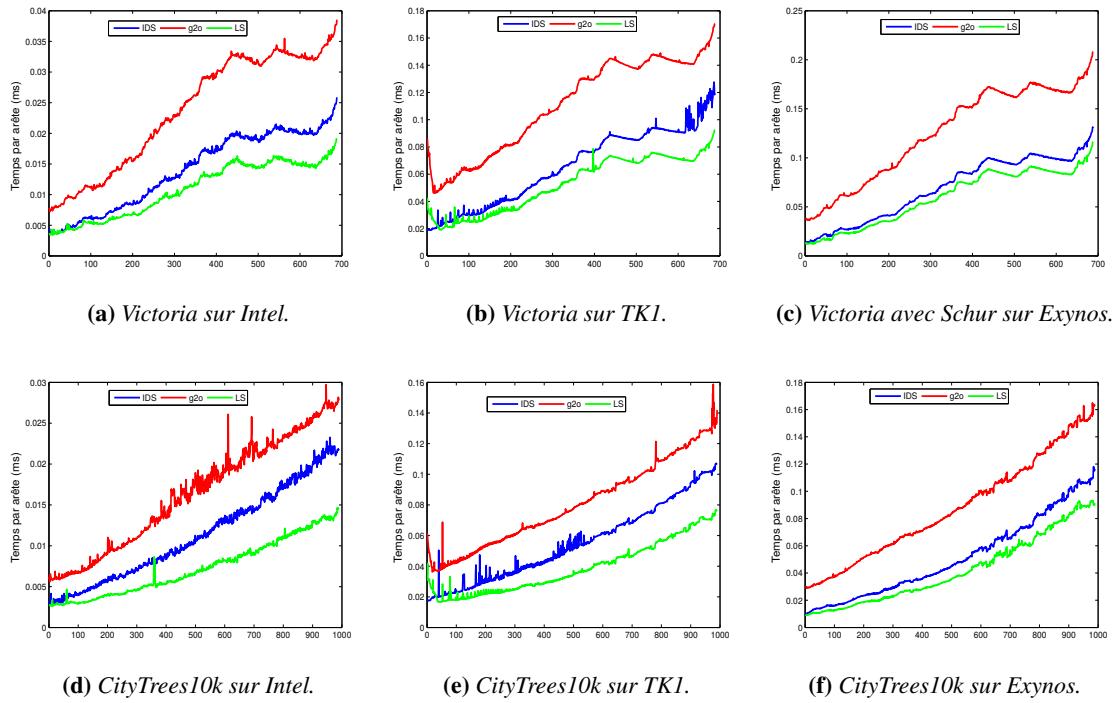


FIGURE 3.14 – Évolution du TPE avec complément de Schur.

3.6 Conclusion

Dans ce chapitre, nous avons présenté deux représentations mémoire pour implanter les problèmes de moindres carrées pouvant être modélisés à l'aide d'un graphe. Le GraphSLAM représente le problème de SLAM au moyen d'un graphe dont les nœuds modélisent la trajectoire et la carte de l'environnement. Du fait de sa modélisation graphique et la caractéristique de lissage de la méthode elle-même, la représentation mémoire du problème joue un rôle déterminant dans les performances temporelles.

Ce travail a donné lieu à deux représentations mémoire : IDS et LS. La première offre un compromis entre l'accès direct aux données et l'espace mémoire. Cependant, IDS ne peut être utilisée pour les très grands graphes impliquant un nombre important de nœud (plus de 4000 nœuds). Pour pallier à cela, nous avons proposé la deuxième représentation mémoire, à savoir LS. Cette dernière fournit une structure de données très compacte. D'où le nom Light SLAM (SLAM léger). La complexité de stockage mémoire de LS est linéaire en nombre d'arêtes et de nœuds. Cette complexité est quasiment linéaire. LS exploite le caractère épars et incrémental du problème de SLAM. La mise à jour de la représentation mémoire s'effectue d'une manière incrémentale au moment de la mise à jour de la structure du graphe grâce aux données capteurs.

Les évaluations ont montré l'efficacité de LS dans les deux modes d'optimisation de graphe : batch et incrémental. Le gain par rapport aux implantations de l'état de l'art est encore plus important en incrémental. LS est jusqu'à trois fois plus rapide que les implémentations de l'état de l'art (g^2o et iSAM). Cela revient principalement à la mise à jour incrémentale qui permet d'éviter de rechercher les voisnages des nœuds à chaque étape ou itération du NLS.

Finalement, la représentation mémoire LS vise à réduire la complexité de calcul du GraphSLAM dans l'optique de porter l'algorithme sur les architectures hétérogènes embarquées. De ce fait, nous avons opté pour LS comme implantation de base afin de porter l'algorithme sur ce type d'architectures.

Chapitre 4

Étude et implantation sur architectures hétérogènes à base de GPU

Sommaire

4.1	Introduction	93
4.2	Parallélisation de l'algorithme	94
4.2.1	Calcul des erreurs	94
4.2.2	Construction du système linéaire	94
4.2.3	Complément de Schur	97
4.2.4	Résolution du système linéaire	98
4.2.5	Calcul des incrémentations des amers	98
4.3	Effet de la fonctionnalité zéro-copie	98
4.4	Adéquation Algorithme-Architecture	101
4.4.1	Scalabilité	102
4.4.2	Analyse de l'évolution du temps par arête	103
4.4.3	Discussion et partitionnement CPU/GPU	105
4.4.4	Architecture mémoire	110
4.5	Résultats expérimentaux	111
4.5.1	Répartition des charges temporelles	111
4.5.2	Optimisation de graphe en mode batch	113
4.5.3	Optimisation de graphe en mode incrémental	114
4.6	Conclusion	116

4.1 Introduction

Les systèmes orientés applications pour la robotique sont au cœur de l'attention de la communauté scientifique actuelle. L'aspect embarquabilité prend toute son importance, tout en faisant évoluer les algorithmes. De par leurs calculateurs parallèles et hétérogènes, les architectures hétérogènes embarquées offrent une grande souplesse d'implantation allant de la gestion des flux de données capteurs jusqu'aux calculs intensifs. Parmi ces architectures, on identifiera les architectures hétérogènes à base de FPGA (Cyclone 5, Zynq-7000) et celles à base de GPU (Tegra K1, Exynos 5422). Malgré une grande puissance de calcul intrinsèque, l'embarquabilité sous différentes contraintes temps-réel, reste difficile à cause de la complexité des algorithmes. Cela nécessite également une expertise pour optimiser l'exécution de ces algorithmes sur des architectures disposant de structures matérielles complexes (de communications et de calcul). D'autre part, l'évolution de la masse de données au fil du temps dans un algorithme ne simplifie pas son implantation. L'évolution de la masse de données dans le GraphSLAM en est un exemple. Par conséquent, l'amélioration des performances temporelles, sur les architectures embarquées, passe par une analyse conjointe de l'algorithme et de l'architecture impliquant une approche d'adéquation algorithme-architecture.

Nous présentons, dans ce chapitre, une étude d'implantation du GraphSLAM sur architectures hétérogènes à base de GPU. L'étude a pour objectif de proposer une implantation adéquate du GraphSLAM tout en explorant les performances de ce type d'architectures. Le choix de l'architecture est justifié par le fait que ce type d'algorithmes sont consommateurs de puissance de calcul, mais surtout de puissance de transfert de données. L'algorithme se découpe en blocs fonctionnels explorant et transformant un espace mémoire important. Notre approche consiste en la répartition des blocs fonctionnels sur les différents processeurs disponibles de telle manière à réduire le temps de calcul global. Cette étude investigue particulièrement la puissance de calcul du récent système sur puce (SoC), à savoir le Tegra K1 (TK1)¹.

Plusieurs travaux de recherche ont déjà utilisé des processeurs graphiques pour accélérer des applications de robotique tels que [Choudhary et al., 2010, Wu et al., 2011, Rodriguez-Losada et al., 2013, Ratter et al., 2013]. Les auteurs ont utilisé des GPUs pour implanter des algorithmes de reconstruction d'environnements. Certaines tâches de ces algorithmes sont plus ou moins similaires à celles présentes dans le GraphSLAM. En termes d'architectures, ces travaux utilisent des GPUs standards (de bureau) pour leurs systèmes de SLAM.

Nous présentons, dans un premier temps, les transformations algorithmiques nécessaires pour porter certains blocs fonctionnels sur des architectures parallèles. Nous ferons, dans un second temps, l'étude d'une implantation du GraphSLAM sur une architecture hétérogène à base de GPU. Nous avons utilisé la plateforme Jetson de Nvidia (Fig. 4.1) qui intègre un système-sur-puce Tegra K1.

1. Une description de cette architecture est donnée dans la section 2.2.1.



FIGURE 4.1 – Plateforme Jetson de Nvidia.

4.2 Parallélisation de l'algorithme

L'implantation d'un algorithme sur une architecture parallèle nécessite un travail de parallélisation de ce dernier en vue d'accélérer efficacement le traitement. Il s'agit plus précisément de réécrire certains blocs fonctionnels (Fig. 1.6) de manière à réduire ou éliminer les dépendances de données, pour assurer un haut degré de parallélisme. Nous rappelons qu'un bloc fonctionnel est défini comme un ensemble d'instructions destinées à accomplir, en un temps fini, un traitement sur les données de l'algorithme.

Dans la suite de cette section, nous présenterons les différentes réécritures et transformations algorithmiques nécessaires pour porter efficacement le GraphSLAM sur une architecture parallèle. Nous mettrons tout d'abord en évidence certains points concernant des traitements particuliers.

4.2.1 Calcul des erreurs

Le calcul d'une erreur e_{ij} n'induit aucune dépendance de données avec le calcul de l'erreur e_{jk} . Ainsi, le calcul des erreurs peut être parallélisé et ne nécessite pas de mécanisme de synchronisation lors de son exécution.

4.2.2 Construction du système linéaire

L'algorithme 5 illustre la construction du système linéaire (matrice et vecteur d'information). A_{ij} et B_{ij} désignent les Jacobiennes de la fonction d'erreur appliquée à l'arête e_{ij} . Elles désignent respectivement les dérivées de e_{ij} par rapport aux noeud i et j . La linéarisation d'une arête e_{ij} met à jour les blocs $H(i,i)$, $H(j,j)$, $H(i,j)$ et $H(j,i)$ de la matrice d'information H , mais également les blocs b_i et b_j du vecteur d'information b . Chaque bloc diagonal $H(i,i)$ (ou b_i) correspond, en réalité, à un nœud dans le graphe, qui peut être aussi bien une pose qu'un amer. Un nœud i peut être une extrémité commune à plusieurs arêtes. Dans un graphe à amers, cela est le cas quand un même amer a été

Algorithme 5 : Construction du système linéaire (FB3)

```

1 // Jacobians computing :
2 foreach edge  $e_{ij} \in Edges$  do
3    $A_{ij} \leftarrow \frac{\partial e_{ij}(y_i, y_j)}{\partial y_i} \Big|_{y_i=\hat{y}_i, y_j=\hat{y}_j};$ 
4    $B_{ij} \leftarrow \frac{\partial e_{ij}(y_i, y_j)}{\partial y_j} \Big|_{y_i=\hat{y}_i, y_j=\hat{y}_j};$ 
5 end
6 // Information matrix and vector computing
7 foreach edge  $e_{ij} \in Edges$  do
8    $H_{ii} \leftarrow H_{ii} + H_i; \quad // H_i = A_{ij}^T \Omega_{ij} A_{ij};$ 
9    $H_{ij} \leftarrow H_{ij} + A_{ij}^T \Omega_{ij} B_{ij};$ 
10   $H_{jj} \leftarrow H_{jj} + H_j; \quad // H_j = B_{ij}^T \Omega_{ij} B_{ij};$ 
11   $b_i \leftarrow b_i + s_i; \quad // s_i = A_{ij}^T \Omega_{ij} err_{ij};$ 
12   $b_j \leftarrow b_j + s_j; \quad // s_j = B_{ij}^T \Omega_{ij} err_{ij};$ 
13 end

```

observé depuis plusieurs positions au cours de la navigation. De même, depuis une même position, le robot peut observer plusieurs amers. Dans ce cas, le bloc $H(i, i)$ est mis à jour à chaque fois que l'on linéarise une arête dont l'une des deux extrémités est le nœud i . Tel que illustré dans la figure 4.2, la linéarisation de chacune des arêtes e_{ij} , e_{ik} et e_{il} génère quatre termes quadratiques. Par exemple, e_{ij} produit H_i , H_j , s_i et s_j . Chaque terme est donc ajouté au bloc correspondant. Dans l'illustration, les blocs $H(m_i, m_i)$ et b_{m_i} reçoivent, chacun, trois termes suite à la linéarisation des arêtes adjacentes.

Pour un traitement parallèle des arêtes, il est évident que l'on a besoin d'un mécanisme de synchronisation pour éviter les accès simultanés aux blocs communs entre arêtes (H_{ii} , H_{jj} , b_i , b_j). Cependant, l'utilisation d'un mécanisme de synchronisation pourrait ralentir l'exécution, particulièrement sur GPU, si les degrés des nœuds sont relativement grands. Le degré d'un nœud dans un graphe représente le nombre de nœuds qui lui sont adjacents. Un thread traitant l'arête e_{ij} et voulant mettre à jour le bloc $H(j, j)$ doit impérativement attendre la fin du traitement de l'arête e_{jk} si celle-ci est en cours de traitement par un autre thread. Sur un GPU, un tel mécanisme de synchronisation casse le parallélisme SIMD.

Pour pallier au problème de dépendance de données et de synchronisation dans la construction du système, nous avons réécrit l'algorithme de calcul de la matrice et vecteur d'information (ligne 7 à 12 de l'algorithme 5). Ces réécritures sont détaillées dans l'algorithme 6. On commence, tout d'abord, par le calcul des termes (H_i , H_j , s_i , s_j) générés par la linéarisation des erreurs. La mise à jour de H_{ij} est effectuée une seule fois, et ce par un seul thread. Nous pouvons, donc, mettre à jour en même temps les blocs hors-diagonaux de la matrice d'information. L'idée du nouvel algorithme est de remplir les blocs diagonaux (correspondant aux poses robot et amers) par nœud au lieu de le faire par arête comme illustré dans la figure 4.3. Un nœud est, maintenant, traité par un et un seul thread. De cette manière, aucun mécanisme de synchronisation n'est requis. Un bloc correspondant à un nœud donné est mis à jour moyennant les termes H_i , H_j , s_i et s_j pré-calculés. Notons que les termes H_j et s_j émanant de la linéarisation d'une arête odométrique sont respectivement rajoutés à H_{ii} et b_i .

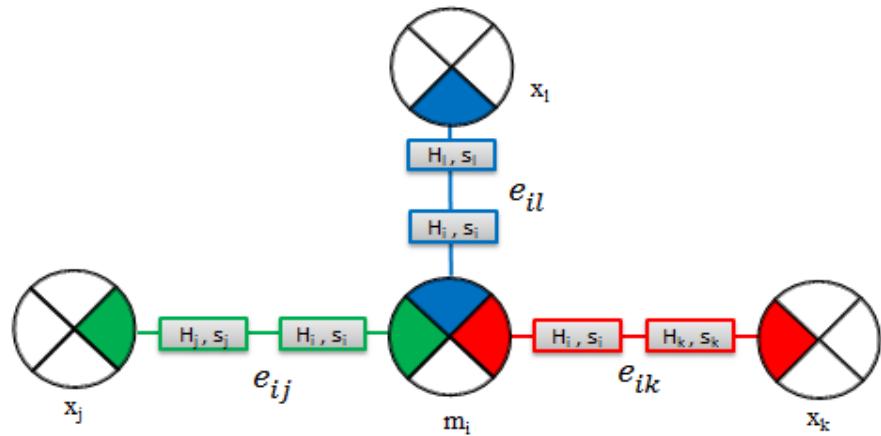


FIGURE 4.2 – Modification concorrente de $H(m_i, m_i)$ et b_{m_i} par 3 threads, pour la mise à jour par arête de la matrice et vecteur d'information. Les données modifiées par un thread sont distinguées par une couleur différente.

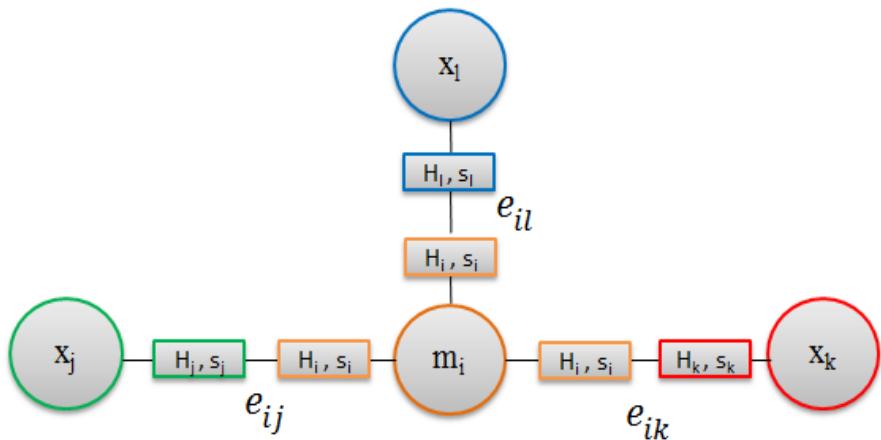


FIGURE 4.3 – Mise à jour par nœud de la matrice et vecteur d'information. Dans l'illustration, 4 threads sont utilisés. Un thread par nœud. Aucun mécanisme de synchronisation n'est requis. Les données modifiées par un thread sont distinguées par une couleur différente.

Pour une implantation GPU, nous utilisons trois kernels. Le premier calcule les termes H_i , H_j , s_i et s_j en mettant, en même temps, à jour les blocs hors-diagonaux de la matrice d'information. Les deux autres kernels sont en charge de la mise à jour des blocs correspondant aux nœuds.

Algorithme 6 : Construction parallèle du système linéaire

```

1 // End-add terms computing
2 foreach edge  $e_{ij} \in Edges$  do
3    $H_i = A_{ij}^T \Omega_{ij} A_{ij};$ 
4    $H_j = B_{ij}^T \Omega_{ij} B_{ij};$ 
5    $H_{ij} = H_{ij} + A_{ij}^T \Omega_{ij} B_{ij};$ 
6    $s_i = A_{ij}^T \Omega_{ij} error_{ij};$ 
7    $s_j = B_{ij}^T \Omega_{ij} error_{ij};$ 
8 end
9 //Hessian Diagonal blocks computing
10 foreach pose  $P_i \in RobotPoses$  do
11   foreach edge  $e$  that  $P_i$  is connected to do
12      $H_{ii} \leftarrow H_{ii} + e.H_i;$ 
13      $b_i \leftarrow b_i + e.s_i;$ 
14   end
15 end
16 foreach landmark  $L_j \in Landmarks$  do
17   foreach edge  $e$  that  $L_j$  is connected to do
18      $H_{jj} \leftarrow H_{jj} + e.H_j;$ 
19      $b_j \leftarrow b_j + e.s_j;$ 
20   end
21 end

```

4.2.3 Complément de Schur

La parallélisation du complément de Schur exploite les voisinages des nœuds ainsi que la table Schur-NonZero préparée d'une manière incrémentale dans le bloc FB2 (voir section 3.4.5 dans le chapitre 3). En se référant à l'algorithme 2 détaillant le complément de Schur, la matrice et le vecteur d'information peuvent être calculés séparément. De plus, tout bloc non nul de la matrice d'information ou du vecteur d'information peut être calculé indépendamment des autres blocs. Par conséquent, grâce à la structure de données incrémentale et à la réécriture algorithmique du bloc fonctionnel de Schur (FB4), aucun mécanisme de synchronisation n'est nécessaire.

Sur GPU, nous pouvons avoir deux implantations différentes du complément de Schur. La première implantation se base sur l'algorithme 2 et utilise la mémoire globale du GPU. La deuxième implantation utilise, par contre, la mémoire partagée du GPU (GPU Shared Memory) où un ensemble de threads collaborent pour limiter les opérations d'interaction avec la mémoire globale. L'idée est très proche de celle utilisée dans le calcul matriciel sur GPU [Bell and Garland, 2008, 2009]. Le temps d'accès à la mémoire partagée est plus faible que le temps d'accès à la mémoire globale du GPU.

Cependant, pour avoir de meilleures performances, l'utilisation de la mémoire partagée implique que le voisinage d'un nœud doit avoir au moins 32 amers. Le but est d'avoir des blocs de threads dont la taille est multiple de 32. Or, ce voisinage est relativement large et ne peut être garanti, dans la pratique, si l'on considère que l'objectif est de localiser le robot plutôt que de reconstruire l'environnement. Pour cette raison, nous n'avons pas utilisé la mémoire partagée du GPU. La première implantation utilisant la mémoire globale donne de meilleures performances.

4.2.4 Résolution du système linéaire

Comme le montrent les évaluations dans le chapitre 3, le bloc de résolution du système linéaire (FB6) peut être très contraignant en temps de calcul. Cela dépend des dimensions du système, du nombre de blocs non nuls, mais également de la répartition de ceux-ci dans la matrice d'information. Il est à noter que ce travail de thèse ne s'intéresse pas à la parallélisation de l'algorithme de Cholesky. Ce problème a fait l'objet de nombreuses études dans la communauté scientifique et a donné lieu à une multitude de solutions aussi bien sur des architectures mono-cœurs que parallèles [Davis, 2006, Gould et al., 2007, Gupta et al., 1997, Chen and Chen, 2015]. Nous avons évalué dans notre travail [Dine et al., 2014] la librairie HSL (Harwell Subroutine Library) [HSL, 2014]. Celle-ci fournit une implantation mono-cœur et multi-cœur. Cette dernière permettait un gain avoisinant les 15% sur une architecture OMAP4430 à deux cœurs ARM opérant à 800MHz. Nous avons jugé convenable de traiter le problème d'implantation sur architectures hétérogènes indépendamment de toute librairie externe qu'elle soit statique ou dynamique. Cependant, nous avons eu recours à une librairie open-source, à savoir CSparse [Davis, 2006]. Les routines nécessaires ont été directement incorporées dans le code.

4.2.5 Calcul des incrément des amers

De la même manière que pour le complément de Schur, le calcul des incrément est basé sur une multiplication matricielle par bloc. L'algorithme 7 décrit l'algorithme réécrit de telle sorte à exploiter les voisinages des nœuds au lieu d'effectuer une multiplication matricielle classique. Les blocs non nuls sont récupérés au moyen des listes d'arêtes entrantes et sortantes. Cet algorithme tire aussi profit du fait que la matrice H_{ll} est diagonale par bloc. Le calcul d'un incrément n'implique aucune dépendance de données avec les autres incrément. Par conséquent, les incrément peuvent être calculés en parallèle.

4.3 Effet de la fonctionnalité zéro-copie

Les modèles de programmation de OpenCL ou CUDA offrent des mécanismes pour partager des zones mémoire entre le processeur hôte et le GPU sans avoir à transférer explicitement les données. Cette fonctionnalité est connue sous le nom zéro-copie. Elle a été introduite à partir de la version

Algorithme 7 : Calcul des incrément des amers

```

1 foreach landmark k in Landmarks do
2   //  $N_k$  : the neighborhood of landmark  $k$  (robot poses)
   //  $b$  : information vector
   //  $y$  : poses increments
   foreach robot pose i in  $N_k$  do
3     |  $sum = sum + Hl p_{ki} * y_i;$ 
4   end
5    $b_k = b_k - sum;$ 
6    $y_k = H_{kk}^{-1} * b_k$ 
7 end

```

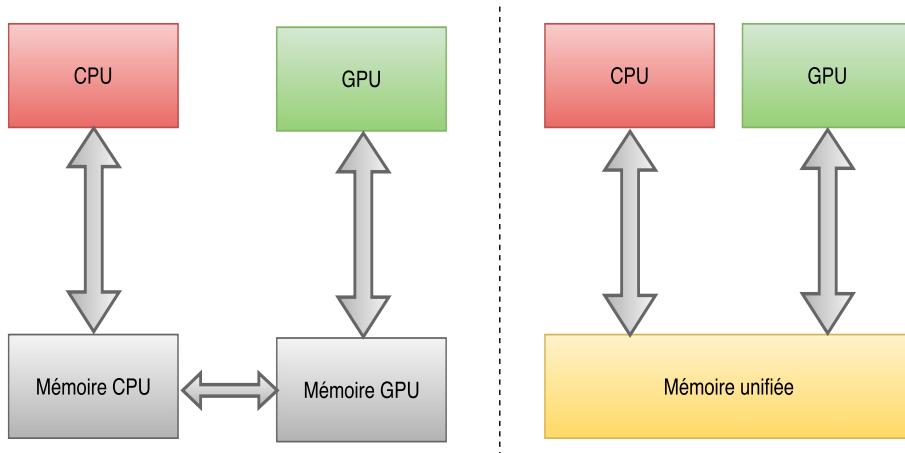


FIGURE 4.4 – Mémoire physique unifiée.

2.0 de CUDA. Il est possible que le processeur principal et les coprocesseurs se partagent des espaces mémoire en lecture/écriture sans avoir à transférer d'un espace privé à l'autre. La zéro-copie implique l'allocation d'une zone mémoire non paginée (pinned-memory). L'espace zéro-copie est alloué dans la mémoire du processeur hôte. Celle-ci est gérée, avec son hiérarchie de mémoire cache, par le système d'exploitation. Cela implique aussi que les données doivent résider en mémoire. Elles ne sont donc pas mises en cache afin de garantir la cohérence de données entre les deux processeurs.

Par ailleurs, dans les architectures classiques, les espaces mémoires de l'hôte et du GPU sont physiquement séparées. La zéro-copie permet au GPU d'accéder à la mémoire de l'hôte via un bus PCI Express (PCIe). Cela a l'avantage d'envoyer des données directement aux multiprocesseurs de flux (Streaming Multiprocessors) sur le bus PCIe. Cet envoi s'effectue indépendamment de la mémoire principale du GPU. Contrairement aux architectures classiques, les nouvelles architectures hétérogènes embarquées telles que le Tegra K1 ont une mémoire physiquement partagée entre les différents calculateurs (Fig. 4.4). Le système sur puce dispose donc d'une mémoire unifiée accessible par tous les calculateurs.

Pour aller plus loin et pallier au problème de l'espace mémoire virtuel entre CPU et GPU, CUDA 4.0 a introduit à partir des architectures Fermi, une intégration complète des espaces virtuels. L'utilisation d'un espace virtuel unifié entre les deux processeurs est alors possible. Concrètement, cela veut dire

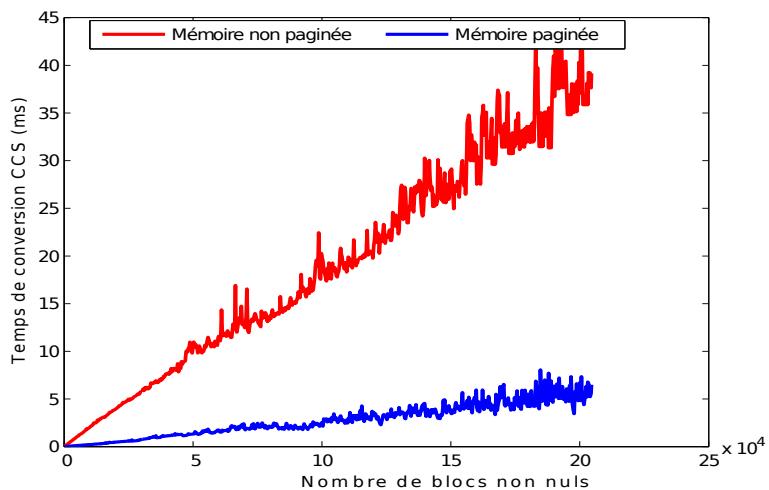


FIGURE 4.5 – Temps de construction du format CCS (FB5) sur le TK1.

qu'il n'y a plus aucune distinction entre un pointeur hôte et un pointeur GPU.

Bien que ces fonctionnalités de mémoire partagée et d'espace virtuel unifié facilitent énormément l'implantation sur les architectures hétérogènes à base de GPU, elles peuvent, tout de même, dégrader les performances à cause du cache. Notre expérience avec le Tegra K1 et le Exynos 5422 a montré que la zéro-copie peut dramatiquement affecter le temps de calcul. La figure 4.5 donne le temps de construction du format CCS, avec et sans zéro-copie, en fonction du nombre de blocs non nuls. Notons que dans ce bloc (FB5) le CPU lit les éléments de la matrice d'information (données en double précision) et réécrit les éléments non nuls dans un format compressé. La matrice d'information est allouée, dans le premier cas, dans une zone mémoire non paginée (zéro-copie). Le format compressé est, par contre, mis dans un espace mémoire paginé. Plus le nombre d'éléments augmente, plus le temps de construction avec un espace non paginé augmente d'une manière dramatique en comparaison avec l'espace paginé. Cela est dû au fait que les données ne sont pas mises dans le cache dans le premier cas.

Du fait que les données ne sont pas mises dans le cache, la mémoire non paginée ne doit être lue ou écrite qu'une seule fois et il faut que les lectures/écritures des threads soient voisines. Par conséquent, le CPU et le GPU ne peuvent pas partager de larges zones modifiables. Dans nos implantations, la zéro-copie est uniquement réservée pour stocker des paramètres et des informations sur la structure du graphe (nombre de noeuds, poses, amers, arêtes,...) et des paramètres de configuration. Le transfert de données d'un processeur à l'autre s'avère inévitable. Cependant, il est important de souligner l'impact positif d'avoir des mémoires physiquement partagées entre CPU et GPU. En effet, cela réduit considérablement le temps d'accès à la mémoire du côté GPU.

4.4 Adéquation Algorithme-Architecture

Notre approche d'implantation sur les architectures hétérogènes s'appuie, en premier lieu, sur le découpage primaire de l'algorithme en blocs fonctionnels. Ce découpage a été fait en fonction de la nature algorithmique des tâches (opérations arithmétiques, interaction avec la mémoire, dépendances de données). Le découpage permet également de minimiser les transferts de données ou encore favoriser les transferts par bloc si nécessaire. Dans ce travail, le mappage des blocs fonctionnels est essentiellement guidé par les performances temporelles globales. Par ailleurs, le GraphSLAM utilise un espace mémoire relativement large dont l'organisation des données impactent directement les performances tel que nous l'avons démontré dans le chapitre précédent. Par conséquent, la répartition des blocs fonctionnels dépendra également du transfert de données entre CPU et GPU.

Soient :

- FB : Ensemble des blocs fonctionnels
- P : Ensemble des calculateurs disponibles (CPU, GPU)
- TC_{ij} : Temps d'exécution du bloc fonctionnel i sur le calculateur j .
- TR_{ij} : Temps de transfert de données entre les blocs fonctionnels i et j
- D : Graphe de dépendances entre les blocs fonctionnels. Certains blocs (ou sous-blocs) peuvent être exécutés en parallèle. On prendra le maximum des temps. La même règle est appliquée pour les transferts de données.
- S : Graphe d'affectation ou de partitionnement. $S = (P, FB)$. Un arc (i, j) implique que FB_i est affecté au processeur P_j . Un graphe d'affectation (partitionnement) doit vérifier les règles de dépendances D .
- S_{tr} : Transferts de données possibles associé à un graphe d'affectation. Un arc (i, j) implique la nécessité d'un transfert de données de FB_i à FB_j

Il s'agit formellement de trouver un graphe d'affectation S^* tel que :

$$S^* = \operatorname{argmin}_S (\sum_{(i,j) \in S} TC_{ij} + \sum_{(x,y) \in S_{tr}} TR_{xy})$$

Cette fonction objectif fait intervenir les temps de calcul des blocs fonctionnels et les différents transferts entre ces blocs. Pour minimiser cette fonction, nous nous appuyons sur une analyse incrémentale du temps de calcul. Plus concrètement, il nous faut étudier le comportement des différents temps de calcul au fil des étapes d'optimisation de graphe. Nous ne nous limitons pas à une seule étape d'optimisation de graphe dans notre analyse. Notre motivation est liée au fait que notre travail vise plus particulièrement le SLAM incrémental. Du point de vue architecture, la réutilisation des données des étapes précédentes (historique de la trajectoire et la carte) augmente l'efficacité du cache processeur. Par ailleurs, une analyse incrémentale des performances permet d'expliquer l'évolution du temps de calcul afin de prévoir le comportement de celui-ci au fur et à mesure que le graphe va s'élargir.

Dans notre implantation LS, la construction du système (FB3) et la résolution du système par Cholesky (FB6) sont les plus importants en termes de charges temporelles. La construction du système (FB3) est caractérisée par un grand nombre d'opérations mémoire. La construction du graphe (FB1) n'est pas pertinente en temps de calcul, comparée aux autres blocs fonctionnels. FB1 implique aussi

l'accès à la structure de données représentant le graphe à chaque nouvelle pose ou observation. L'affectation de FB1, à un processeur ou un autre, dépendra de l'affectation de FB3 pour éviter le transfert de données. FB1 et FB3 seront donc attribués au même processeur. Par ailleurs, la construction du format CCS (FB5) contient majoritairement des opérations de chargement/rangement en mémoire avec peu d'opérations arithmétiques. Pour cette raison, il est convenable d'effectuer cette tâche, sans parallélisation, sur CPU. La résolution du système linéaire n'est pas parallélisée et est assurée par le solveur de la librairie CSparse que nous avons incorporé dans le code source. Ces considérations permettent de simplifier la fonction objectif définie auparavant.

4.4.1 Scalabilité

Avant d'analyser les performances globales des blocs fonctionnels sur les deux processeurs (CPU et GPU) du TK1, il est important d'étudier la scalabilité, c'est-à-dire la variation des performances en fonction du nombre de threads. Pour ce faire, nous avons pris comme bloc fonctionnel la construction du système linéaire (FB3). Celui-ci est parallélisé et peut être très gourmand en temps de calcul. Nous avons fait varier le nombre de threads pour exécuter FB3, sur le jeu de données de Victoria Park (Fig. 3.5b), avec une optimisation de graphe toutes les 10 étapes. Notons, par ailleurs, que nous avons activé la vectorisation automatique du code (dans les options de compilation) en vue d'exploiter l'unité NEON présente dans le SoC TK1. Cependant, une investigation du code assembleur généré montre que l'exploitation de l'unité NEON est limitée dans LS. Nous n'avons malheureusement pas étudié, dans ce travail, l'apport de la vectorisation manuelle qui peut améliorer encore le temps de calcul.

Les figures 4.6 et 4.7 illustrent la variation du temps de construction (FB3) sur respectivement le CPU et le GPU. On constate d'abord que les temps ont un comportement linéaire, sur CPU comme sur GPU, quelque soit le nombre de threads. Avec ces quatre processeurs Cortex-A15, les performances optimales, sur CPU, sont atteintes en utilisant 8 threads. Au-delà de 8 threads, le temps de calcul ne s'améliore plus et une grande variation est observée vers la fin de la trajectoire où le nombre d'arêtes est relativement important. Cela est expliqué par le goulot d'étranglement qui se produit sur les accès mémoires (accès concurrents par les threads).

Quant au GPU, nous avons fait varier le nombre de threads au sein d'un même bloc CUDA. Le nombre total de threads lancés est calculé dynamiquement suivant la taille du bloc CUDA et le nombre d'arêtes à traiter. On constate qu'au-delà de 32 threads par bloc, le temps ne s'améliore plus. Un ralentissement relatif est même observé sur la fin de la trajectoire. Comme expliqué dans le chapitre 2, les blocs CUDA sont automatiquement divisés en petits paquets (wraps) de 32 threads. Comme le TK1 ne dispose que d'un seul multiprocesseur de flux et les threads sont organisés en wraps de 32, des blocs de 32 threads donnent les meilleures performances.

Par ailleurs, on remarque qu'une configuration de 32 threads par bloc, pour le GPU, accélère d'un facteur 5 l'exécution parallèle et d'un facteur 22 l'exécution séquentielle sur le CPU.

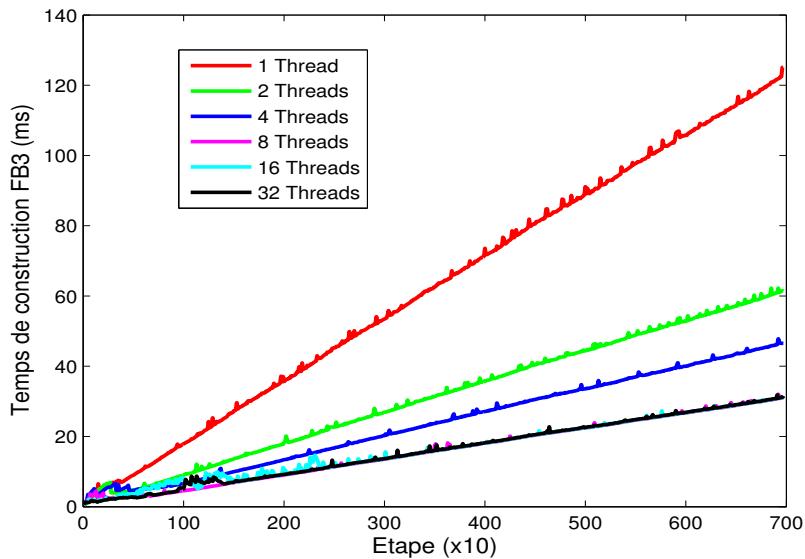


FIGURE 4.6 – Variation du temps de construction du système linéaire en fonction du nombre de threads sur CPU.

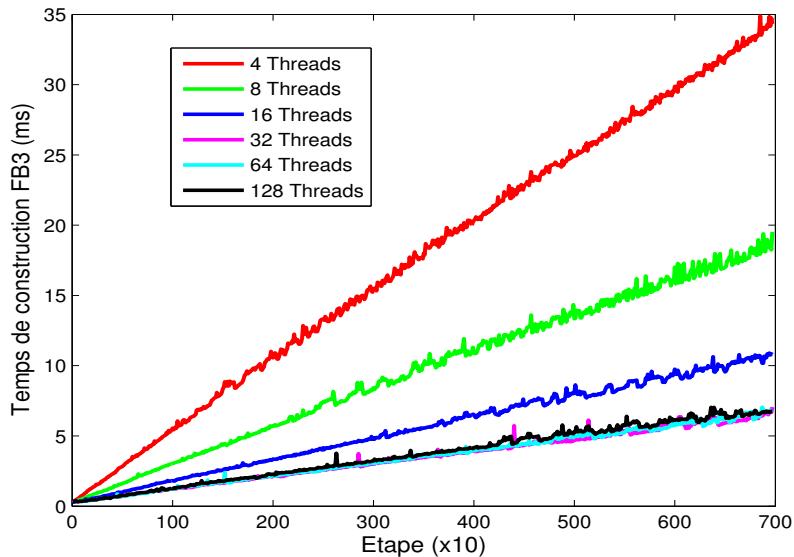


FIGURE 4.7 – Variation du temps de construction du système linéaire en fonction du nombre de threads sur GPU.

4.4.2 Analyse de l'évolution du temps par arête

Dans cette section, nous analysons l'évolution temporelle en mode incrémental des blocs fonctionnels. Le mode incrémental permet d'analyser les performances temporelles au fur et à mesure que le robot avance dans l'environnement. Nous pourrons voir l'impact de l'augmentation du nombre d'arêtes et la structure du graphe sur le temps de calcul global. Pour cela, nous avons utilisé un simulateur pour générer des environnements synthétiques [Vincke, 2012]. L'utilisation d'un simulateur

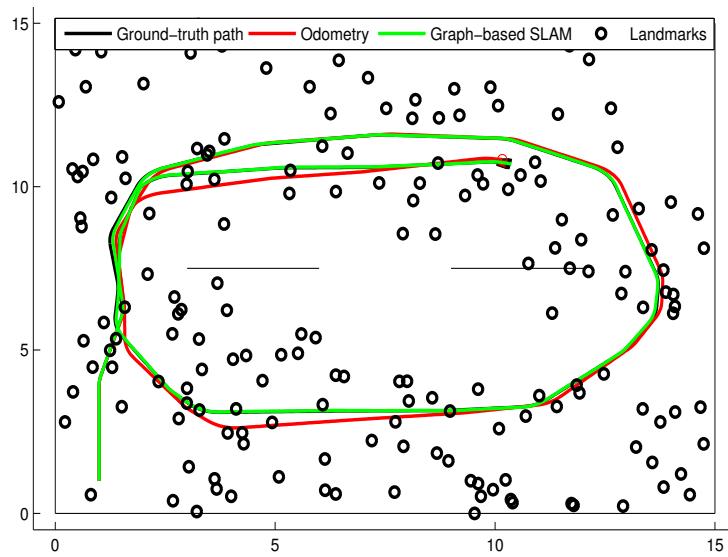


FIGURE 4.8 – Exemple d'un environnement synthétique utilisé pour les évaluations [Vincke, 2012].

permet de générer des trajectoires et environnements qui ne sont pas disponibles dans les jeux de données fournis par la communauté scientifique. La figure 4.8 présente un exemple d'un environnement synthétique.

De la même manière que dans l'évaluation de la structure de données LS, nous utilisons la métrique TPE (Time Per graph-Element) pour l'analyse des caractéristiques temporelles d'un bloc fonctionnel sur l'une ou l'autre des architectures. Nous rappelons qu'un élément de graphe est une arête pour FB2, FB3, FB4 et un nœud pour FB7 et FB8.

En se basant sur l'analyse incrémentale, nous discuterons l'adéquation algorithme-architecture qui offre les meilleures performances. L'étude ne portera pas sur les blocs FB1, FB5, FB6. La construction du format CCS (FB5) et la résolution du système linéaire (FB6) n'entrent pas dans les comparaisons car ils seront affectés au CPU comme expliqué auparavant. La construction du graphe (FB1) ne fait pas non plus partie de l'étude car FB1 sera mappé sur le même processeur que FB3.

a. TPE évolution

Pour analyser le comportement du TPE, nous lançons l'optimisation de graphe sur 9500 étapes d'un graphe simulé. Pour éviter les erreurs de mesures, le nombre d'itérations du processus Gauss-Newton a été fixé à 20 itérations. Nous prélevons ensuite, à chaque étape d'optimisation, le temps moyen de chaque bloc fonctionnel. La figure 4.9 montre l'évolution de la structure du graphe le long de la trajectoire. Sur cet environnement, le nombre d'éléments de graphe augmente rapidement au début de la navigation. Comme l'environnement simule un robot explorant une salle de $15 \times 15 m^2$, il a tendance à réobserver les mêmes amers. Cela implique que le nombre d'amers devient stable vers la fin de

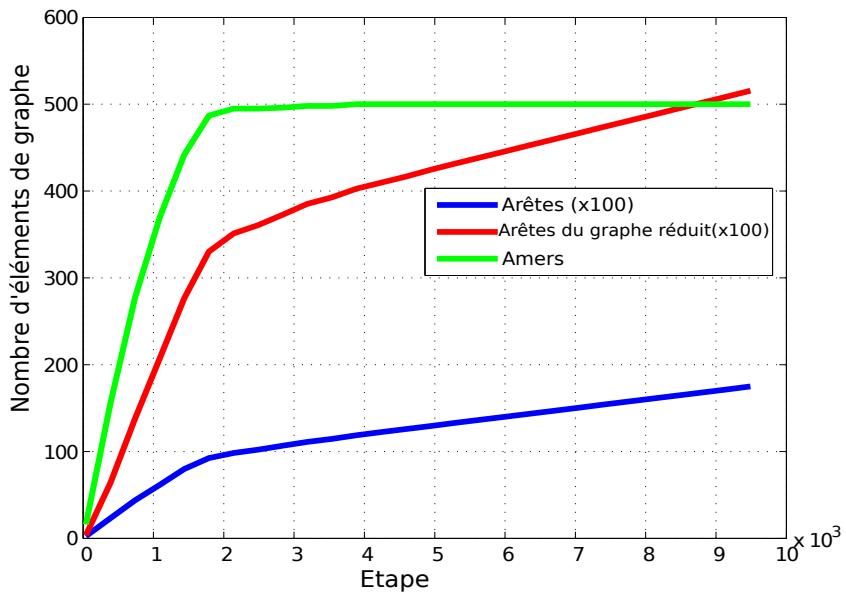


FIGURE 4.9 – Évolution de la structure du graphe de l’environnement simulé.

la trajectoire. L’absence de nouveaux amers implique que le nombre d’arêtes, dans les deux graphes initial et réduit, croît moins rapidement au bout d’un certain nombre d’étapes. Il est à noter qu’un élément de graphe dans le graphe réduit (FB4) correspond en réalité à une pose robot ou une arête. Les éléments non nuls subissent le même traitement dans FB4. Ainsi, pour des raisons de simplicité, nous utiliserons le terme arête pour désigner un élément non nul dans FB4.

4.4.3 Discussion et partitionnement CPU/GPU

L’évolution des blocs fonctionnels, calcul des erreurs (FB2) et construction du système linéaire (FB3), est représentée dans la figure 4.10. Au début, le TPE est relativement important sur les deux architectures. Le temps est dominé par les latences de lancement des threads sur CPU comme sur GPU. Le TPE sur le CPU Intel tend par la suite à être relativement stable. Au-delà de 10000 arêtes, ce dernier n’est plus stable et commence à croître rapidement. Le même comportement est constaté sur le CPU du Tegra K1 au-delà de 16000 arêtes. Le nombre de threads pouvant coopérer pour exécuter en parallèle une tâche sur CPU est limité par le nombre de coeurs physiques disponibles, mais également par les caractéristiques de la mémoire (nombre de ports de lecture/écriture). Pour les plateformes utilisées dans ce travail, la mémoire est partagée entre les différents coeurs physiques (CPU). Les threads accèdent de manière concurrente à la mémoire. Les latences des opérations mémoire augmentent. Un thread peut être bloqué en attente de la fin des opérations mémoire des autres threads. Par ailleurs, au fur et à mesure que le robot avance dans l’environnement le nombre d’arêtes augmente. Le nombre de threads devient alors insuffisant pour masquer la latence d’accès mémoire.

Les processeurs graphiques disposent, contrairement aux CPUs, d’une hiérarchie mémoire plus complexe. A la différence des CPUs, l’accès aux mémoires locales et privées du GPU est plus rapide que

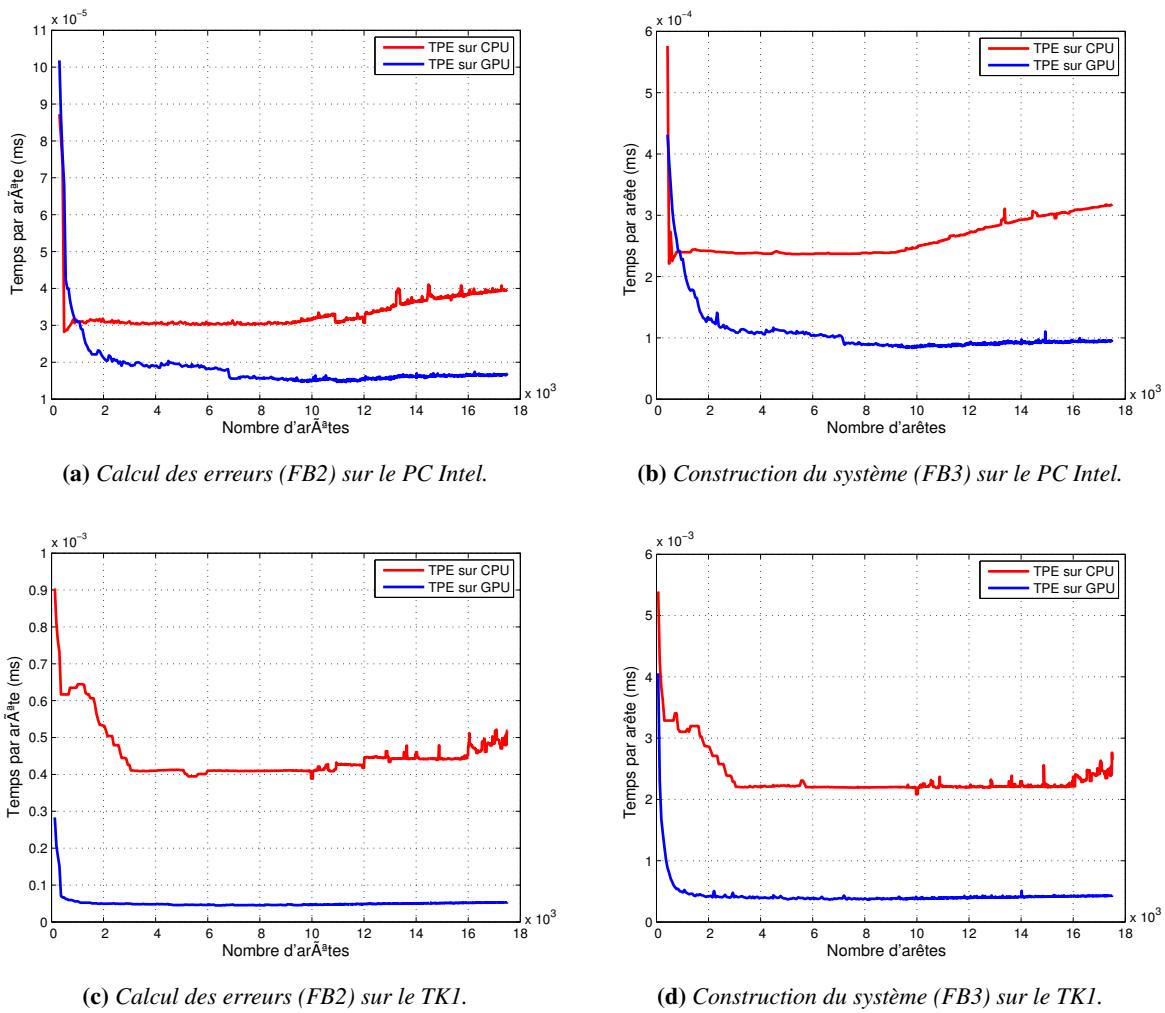


FIGURE 4.10 – Évolution du TPE.

l'accès à la mémoire globale. En plus, sur le GPU, les threads sont lancés en mode SIMD. L'unité de chargement/rangement du multiprocesseur de flux permet de réduire la charge d'interaction avec la mémoire (jusqu'à 32 opérations en même temps). Pour ces raisons, le TPE dans cette expérimentation tend à être relativement stable en dépit de la croissance continue du nombre d'arêtes. Vers la fin de la trajectoire, sur le Tegra k1, le TPE sur GPU est approximativement 5 fois plus faible que le TPE sur CPU. En revanche, sur le PC Intel, le TPE sur GPU correspondant à la construction du système est 3 fois plus faible que le TPE sur CPU. Notons, par ailleurs, que les TPEs sur les blocs FB7 et FB8 ont pratiquement le même comportement que sur FB2 et FB3.

a. Complément de Schur

Globalement, le complément de Schur se caractérise par un temps quadratique à cause des multiplications matricielles. La réécriture proposée de l'algorithme de Schur est relativement linéaire en nombre d'arêtes. Le temps de calcul de FB4 dépend essentiellement du nombre d'arêtes après marginalisation des amers. D'autre part, la taille du voisinage d'un nœud dans le graphe réduit affecte également le

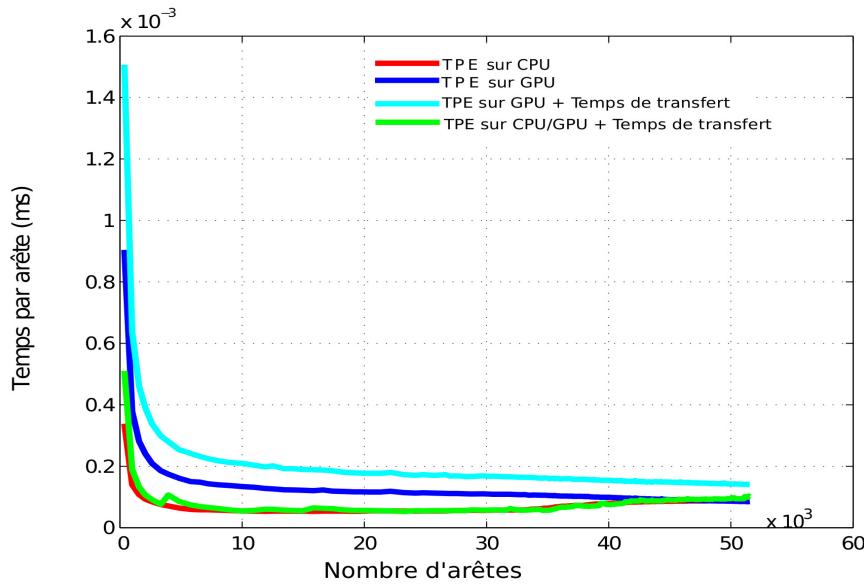


FIGURE 4.11 – Évolution du TPE dans le complément de Schur sur le TK1.

temps de calcul. Plus le voisinage est petit, plus le calcul est rapide. La taille de voisinage détermine le nombre d’itérations séquentielles qui pourraient ralentir l’exécution (ligne 13 à 17 dans l’algorithme 4). La figure 4.11 présente l’évolution du TPE du complément de Schur (FB4). Si l’on ne prend pas en compte le transfert du nouveau système linéaire représentant le graphe réduit, le CPU donne le meilleur TPE au début de la trajectoire. Avec le déplacement du robot, le nombre d’arêtes augmente et le TPE sur GPU dépasse légèrement celui sur CPU. Ce gain est aussi lié à la taille des voisinages des nœuds. De même, plus le voisinage est petit, plus le gain en TPE est élevé par rapport au CPU. Chaque nœud, vers la fin de la trajectoire, est relié à au plus trois amers. Cela explique le léger gain sur la fin de la trajectoire. Il est important de souligner que le calcul de FB4 sur GPU est tout de même ralenti par le grand nombre d’accès en lecture/écriture à la mémoire globale du GPU.

La marginalisation d’amers génère, néanmoins, beaucoup plus d’arêtes (blocs non nuls) qu’il n’y en avait dans le graphe initial. Le transfert de ces arêtes (le nouveau système ; H_{pp} et B_p) présente un facteur limitant pour l’exécution de FB4 sur GPU. En fonction du graphe, le temps de transfert peut devenir prohibitif par rapport au temps total du bloc fonctionnel. Dans cette expérimentation, 51574 blocs de H_{pp} (soit 3.54Mo) sont transférés à la dernière étape. La figure 4.11 montre également qu’en considérant le temps de transfert, le TPE sur GPU augmente considérablement comparativement au TPE sur CPU.

b. Discussion

En se basant sur l’analyse des évaluations décrites ci-dessus, nous pouvons dire que le GPU donne de meilleures performances, en termes de TPE, par rapport au CPU pour les blocs : calcul des erreurs (FB2), construction du système (FB3), calcul des incrémentations des amers (FB7) et mise à jour de l’état du système (FB8). Cependant, le CPU s’avère plus convenable pour calculer le complément de Schur

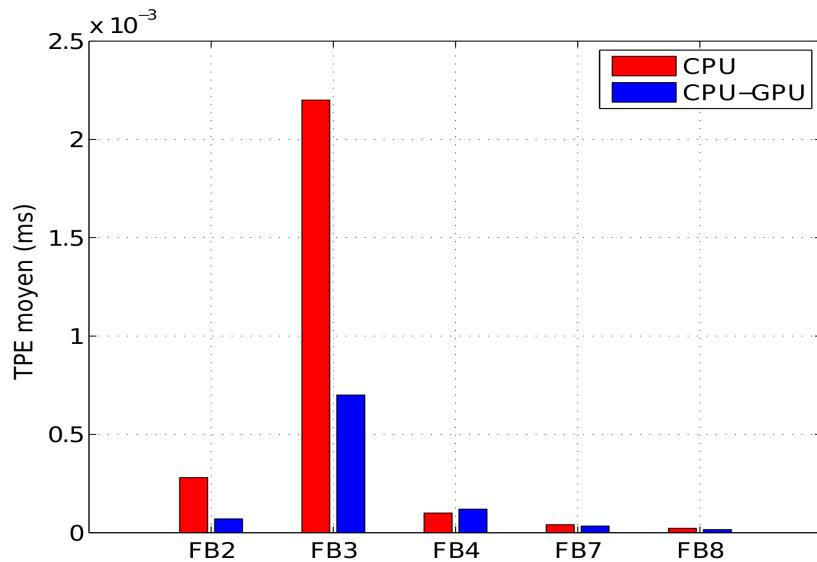


FIGURE 4.12 – TPEs moyens des blocs fonctionnels sur le TK1.

(FB4) à cause du transfert de données. Maintenant que le CPU est plus intéressant pour exécuter FB4, on a toujours besoin de transférer la matrice d'information H à l'hôte (CPU). Ce transfert n'est pas aussi prohibitif que le premier du moment que seulement les blocs non nuls sont transférés. Pour améliorer encore le TPE et éviter de retransférer les données au GPU, nous décomposons FB4 en trois sous-blocs. Le premier inverse la matrice H_{ll} sur GPU. Le deuxième calcule le vecteur d'information B_p sur GPU. Le dernier calcule, sur CPU, la nouvelle matrice d'information H_{pp} après réception de la matrice d'information initiale H . L'utilisation de ce modèle coopératif entre CPU et GPU permet d'avoir le meilleur TPE pour FB4. Même en tenant compte du temps de transfert de données, le TPE fourni par la coopération CPU-GPU recouvre relativement le TPE basé CPU.

La répartition finale des blocs fonctionnels sur les deux processeurs (CPU et GPU) a été choisie de telle manière à améliorer le temps de calcul global. La figure 4.13 donne le partitionnement choisi, incluant les espaces mémoire et les différents processeurs. Uniquement une partie du complément de Schur (FB4), la construction du format CCS (FB5) et la résolution du système linéaire (FB6) sont exécutées sur le CPU. Les TPEs moyens sont représentés dans le graphique de la figure 4.12. En général, pour FB2, FB3, FB7 et FB8, le GPU peut accélérer le traitement jusqu'à 5 fois en comparaison avec le CPU. Notons aussi que le temps moyen de FB1 (mise à jour de la structure de graphe) est d'environ $800\mu s$ sur le GPU et $200\mu s$ sur le CPU du Tegra K1. FB1 est, tout de même, affecté au GPU pour éviter de transférer la structure du graphe à l'hôte car le bloc le plus important, à savoir FB3, est affecté au GPU.

Le flux de traitement du modèle CPU/GPU est répété à chaque optimisation de graphe, et plus précisément à chaque itération du processus de Gauss-Newton. Le temps de transfert dans le modèle proposé CPU/GPU est relativement petit en comparaison avec les temps des autres blocs fonctionnels. Pour illustrer encore le partitionnement élaboré, la figure 4.14 donne l'ordonnancement des blocs fonctionnels pour une itération de Gauss-Newton. FB4 est partitionné sur les deux processeurs (CPU et GPU).

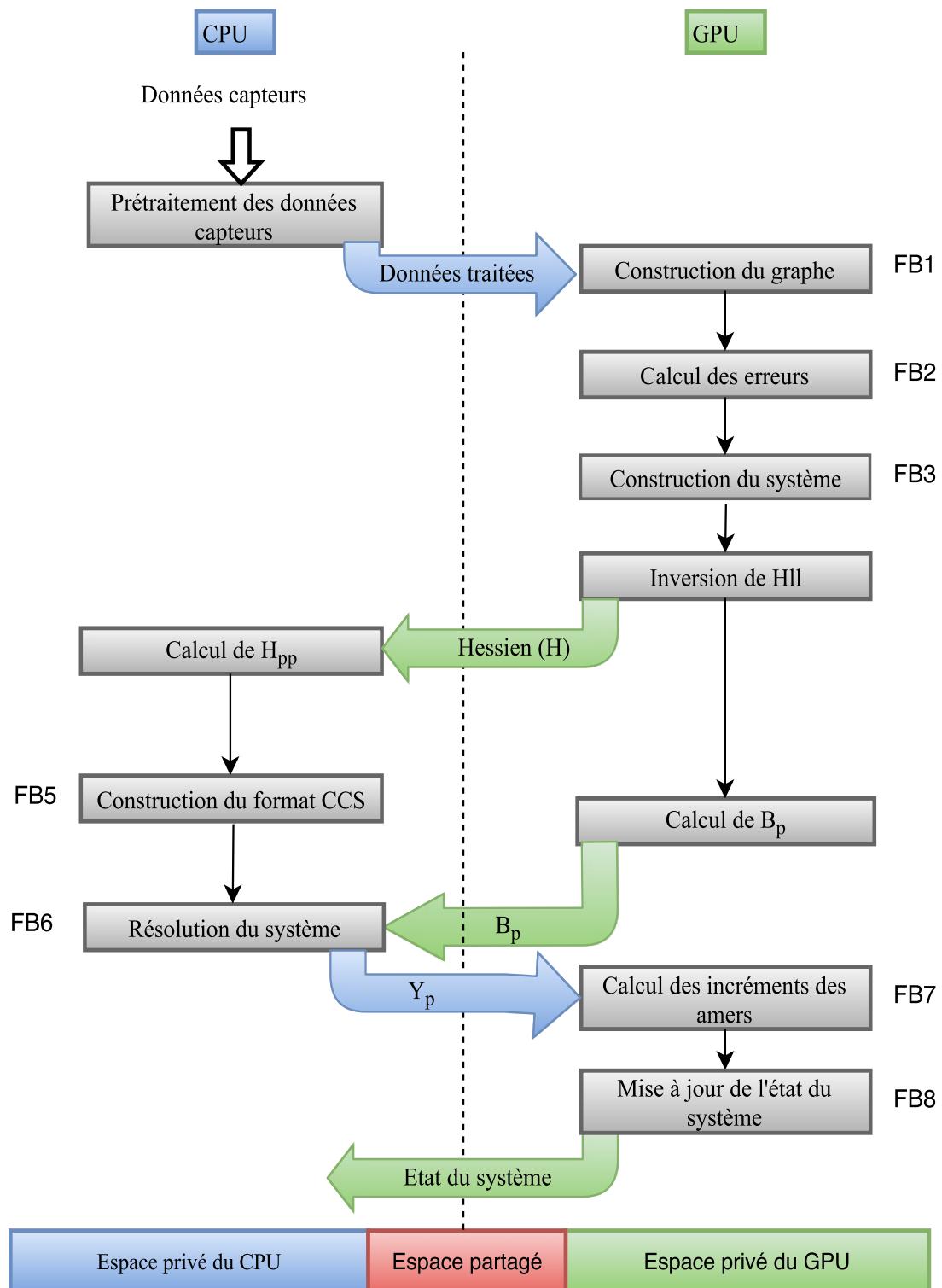


FIGURE 4.13 – Modèle CPU/GPU.

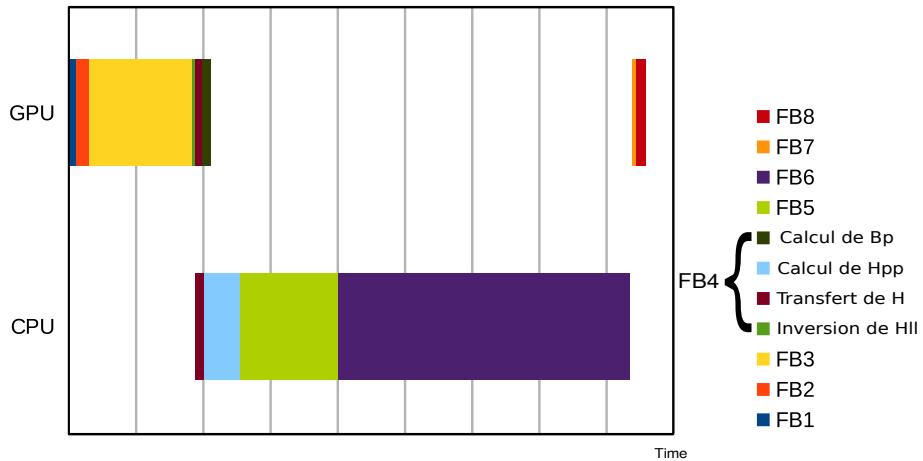


FIGURE 4.14 – Ordonnancement des blocs fonctionnels dans l’implantation hétérogène CPU/GPU.

Cet ordonnancement est fonction des dépendances de données entre blocs fonctionnels. Le transfert de B_p et Y_p est négligeable (moins de 256KO pour un graphe de 10000 noeuds).

4.4.4 Architecture mémoire

L’architecture mémoire a été choisie de telle sorte à ce qu’elle soit efficacement accessible par le CPU et le GPU. Suivant le partitionnement CPU/GPU élaboré, il est important d’adapter la représentation mémoire aux architectures hétérogènes à base de GPU. Comme le montre la figure 4.13, nous utilisons trois espaces mémoire différents. Ce choix a été fait dans le but de minimiser le transfert de données ainsi que l’utilisation de la mémoire non paginée (zéro-copie).

- Espace zéro-copie : Cet espace est alloué en zone mémoire non paginée. On y stocke particulièrement des paramètres d’optimisation de graphe et des données sur la structure du graphe (nombre de poses, amers, arêtes,...). Le but est de garantir la cohérence entre le CPU et le GPU sans pour autant avoir à transférer ces données à chaque fois qu’elles changent. Cependant, l’accès à cet espace étant coûteux des deux cotés CPU et GPU, il est important de réduire autant que possible les accès à cette zone.
- Espace privé GPU : Cet espace est utilisé pour la représentation mémoire du graphe et la construction du système linéaire. On y retrouve plus particulièrement la représentation mémoire (LS) décrite dans le chapitre 3 (Fig. 3.3).
- Espace privé CPU : Cette zone mémoire est essentiellement allouée pour résoudre le système linéaire (FB5 et FB6). Il est aussi nécessaire de garder trace de la connectivité dans le graphe sur CPU. Cela permettra d’accélérer la conversion en format CCS de la matrice d’information. Pour ce faire, on utilise un vecteur, comme sur GPU, où chaque entrée contient deux listes indexant les arêtes-sortantes et arêtes-entrantes du nœud correspondant. Ce vecteur est mis à jour de manière incrémentale, sur CPU, en parallèle avec la construction incrémentale sur GPU (algorithme 4).

TABLE 4.1 – Temps des blocs fonctionnels après 1 itération (ms).

	FB2	FB3	FB4	FB5	FB6	FB7	FB8
g^2o	13.80	112.87	38.49	294.76	845	2.81	8.12
LS-CPU	6.21	91.77	8.41	51.03	821.90	0.35	3.74
LS-GPU	1.23	5.83	7.88	50.56	823.11	0.33	1.12

4.5 Résultats expérimentaux

Nous présentons dans cette section les évaluations temporelles des deux implantations parallèles de LS ; homogène (CPU) et hétérogène (CPU/GPU). Pour une comparaison avec l'état de l'art, nous avons aussi évalué une version parallèle de g^2o . Les trois implantations utilisent la même librairie CSparse pour la résolution du système linéaire. Les mêmes jeux de données présentés dans le chapitre 3 sont utilisés (Fig. 3.5). La plateforme d'évaluation principale est la Jetson qui embarque un TK1. Nous présentons également, pour des raisons de comparaison, certains résultats sur un PC doté d'un processeur Intel.

4.5.1 Répartition des charges temporelles

Les graphes les plus fréquents en pratique sont les graphes à amers tels que celui de Victoria Park. Pour LS, les évaluations en mode mono-thread ont montré que les blocs les plus gourmands en temps étaient la construction du système (FB3) et sa résolution (FB6). FB3 avait pratiquement la même portion de temps que FB6. Les optimisations introduites en parallélisant plus particulièrement FB3 sur CPU ou sur GPU affectent la distribution du temps sur les blocs fonctionnels. La répartition du temps sur le PC Intel et le TK1 est résumée dans la figure 4.15. On remarque que le bloc FB6 devient majoritairement le plus important. Pour le PC Intel, FB3 ne représente maintenant que 20% pour l'implantation homogène, contre 10% seulement pour l'implantation hétérogène de LS. Quant au TK1, FB3 ne prend que 35% de la charge totale. Celle-ci représentait plus de 60% dans la version mono-cœur. Cette charge est réduite jusqu'à 12% grâce à l'implantation hétérogène CPU/GPU. Notons aussi que la construction du format CCS (FB5) prend un peu plus de charge par rapport à la version mono-cœur. Cette nouvelle distribution est due à la parallélisation du bloc FB3. Par ailleurs, les transferts dans l'implantation hétérogènes représentent moins de 4% du temps global. Cette répartition du temps reste valable dans le cas où on utilise le complément de Schur sur le graphe de Victoria. Le temps de résolution du système linéaire (FB6) est de loin le plus important.

TABLE 4.2 – Temps moyens des blocs fonctionnels après 15 itérations (ms).

	FB2	FB3	FB4	FB5	FB6	FB7	FB8
g^2o	10.81	102.25	35.22	75.65	703.47	2.92	8.02
LS-CPU	5.35	77.81	7.99	31.35	702.35	0.34	3.75
LS-GPU	1.17	5.95	7.56	29.02	701.89	0.31	1.07



FIGURE 4.15 – Charges des blocs fonctionnels.

TABLE 4.4 – Temps de traitement total en mode incrémental. Optimisation de graphe effectuée toutes les 10 étapes.

	Manhattan	10k	City10k	Intel	FR H	FR079	Victoria	CityTrees10K
g^2o	21.611	605.239	244.247	1.542	3.287	1.333	89.179	175.356
LS-CPU	12.765	316.535	135.867	0.845	1.740	0.676	30.214	60.187
LS-GPU	8.810	218.517	108.223	0.610	1.123	0.495	23.649	45.822

4.5.2 Optimisation de graphe en mode batch

Avant de présenter les temps d'optimisation des jeux de données en mode batch, nous présentons les temps détaillés de chaque bloc fonctionnel sur le graphe de Victoria Park. L'objectif est de voir l'impact du nombre d'itérations du processus de Gauss-Newton, sur le temps de calcul de certains blocs fonctionnels. Le nombre d'itération² de GN a été fixé à 15.

Les tableaux 4.1 et 4.2 résument les temps après, respectivement, 1 et 15 itérations. Pour le dernier cas, nous donnons plutôt le temps moyen par itération de GN. Pour LS, on remarque qu'après une itération, la construction du système (FB3) sur GPU est approximativement 16 fois plus rapide que sur CPU. Après 15 itérations, alors que les temps sur GPU ne changent pratiquement pas (sauf FB5 et FB6), le temps par itération de FB3 s'améliore encore. Le facteur de gain du GPU par rapport au CPU passe de 16 à 13. Par ailleurs, notons que dans g^2o , la structure du format CCS est préparée lors de la première itération. Aux itérations suivantes, g^2o met seulement à jour les valeurs non nulles. Cela explique la baisse considérable du temps de FB5 après la première itération. Pour FB6, la factorisation symbolique n'est effectuée qu'une seule fois, et ce lors de la première itération. C'est pour cette raison que le temps de FB6 décroît également après la première itération de GN.

TABLE 4.3 – Temps d'optimisation en mode batch sur le TK1 (s).

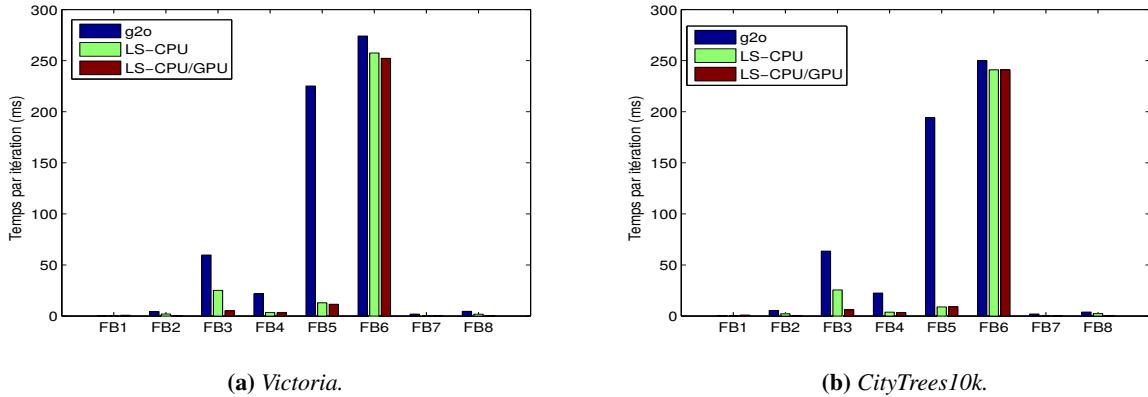
	Manhattan	10k	City10k	Intel	FR H	FR079	Victoria	CityTrees10K
g^2o	0.42	3.82	2.25	0.11	0.09	0.10	1.91	0.937
LS-CPU	0.50	3.79	2.51	0.11	0.06	0.07	1.79	0.89
LS-GPU	0.31	2.83	2.24	0.029	0.019	0.025	1.55	0.76
Itérations	5	5	5	2	1	3	15	5

Le tableau 4.3 présente les temps d'optimisation en mode batch des différents jeux de données utilisés. Le partitionnement CPU-GPU permet d'améliorer encore le temps d'optimisation, par rapport à LS-CPU et g^2o , particulièrement grâce à l'accélération de FB3. En fonction de la charge des blocs FB3 et FB6 et du nombre d'itérations de GN, l'implantation hétérogène accélère l'exécution, par rapport à LS-CPU et g^2o jusqu'à un facteur 1.6. Dans le cas d'utilisation du complément de Schur, le gain apporté par l'implantation hétérogène pour Victoria Park et CityTrees10k est de 10% à cause de l'importante charge de FB6.

2. Dans nos évaluations, le nombre d'itérations minimum pour la convergence du graphe de Victoria Park en mode batch était de 15.

TABLE 4.5 – Temps cumulatif en utilisant le complément de Schur (s).

	g^2o	LS-CPU	LS-CPU/GPU
Victoria Park	413.09	211.54	191.07
CityTrees10k	378.20	198.32	182.58

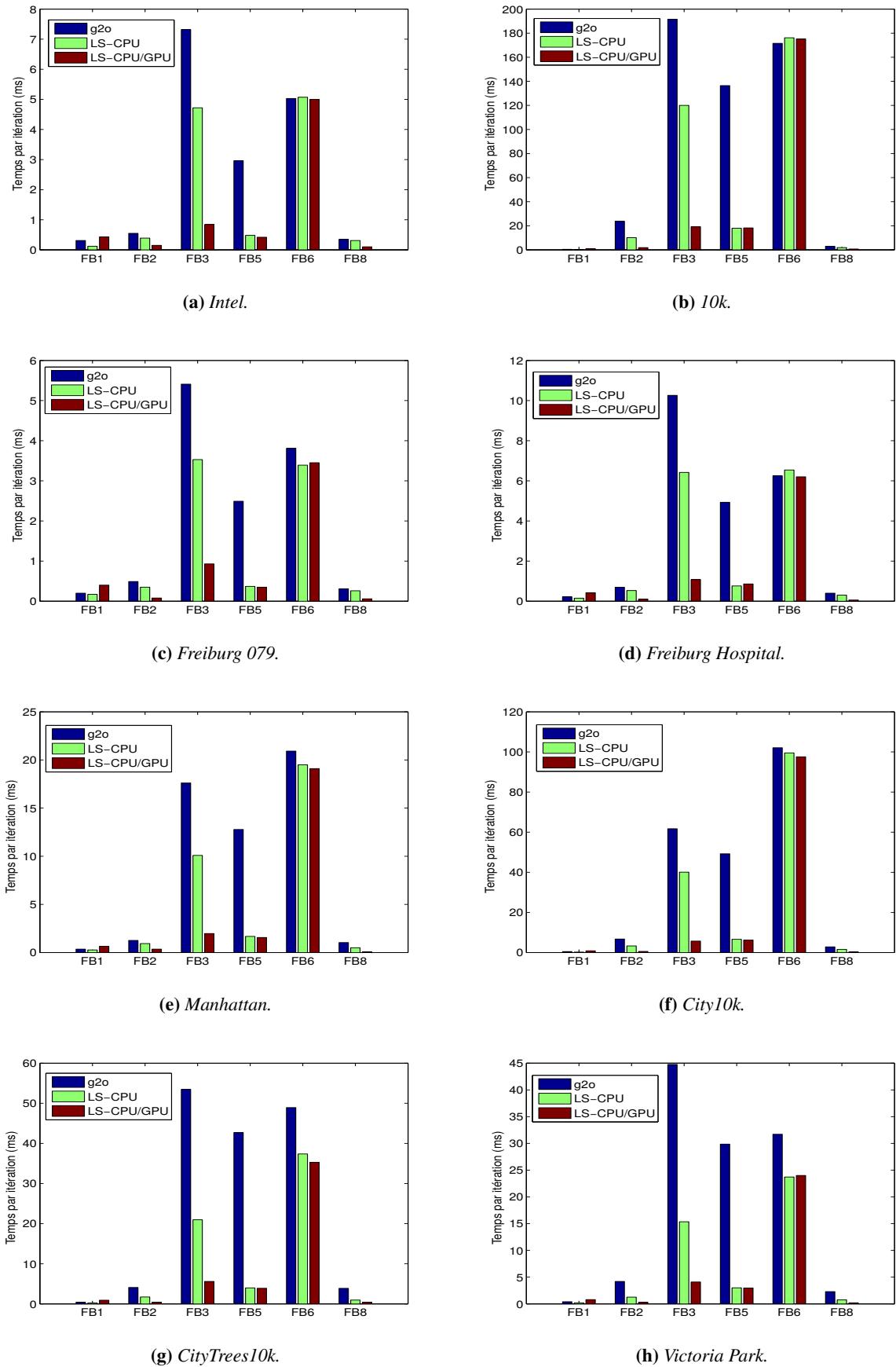
**FIGURE 4.16 – Comparaison des temps moyens par étape des blocs fonctionnels avec complément de Schur.**

4.5.3 Optimisation de graphe en mode incrémental

Pour l'évaluation en mode incrémental, l'optimisation de graphe a été effectuée toutes les 10 étapes. Le nombre d'itérations de Gauss-Newton a été fixé à une itération par étape. La figure 4.17 détaille les temps moyens par étape des blocs fonctionnels. Notons avant tout que FB1 est exécuté par le GPU. Donc, son temps augmente relativement sans pour autant être contraignant par rapport aux autres blocs. FB5 est optimisé de par l'optimisation de la représentation mémoire présentée dans le chapitre 3. Quant à FB6 (Résolution du système), il n'est pas parallélisé. On obtient pratiquement les mêmes temps pour les trois implantations du moment qu'on utilise globalement le même solveur CSparse.

Pour l'implantation hétérogène de LS (LS-CPU/GPU), les blocs fonctionnels sont en moyenne 2.5 fois plus rapide que l'implantation homogène de LS (LS-CPU). Le bloc FB3 est jusqu'à 5 fois plus rapide grâce à l'accélération sur GPU. L'exécution de celui-ci est même 10 fois plus rapide en comparaison avec g^2o . Par ailleurs, nous présentons dans le tableau 4.4 les temps d'optimisation cumulatifs pour chaque jeu de données. En moyenne, LS-CPU/GPU améliore le temps de calcul d'un facteur 1.37 par rapport à LS-CPU. Cela donne une accélération moyenne de 2 fois par rapport à la version mono-cœur de LS. D'autre part, notre implantation hétérogène (LS-CPU/GPU) est jusqu'à 3.88 fois plus rapide que g^2o .

La figure 4.16 donne les temps moyens des blocs fonctionnels lorsque le complément de Schur est utilisé pour les graphes de Victoria Park et CityTrees10k. Le complément de Schur dans les implantations de LS est approximativement 7 fois plus rapide que dans g^2o . Cependant, les temps cumulatifs, résumés dans le tableau 4.5, augmentent considérablement à cause de l'augmentation importante du temps de FB6.

**FIGURE 4.17 – Comparaison des temps moyens par étape des blocs fonctionnels.**

4.6 Conclusion

Nous avons présenté, dans ce chapitre, une étude d'implantation du GraphSLAM sur architectures hétérogènes à base de GPU. Dans l'optique d'embarquer le GraphSLAM sur des architectures embarquées bas-coût. Cette étude a exploré les possibilités qu'offre une architecture hétérogène, disposant d'un GPU comme coprocesseur, pour atténuer la complexité de calcul du GraphSLAM. Comme architecture d'évaluation, notre choix s'est porté sur un système sur puce destiné, en particulier, à la téléphonie mobile, à savoir le Tegra K1.

Notre travail vise particulièrement le GraphSLAM incrémental où la correction de la trajectoire et la carte est effectuée périodiquement, voire à la réception de nouvelles mesures capteurs. Une analyse incrémentale des temps de calcul nous a permis de comprendre l'impact de la croissance continue de la taille du graphe sur les performances temporelles. En se basant sur cette analyse, nous avons proposé un modèle d'implantation du GraphSLAM adéquat aux architectures hétérogènes à base de GPU. Dans un contexte d'adéquation algorithme-architecture, ce modèle répartit les blocs fonctionnels sur les deux processeurs (CPU et GPU) en minimisant les transferts entre ceux-ci. Par ailleurs, ce modèle CPU/GPU préserve et tire profit de la représentation mémoire présentée dans le chapitre 3.

Nous avons montré, à travers les évaluations sur le TK1, l'avantage du modèle CPU/GPU par rapport au modèle homogène exploitant uniquement le CPU. Nous réalisons, en moyenne, une accélération de 1.37 et 2 par rapport, respectivement, au modèle homogène multi-coeur et mono-coeur. Le modèle CPU/GPU est jusqu'à 4 fois plus rapide qu'une implantation multi-coeur de g^2o . Il est à noter, cependant, que ces facteurs de gain dépendent de la structure du graphe. Plus celle-ci est dense, plus le gain apporté par le modèle CPU/GPU est moins important à cause de la résolution du système linéaire qui est effectuée sur CPU. Si le graphe est épars, la construction du système linéaire peut être aussi importante que sa résolution. Le GPU peut accélérer cette construction jusqu'à 5 fois par rapport au modèle homogène multi-coeur.

Le modèle proposé peut être utilisé pour planter des problèmes pouvant être modélisés à l'aide d'un graphe et formulés par un problème de moindres carrés. Bien que l'étude se soit focalisée sur le GraphSLAM, le modèle CPU/GPU peut être adapté à l'algorithme d'ajustement de faisceaux (Bundle Adjustment). Dans le même contexte, ce dernier cherche à minimiser l'erreur entre les observations réelles et les mesures prédites de n amers observés depuis un ou plusieurs capteurs.

Finalement, il est à souligner que l'objectif de cette étude était avant tout de montrer les possibilités qu'offre une architecture hétérogène de type TK1, pour accélérer des algorithmes basés graphe. Les évaluations effectuées dans cette étude ne prennent pas en compte la fréquence et le nombre de coeurs des calculateurs. Les performances temporelles d'implantations hétérogènes dépendront de l'architecture utilisée. Il est évident que le nombre de coeurs parallèles, les ressources mémoire et leurs fréquences auront un impact important. Néanmoins, nous pouvons conclure que les GPUs donnent globalement de meilleures performances par rapport aux CPUs, en particulier pour construire le système linéaire.

Chapitre 5

Étude et implantation sur architectures hétérogènes à base de FPGA

Sommaire

5.1	Introduction	118
5.2	Architecture et optimisation de ressources	118
5.2.1	SLAM 2D	119
5.2.2	Calcul analytique des Jacobiennes	119
5.2.3	Unification des calculs	120
5.2.4	Incrémentalité	120
5.2.5	Calcul trigonométrique sur CPU	120
5.2.6	Calcul flottant	120
5.2.7	Modèle d'implantation HPS-FPGA	121
5.3	Approche OpenCL	121
5.3.1	Mémoire partagée	123
5.3.2	Ressources FPGA	124
5.3.3	Évaluation temporelle	124
5.4	Approche LegUp	126
5.4.1	Architecture	127
5.4.2	Évaluation temporelle	128
5.5	Conclusion	131

5.1 Introduction

Dans ce chapitre, nous abordons l'expérimentation d'une architecture hétérogène à base de FPGA pour l'implantation du GraphSLAM. Pour cela, nous avons utilisé la plateforme DE1-SoC décrite dans le chapitre 2. Nous présentons, à travers cette étude, les résultats d'une expérimentation pratique d'utilisation d'un FPGA comme coprocesseur d'un processeur principal. Les ressources limitées, en éléments logiques et blocs mémoire, sur une architecture bas-coût telle que le DE1-SoC nécessitent une forte optimisation des ressources ainsi qu'une réflexion profonde dans l'implantation de l'algorithme. D'autre part, ce travail a pour objectif d'investiguer l'adéquation des architectures hétérogènes à base de FPGA pour l'implantation du GraphSLAM.

Les langages de description matérielle HDL (Hardware Description Language) tels que VHDL et Verilog sont des points d'entrées pour la conception de systèmes matériels à base de FPGA. On distingue deux types de descriptions de circuits intégrés : description manuelle et automatisée. La description manuelle se base sur l'expertise technique humaine qui mène à des architectures fortement optimisées.

Bien qu'elle soit intéressante pour acquérir de l'expertise et mieux exploiter les performances d'implantation d'un algorithme sur un FPGA, la description manuelle présente des inconvénients. Elle nécessite souvent un temps relativement important pour le développement, la vérification et surtout la maintenance. En revanche, les outils de synthèse de haut niveau (génération de descriptions HDL) ont été développés pour faire face aux inconvénients cités auparavant tout en préservant les langages HDL. Les outils de synthèse de haut niveau (HLS), présentés dans le chapitre 2, permettent de gagner énormément de temps en comparaison avec la description manuelle. Par ailleurs, la synthèse de haut niveau permet d'explorer automatiquement des solutions difficilement réalisables à la main. Cependant, ces outils n'ont pas que des avantages. Leur inconvénient majeur réside dans le fait qu'ils consomment beaucoup de ressources en termes d'éléments logiques et blocs mémoire. De par leur haut niveau d'abstraction, l'optimisation des ressources nécessite une grande expertise humaine.

Dans ce travail, nous avons opté pour les outils de synthèse de haut niveau. En particulier, nous avons choisi d'utiliser le standard OpenCL et LegUp. Ceux-ci permettent d'effectuer conjointement une implantation logicielle/matérielle dans le même flux de conception. La synthèse de haut-niveau permet de prototyper de premières instances architecturales, et de vérifier certaines propriétés avec un temps de développement relativement moins long que la description manuelle.

Nous présentons, dans la suite, le modèle d'implantation CPU-FPGA. Nous discuterons ensuite les détails d'implantations basées OpenCL et LegUp en présentant les différents résultats.

5.2 Architecture et optimisation de ressources

Il s'agit dans cette section d'expliquer comment nous avons réparti les différents blocs fonctionnels, ainsi que la structure de données sur les deux parties : processeur HPS et la fabrique FPGA (éléments

logiques). Bien que le FPGA puisse jouer un rôle de coprocesseur comme le GPU, le modèle hétérogène proposé pour les architectures à base de GPU ne peut pas être adopté directement pour les architectures à base de FPGA. Cela est dû au fait que la nature des ressources de calcul et de mémoire ainsi que leur exploitation sont différentes. Par conséquent, il est important d'adapter l'implantation en fonction des ressources de l'architecture à disposition. D'autre part, grâce aux bus interconnectant le FPGA au HPS (FPGA-to-HPS bridge), le FPGA peut accéder directement à la mémoire HPS. Pour OpenCL, Il est plus intéressant que les kernels accèdent à la mémoire partagée (en mémoire HPS) au lieu de la mémoire FPGA [Altera, 2016]. La mémoire FPGA est accessible avec une grande bande passante. Cependant, les opérations de lecture/écriture effectuées par le HPS sont très lentes car celles-ci n'utilisent pas le contrôleur DMA (Direct Memory Access). La mémoire FPGA est réservée pour passer des données temporaires entre briques calculatoires. Il serait donc plus intéressant en termes de performances que les données soient mises en mémoire HPS.

Le bloc de construction du système (FB3) est important en termes de calcul. La linéarisation des erreurs et le calcul des termes ($H_{ij}, H_i, H_j, s_i, s_j$) sont adaptés pour un calcul parallèle sur FPGA (Algorithm 5). Cependant, compte tenu des ressources disponibles sur le DE1-SoC, il nous a été difficile d'implanter puis dupliquer la brique entière de FB3. De ce fait, nous avons été amenés à apporter un certain nombre d'optimisations par rapport aux implantations logicielles sur CPU et GPU. Nous détaillons, dans ce qui suit, ces optimisations.

5.2.1 SLAM 2D

Nous rappelons, d'abord, que pour les implantations LS et IDS, nous n'avons pas introduit d'optimisations bas niveau ou dépendant d'une paramétrisation donnée. Nous avons gardé la même granularité des données que g^2o . Les opérations arithmétiques sont effectuées entre de petites matrices (3x3,3x2,...) dont les dimensions dépendent des degrés de liberté des nœuds impliqués. L'objectif était de permettre des comparaisons équitables avec l'état de l'art (g^2o et iSAM). Nos implantations sont facilement adaptables pour le SLAM 3D. En revanche, pour planter FB3 sur FPGA, nous nous focalisons sur un type de SLAM bien précis. L'implantation de FB3 sur FPGA concerne le SLAM 2D. Le robot se déplace sur un plan 2D. Les amers, quant à eux, sont représentés par des points 2D. L'idée est de permettre des optimisations bas niveau ou très personnalisées pour optimiser l'utilisation des ressources logiques et améliorer les performances temporelles.

5.2.2 Calcul analytique des Jacobiennes

Pour une meilleure optimisation des ressources logiques du FPGA, nous avons utilisé les fonctions analytiques des Jacobiennes (section 1.4.5). Le passage aux fonctions analytiques améliore relativement le temps de linéarisation. Il est à noter qu'en mode incrémental, nous n'avons pas constaté de différence significative en termes de précision entre l'approximation numérique et les fonctions analytiques. En mode batch, par contre, il peut y avoir relativement une différence en termes de précision de localisation et de cartographie.

5.2.3 Unification des calculs

Dans le souci de minimiser le nombre d'éléments logiques sur le FPGA, nous avons également unifié les calculs relatifs aux deux types d'arêtes (mouvement et observation). En effet, les fonctions analytiques des dérivées et le calcul des termes ($H_{ij}, H_i, H_j, s_i, s_j$) des deux types peuvent être agrégées dans une même brique. L'idée est d'exploiter les ressemblances des termes pour s'arranger à ce qu'on ait une même brique de calcul dont les entrées dépendent du type d'arête.

5.2.4 Incrémentalité

La linéarisation a pour but de calculer la matrice d'information (H) et le vecteur d'information (b) constituant le système linéaire. Le fait d'utiliser les fonctions analytiques des dérivées permet d'identifier des termes qui seront tout le temps nuls, ou encore des termes qui ne sont modifiés qu'une seule fois. Ces derniers sont alors calculés par le HPS lors de l'insertion de l'arête. Cela réduit significativement l'usage des ressources FPGA.

5.2.5 Calcul trigonométrique sur CPU

L'implantation d'opérateurs trigonométriques, tels que le sinus et le cosinus sur FPGA, peut avoir un impact très négatif. En effet, l'usage de modules IP pour le calcul flottant du sinus/cosinus fait que les ressources consommées augmentent et limitent la duplication des briques. Par ailleurs, nous avons aussi expérimenté une fonction tabulée du sinus (LUT). Cette solution, bien qu'elle requière moins de ressources que la première, peut causer une importante imprécision dans le calcul. La qualité de correction de l'état du système (trajectoire et carte) est très sensible aux matrices de variances-covariances caractérisant le bruit sur les mesures capteurs. Ces matrices de variances-covariances sont représentées en flottant contrairement à la fonction tabulée du sinus qui est une une représentation en virgule fixe. En effet, même avec une précision de 10^{-5} pour le sinus¹, le phénomène d'accumulation d'erreurs (bruit de calcul) au fil du temps fait que le calcul de la trajectoire dérive. Pour ces raisons, il est plus judicieux de tirer profit du HPS pour effectuer le calcul trigonométrique. Plus concrètement, pour chaque arête dans le graphe, le HPS calcule le sinus et/ou cosinus des quantités nécessaires. Le FPGA se charge, alors, de récupérer les résultats du calcul trigonométrique avant de s'en servir pour calculer les erreurs et les termes ($H_{ij}, H_i, H_j, s_i, s_j$).

5.2.6 Calcul flottant

De par la nature des opérations arithmétiques impliquées dans le GraphSLAM, et de par son caractère itératif et incrémental, les erreurs de calcul ont tendance à s'accumuler. Le GraphSLAM est donc très sensible au bruit calculatoire. Les implantations de l'état de l'art (g^2o [[Kummerle et al., 2011](#)], iSAM

1. Une LUT à 2^{14} entrées, la valeur récupérée est corrigée par interpolation.

[Kaess et al., 2008, 2012]) utilisent toutes le format double-précision (64 bits) pour les données et les calculs. Néanmoins, l'utilisation de ce format fait que FB3 consomme énormément de ressources. Pour pallier à cela, nous sommes passés à la représentation en simple-précision (32 bits). Il est à noter que pour maîtriser le bruit de calcul du à la représentation en simple-précision, nous avons paramétré les matrices de variances-covariances pour faire converger l'optimisation de graphe. Le passage en simple-précision a permis de réduire la taille de la surface consommée d'un facteur deux.

5.2.7 Modèle d'implantation HPS-FPGA

Indépendamment de l'outil de synthèse utilisé (OpenCL ou LegUp), le modèle d'implantation HPS-FPGA est schématisé dans la figure 5.1. Globalement, le FPGA traite FB3 et sert d'accélérateur. Les blocs FB2 et FB3 ont été fusionnés dans une même brique (FB2-FB3) pour optimiser les ressources et diminuer les accès mémoire. Le HPS est en charge du calcul des quantités trigonométriques pour toutes les arêtes (T). L'accélérateur FB2-FB3 calcule les erreurs des arêtes et les termes ($H_{ij}, H_i, H_j, s_i, s_j$) et stocke les résultats en mémoire HPS. Ce dernier récupère ces termes pour calculer les blocs diagonaux de la matrice et vecteur d'information. Dans la suite, nous allons nous focaliser sur les détails d'implantation de FB2-FB3. Nous discuterons l'architecture générée par l'utilisation d'OpenCL et LegUp. La manière dont les données T et ($H_{ij}, H_i, H_j, s_i, s_j$) sont récupérées sera également abordée.

5.3 Approche OpenCL

Comme présenté dans le chapitre 2, le paradigme OpenCL a été adopté pour la conception d'architectures FPGA. OpenCL offre intrinsèquement la capacité de décrire des architectures d'algorithmes parallèles. Il permet de faire des descriptions à un niveau d'abstraction plus haut que la description des langages HDL. Cette approche, depuis son émergence, est en pleine expansion. Nous avons utilisé cette approche pour générer l'architecture du modèle illustré dans la figure 5.1.

Dans l'approche OpenCL, le FPGA est considéré comme un accélérateur pour le processeur (comme le GPU). OpenCL pour FPGA reprend les modèles de programmation, de mémoire et d'architecture qui permettent à l'utilisateur d'avoir des implantations pour architectures hétérogènes à base de GPU. Le GPU et le FPGA sont architecturalement différents. L'architecture de l'accélérateur sur FPGA n'est pas figée contrairement au GPU. Ce dernier dispose de plusieurs multiprocesseurs de flux. En revanche, le compilateur AOCL (Altera Offline Compiler) décrit une architecture matérielle spécifique à chaque kernel en entrée.

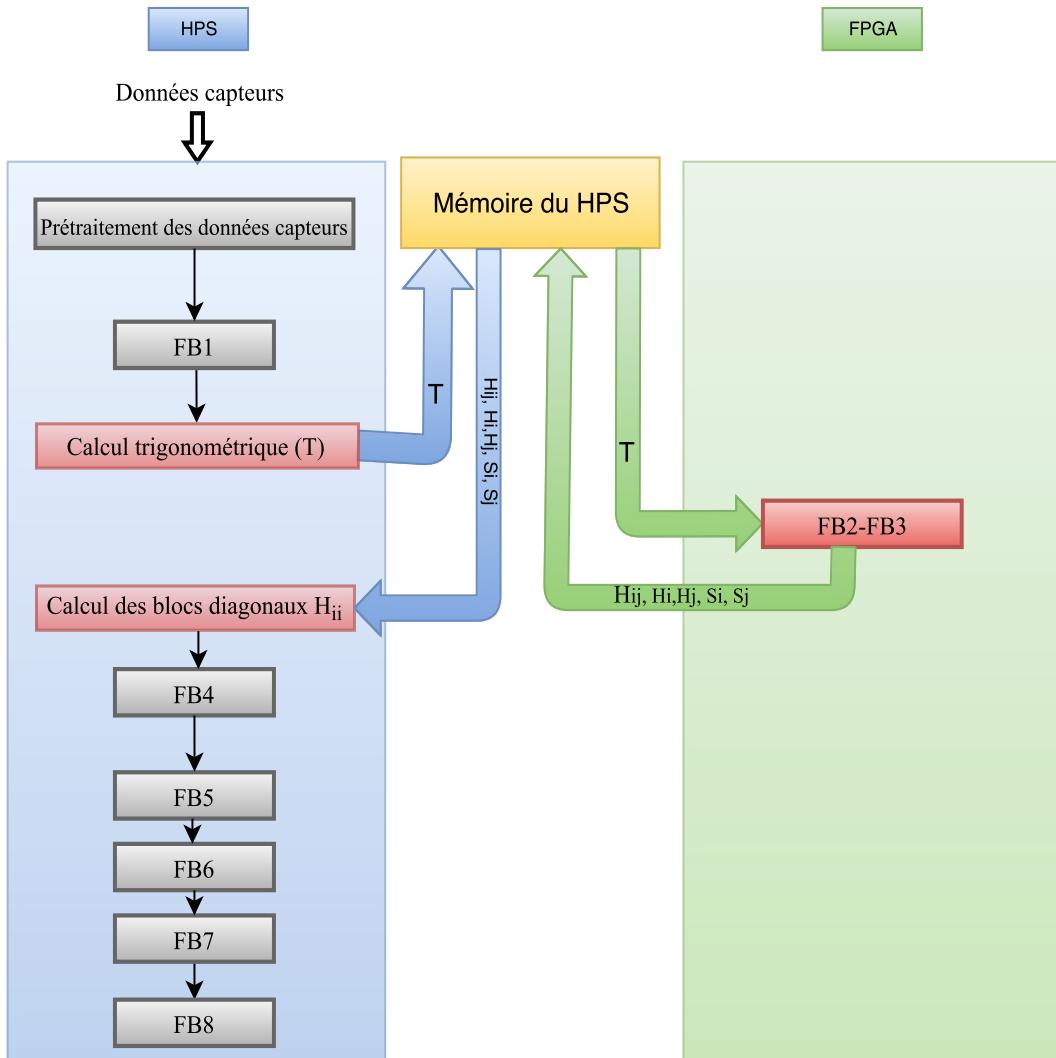


FIGURE 5.1 – Modèle d’implantation CPU (HPS)/FPGA.

La différence majeure entre l’exécution d’un kernel sur GPU et sur FPGA réside dans la manière dans le parallélisme est traité. Les GPUs sont des processeurs SIMD (Single-Instruction Multiple-Data) où un groupe de cœurs (généralement des groupes de 32) effectue la même opération sur plusieurs données. En revanche, sur le FPGA, un kernel est synthétisé en un circuit dédié. Les FPGAs exploitent, en premier lieu, le parallélisme pipeliné à l’intérieur d’un kernel. En d’autres termes, une instruction est divisée en plusieurs étages. Ces derniers sont affectés, à un instant donné, à plusieurs threads. Le pipeline masque considérablement les latences d’accès à la mémoire permettant ainsi d’augmenter le débit par cycle (flot de données en sortie). Par ailleurs, en fonction de la disponibilité des ressources sur FPGA, il est possible de dupliquer les kernels pour améliorer les performances. Malheureusement, ce dernier cas n’a pas été évalué dans notre étude par manque de ressources sur l’architecture d’évaluation (DE1-SoC).

Finalement, l’étude d’une implantation OpenCL a également pour objectif d’évaluer l’apport d’une architecture pipelinée dans l’accélération du GraphSLAM.

5.3.1 Mémoire partagée

L'architecture d'évaluation (DE1-SoC) est dotée d'une mémoire DDRAM de 1Go du coté HPS et d'une autre de 64Mo du coté FPGA. La fonctionnalité zéro-copie permet au FPGA d'accéder à la mémoire HPS. De même que sur les architectures hétérogènes à base de GPU, l'utilisation de la fonctionnalité zéro-copie dégrade énormément les performances. Cela est également expliqué par le fait que les données sont stockées dans une zone mémoire non paginée. Les données ne sont, donc, pas mises en mémoire cache par le processeur. Cela augmente les temps d'accès mémoire du coté HPS.

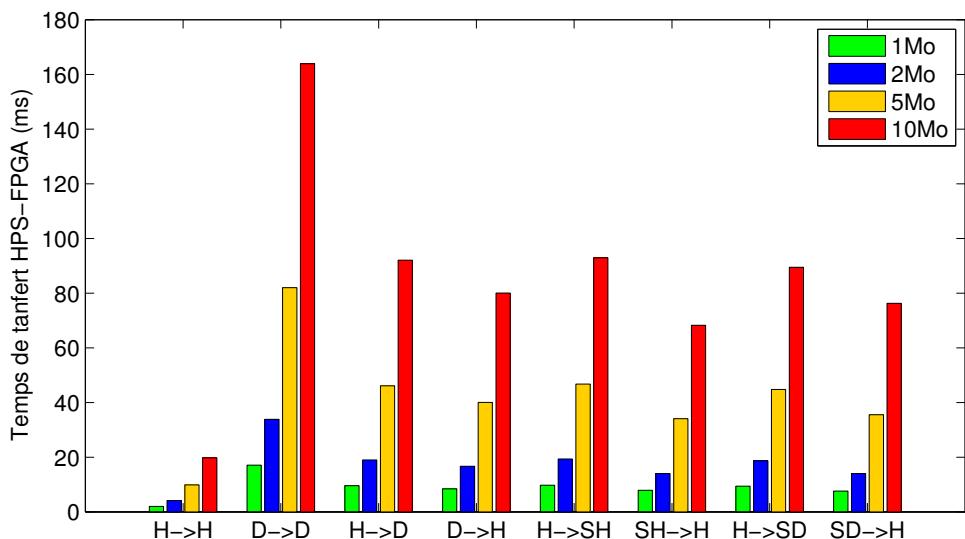


FIGURE 5.2 – Temps de transfert HPS-FPGA. D : Device (FPGA), H : Host (HPS), SH : Shared Host (zone non paginée en mémoire HPS), SD : Shared Device (zone non paginée mappée dans l'espace d'adressage du FPGA).

Pour avoir de meilleures performances, il est alors inévitable d'effectuer des transferts de données entre le HPS et le FPGA. La figure 5.2 montre les temps nécessaires pour transférer des zones mémoire de 1, 2, 5 et 10 Mo. Les temps diffèrent en fonction de la source et destination des données, mais également selon que les zones mémoire sont paginées ou non dans le cas où elle sont allouées en mémoire HPS. Le transfert se fait par de 32 bits ; par conséquent le temps de transfert est linéaire en nombre d'octets. Le transfert le plus important est le transfert mémoire FPGA à mémoire FPGA. On constate, par ailleurs, que le transfert le plus optimal est entre une zone mémoire paginée et une autre non paginée. En effet, le transfert s'effectue entre deux zones mémoire HPS. Néanmoins, le fait de transférer par mot de 4 octets augmente le temps par rapport au transfert paginée-paginée. Notons, d'autre part, que le transfert entre mémoire FPGA et mémoire HPS nécessite approximativement le même temps qu'un transfert entre mémoire paginée et non paginée. En réalité, pour la plateforme utilisée (DE1-SoC) et son Framework correspondant, OpenCL alloue systématiquement de l'espace mémoire FPGA en mémoire HPS plutôt qu'en mémoire FPGA (64Mo). Nous prenons cela comme une première optimisation. En effet, il est plus efficace d'accéder à la mémoire HPS qu'à la mémoire

FPGA pour les kernels. Cela s'explique par le fait le contrôleur mémoire du côté HPS est un IP hardware à 800MHz, alors que le contrôleur mémoire du côté FPGA est cadencé à la fréquence maximale de l'architecture, c'est-à-dire avoisinant les 100Mhz.

En conclusion, partant du modèle d'implantation de la figure 5.1 et pour maximiser les performances, les transferts mémoire des quantités trigonométriques (T) et des termes ($H_{ij}, H_i, H_j, s_i, s_j$) sont réalisés via quatre tampons mémoire. Le HPS écrit et lit toujours dans un tampon en mémoire paginée. Le FPGA, quant à lui, lit puis écrit dans un tampon en mémoire non paginée. Les quatre tampons sont alloués en mémoire HPS.

5.3.2 Ressources FPGA

Les architectures générées par OpenCL sont très consommatoires, voire contraignantes en termes de ressources. Les ressources utilisées pour produire l'architecture de la brique FB2-FB3 sont résumées dans le tableau 5.1. La brique FB2-FB3 prend 89% de la logique (ALMs) disponible sur le FPGA. De même, 49% des blocs de RAMs ont été utilisés. Les architecture fortement pipelinées consomment beaucoup d'éléments logiques et de blocs mémoire pour faire face aux aléas structurels et de données.

TABLE 5.1 – Ressources consommées de l'architecture produite par OpenCL.

Ressources	Pourcentage de ressources consommées	Taux (%)
ALMs	28,640	89
Registres	53160	
onchip memory	748,896	18
Blocs de RAMs	196	49
DSPs	61	70

5.3.3 Évaluation temporelle

Nous rappelons que les blocs de calcul des erreurs (FB2) et de construction du système (FB3) ont été fusionnés puis divisés en trois sous-blocs (Fig. 5.1), à savoir le calcul trigonométrique, FB2-FB3 (calcul d'erreurs et linéarisation) et enfin le calcul du vecteur d'information et des blocs diagonaux de la matrice d'information. Les évaluations temporelles ont été effectuées, en mode incrémental, sur le graphe de CityTrees10K. L'optimisation de graphe a été effectuée toutes les 100 étapes. Nous utilisons encore la métrique TPE pour comparer les différentes solutions et configurations.

Dans la figure 5.3, nous présentons l'évolution du TPE de FB2-FB3. Si l'on ne tient pas compte du temps de transfert dans l'implantation hétérogène, le TPE obtenu sur FPGA devient, après un certain nombre d'étapes, 1.83 fois plus faible que celui fourni par la version HPS mono-cœur, et seulement 1,13 fois plus faible que celui de la version multi-cœur. En considérant le temps de transfert des quantités trigonométriques (T) et les termes ($H_{ij}, H_i, H_j, s_i, s_j$), le TPE réalisé grâce au FPGA est remis en cause par le temps important du transfert. En effet, le TPE du FPGA devient jusqu'à 4,12

fois plus grand que celui de la version HPS mono-cœur, et approximativement 7 fois plus grand par rapport à la version parallélisée. Pour réduire encore le TPE du FPGA, il est possible d'envoyer les données trigonométriques par tranches en parallèle avec le calcul. Cela masque considérablement le temps d'envoi du moment que le calcul des quantités trigonométriques est plus lent que le transfert. Cependant, il est difficile, ou encore non efficace de transférer par tranches les quantités (H_{ij} , H_i , H_j , s_i , s_j) de l'espace mémoire FPGA vers l'espace mémoire HPS. Au niveau de l'accélérateur FB2-FB3 sur FPGA, les arêtes peuvent ne pas être traitées dans l'ordre, ce qui complique l'envoi des données par tranches. A cela s'ajoute le fait que le calcul sur FPGA de FB2-FB3 nécessite moins de temps que le transfert (jusqu'à 3 fois moins faible). Cela rend l'envoi parallèle moins intéressant.

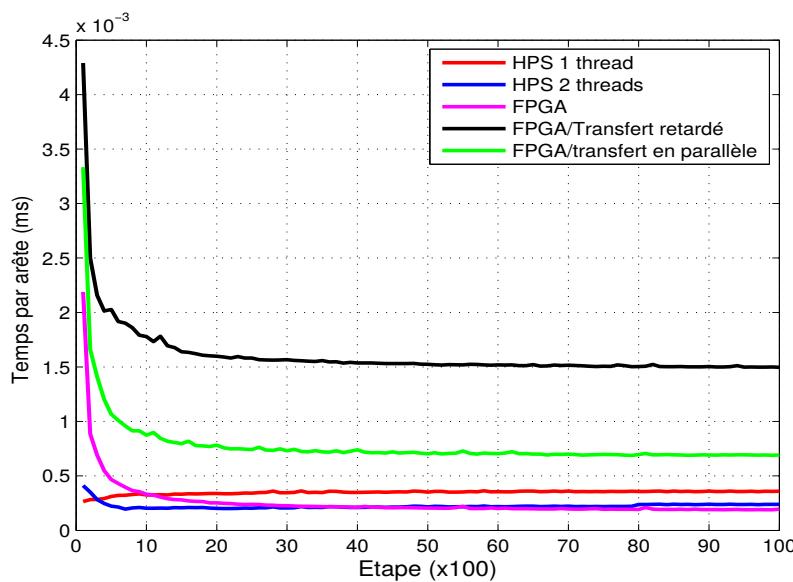


FIGURE 5.3 – Comparaison des TPEs.

Le tableau 5.2 résume les temps cumulatifs des sous-blocs de FB3. Globalement, une implantation sur le HPS s'avère plus rentable en termes de performances temporelles compte tenu du temps de transfert contraignant entre l'espace mémoire HPS et l'espace mémoire FPGA. Nous rappelons que ces deux espaces mémoire sont alloués dans la mémoire HPS (Fig. 5.1).

TABLE 5.2 – Temps cumulatifs des sous-blocs de FB2 et FB3.

	Calcul trigo. (T)	Transfert de T	FB2-FB3	Transfert de H	Calcul des blocs diag.	Total (ms)
HPS	431.17	-	260.11	-	435.06	1126.34
HPS OpenMP	323.03	-	156.06	-	304.54	783.99
FPGA-Trans. reatrdé	335.03	603.11	151.97	363.97	353.62	1596.61
FPGA-Trans. en parallèle	345.03	3.27	150.88	366.23	352.89	1218.30

Malgré l'accélération relative du sous-bloc FB2-FB3, les performances totales du bloc FB3 se dégradent à cause du transfert de données. Néanmoins, indépendamment du transfert, il est important de remarquer le gain significatif obtenu en implantant FB2-FB3 sur FPGA. En effet, malgré la fréquence limitée du FPGA (50MHz à 100MHz contre 800MHz pour le HPS), le FPGA arrive à surpasser

ser le HPS en termes de calcul. Notons, par ailleurs, qu'une seule brique de FB2-FB3 a été utilisée par manque de ressources matérielles. Pour une comparaison plus équitable, nous montrons dans la figure 5.4 l'évolution du CPE (Cycles machine Par Élément de graphe).

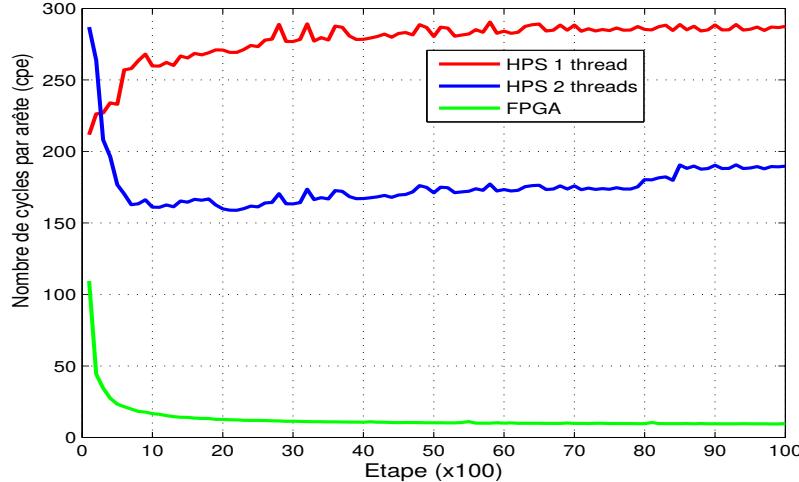


FIGURE 5.4 – Évolution du CPE.

Le CPE donne le nombre de cycles nécessaires pour traiter un élément de graphe. Le CPE est donné par la formule suivante :

$CPE = (T/n) * f$, où n est le nombre d'éléments de graphe (arêtes en l'occurrence) et f est la fréquence du calculateur utilisé. On constate qu'indépendamment de la fréquence, le traitement d'une arête requiert 29 fois moins de cycles par rapport à la version mono-cœur et 17 fois, comparé à la version parallélisée sur HPS.

Par ailleurs, alors que le CPE sur HPS augmente relativement avec le temps, le CPE sur FPGA a tendance à diminuer au fur et à mesure que le nombre d'arêtes croît. Sur la fin de la trajectoire, le CPE est estimé à 10 cycles. Cela revient principalement à l'architecture pipelinée de la brique. Les nombreux étages de calcul et d'accès mémoire (load/store) qui y sont implantés permettent de masquer, avec la croissance du nombre d'arêtes, les latences de lecture/écriture en mémoire. Cela conduit à une forte croissance du débit de sortie.

5.4 Approche LegUp

LegUp est un outil de synthèse qui s'appuie sur une approche de synthèse et un flux de conception différents de ceux de OpenCL. LegUp, dans sa version 4.0, accepte en entrée un programme C écrit selon la norme ANSI C. Pour utiliser le FPGA, LegUp fournit deux flux de conception différents, une conception purement matérielle et une autre hybride ou hétérogène. Dans la première, tout le programme C est synthétisé sur FPGA sans aucun processeur embarqué. Concernant la conception

hybride, LegUp implante un modèle d'architecture sur laquelle une partie du code est exécutée sur un processeur embarqué, et le reste est synthétisé en un accélérateur sur FPGA.

Si la plateforme ne dispose que d'une fabrique FPGA (comme la plateforme DE2-115 de Altera, par exemple), il est toujours possible de concevoir une architecture hétérogène moyennant LegUp. Celui-ci permet, en effet, d'instancier un processeur TigerMIPS (softcore) pour l'implanter sur le FPGA et l'interfacer aux briques accélératrices. Mais, dans la version actuelle (4.0), LegUp fournit un support pour les plateformes hétérogènes SOC à base de FPGA (SoCKit et DE1-SoC). Il est donc possible d'intégrer le HPS dans le flux de conception. Cela nous a permis d'instancier l'architecture décrite dans la figure 5.1, sur le DE1-SoC où la brique FB2-FB3 a été déportée sur le FPGA.

Nous avons vu que OpenCL repose principalement sur un parallélisme orienté pipeline (avec duplication de briques calculatoires). Bien que LegUp apporte des optimisations automatiques telles que le déroulage et le pipelinage de boucles, il reste très proche de la conception manuelle, qui se base, entre autres, sur la duplication de blocs calculatoires pour accélérer les traitements. Il est, maintenant, intéressant de voir l'impact d'utiliser plusieurs briques parallèles sur les performances.

5.4.1 Architecture

La solution purement matérielle consiste à mettre tous les blocs fonctionnels sur FPGA. Cette première solution est très contraignante en termes de ressources notamment pour implanter le solveur épars de Cholesky. Nous avons opté pour la solution hétérogène (Fig. 5.1). D'autre part, nous n'avons utilisé que le HPS comme processeur embarqué. Le processeur TigerMIPS a été utilisé dans une autre étape pour valider et préparer une implantation synthétisable par LegUp.

De même qu'avec OpenCL, il s'agit d'implanter la brique FB2-FB3 sur FPGA. LegUp offre, par ailleurs, la possibilité d'instancier plusieurs accélérateurs opérant en parallèle. Cela est possible grâce à l'utilisation de Pthreads ou OpenMP. Chaque thread est synthétisé en un accélérateur. Le nombre d'accélérateurs correspond au nombre de threads choisi par l'utilisateur.

Il est à noter que pour le HPS, nous n'avons pas utilisé de système d'exploitation (bare-metal application). LegUp génère automatiquement un BSP (Board Support Package) qui prend en charge l'exécution du programme HPS et la communication avec le FPGA. Par conséquent, il est important de souligner que l'absence de système d'exploitation, et ainsi un gestionnaire de mémoire virtuelle, fait que toutes les données doivent résider en mémoire. Le HPS est utilisé, dans ce cas, comme un puissant microcontrôleur. D'autre part, la parallélisation sur les deux coeurs Cortex-A9 est à la charge de l'utilisateur. Malheureusement, LegUp ne fournit pas encore cette fonctionnalité. Cependant, nous rappelons que l'objectif, ici, est d'analyser l'apport des briques FPGA plutôt que le CPU (HPS).

Pour maximiser les performances, il est intéressant de lancer le calcul trigonométrique et la linéarisation (FB2-FB3) en parallèle. Suivant le flux de conception de LegUp, on crée n threads ; n étant le nombre d'accélérateurs voulu. Ces n accélérateurs sont lancés au démarrage de l'application. Chacun des accélérateurs se charge de traiter une partie des arêtes. Pour synchroniser les échanges entre les

accélérateurs FPGA (threads) et le HPS, on réserve en mémoire pour chaque arête un signal “start”. Celui-ci indique si les données trigonométriques de l’arête en question sont prêtes (déjà préparées par le HPS) à utiliser par l’accélérateur correspondant. Le signal “start” est remis à zéro à la fin du traitement de l’arête en question par l’accélérateur. Chaque accélérateur signale la fin du traitement de sa part d’arêtes au HPS. Celui-ci attend, à son tour, la terminaison des n accélérateurs avant de passer au calcul des blocs diagonaux.

L’architecture produite par LegUp est décrite dans la figure 5.5. La brique FB2-FB3 peut être instanciée plusieurs fois. Le tableau 5.3 détaille l’utilisation des ressources en fonction du nombre d’accélérateurs. Chaque accélérateur est connecté à une interface de synchronisation que l’on appelle wrapper. Celui-ci envoie les données à l’accélérateur matériel puis lance (enable) le calcul sur l’accélérateur. Il véhicule également les données vers la mémoire. Comme les accélérateurs accèdent à la mémoire du HPS, un contrôleur mémoire est nécessaire. Dans cette architecture, la communication entre le HPS et les accélérateurs est assurée par les bus (bridge) HPS-to-FPGA et FPGA-to-HPS.

TABLE 5.3 – Ressources consommées, après synthèse, par l’architecture générée par LegUp.

Ressources	1 Accél.	2 Accél.	3 Accél.	4 Accél.	5 Accél.
Fmax (MHz)	86.01	82.16	82.69	78.12	69.66
ALMs (%)	22	38	55	71	86
Registres	9457	16283	23148	30020	36774
onchip memory (%)	1	1	1	1	1
Blocs de RAMs (%)	2	2	3	3	4
DSPs (%)	5	11	17	23	29

5.4.2 Évaluation temporelle

Pour évaluer l’impact d’augmenter le nombre d’accélérateurs, nous avons synthétisé puis évalué plusieurs instances d’architectures en fonction du nombre d’accélérateurs. Toutes les architectures ont été évaluées avec une fréquence de 50MHz pour le FPGA. Le graphe de CityTrees10K est optimisé toutes les 1000 étapes. Nous avons mesuré, à chaque étape d’optimisation de graphe, le nombre de cycles pour le calcul trigonométrique et le sous-bloc FB2-FB3 représenté par les accélérateurs FPGA. Ces deux calculs sont lancés en parallèle comme expliqué auparavant. Notons que pour mesurer le nombre de cycles, nous avons utilisé une routine logicielle fournie livrée par LegUp. Celle-ci donne un nombre de cycles sur la base d’une fréquence de 800MHz (fréquence du cycle d’accès à la mémoire centrale). Dans la suite des explications, le nombre de cycles est ramené à la base de 50MHz pour une comparaison plus claire.

La figures 5.6 indique le nombre total de cycles correspondant à la brique FB2-FB3 et le calcul trigonométrique qui s’effectuent en parallèle. Si le rajout d’un deuxième accélérateur diminue le nombre de cycles d’un facteur 1,45, le gain apporté par le rajout d’autres accélérateurs est très faible. Les accélérateurs accèdent d’une manière concurrente à la mémoire HPS. L’utilisation de deux accélérateurs est avantageuse en termes d’accélération. Par exemple, alors que le premier accélérateur est

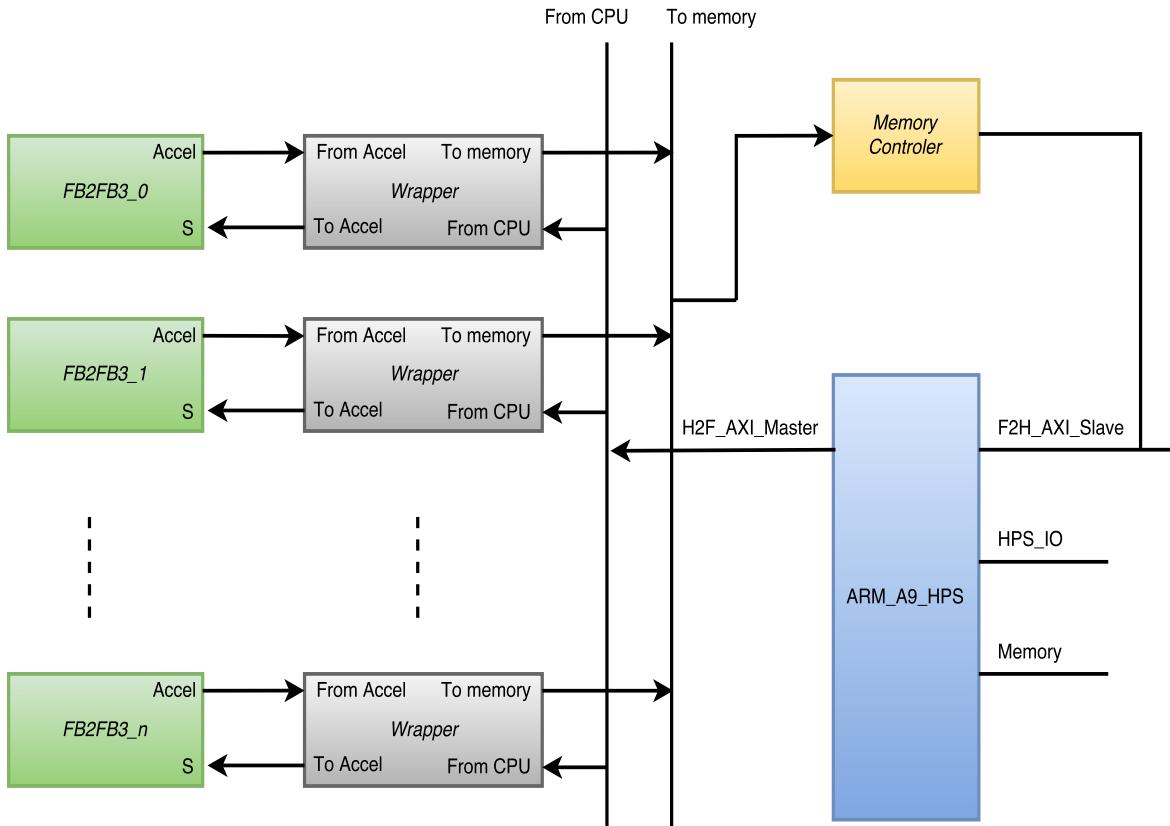


FIGURE 5.5 – Architecture générée par LegUp.

en train d'effectuer une opération flottante, le second peut en profiter pour aller lire ou écrire en mémoire. Cependant, en utilisant plusieurs accélérateurs, le goulot d'étranglement va se situer au niveau des accès mémoire. En effet, les accélérateurs passent plus de temps à attendre la synchronisation d'accès au bus qu'à effectuer un calcul. Cela explique pourquoi l'utilisation de cinq accélérateurs en comparaison avec quatre accélérateurs n'améliore pas les performances d'une manière significative.

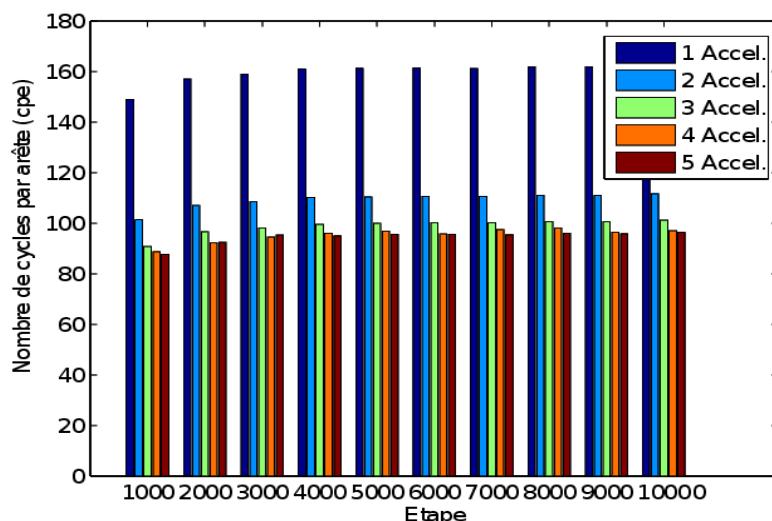


FIGURE 5.6 – Nombre de cycles par étape.

La figure 5.7 donne le nombre de cycles par arête pour chacune des étapes d'optimisation de graphe. Contrairement aux autres architectures (CPU, GPU et FPGA avec OpenCL), on remarque que le CPE de chaque configuration (nombre d'accélérateurs), n'est pas impacté par le nombre d'arêtes. Par exemple, avec cinq accélérateurs, le CPE est pratiquement le même peu importe le nombre d'arêtes (ou l'étape). Le fait de ne pas utiliser de pipeline pour accélérer les boucles, et de ne pas utiliser de système d'exploitation pour le HPS font que le CPE reste très stable au fil du temps.

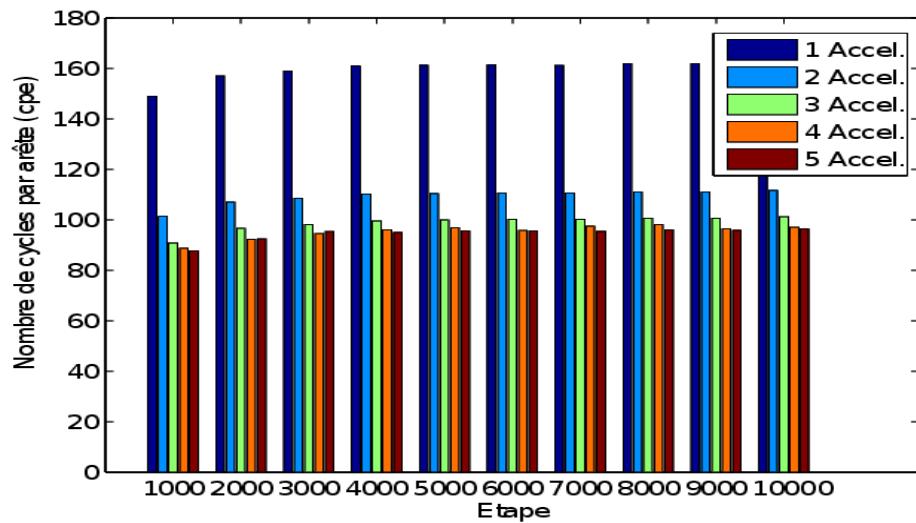


FIGURE 5.7 – Comportement du CPE de la brique FB2-FB3 sur l'architecture produite par LegUp.

5.5 Conclusion

Ce chapitre a présenté une étude d'implantation du GraphSLAM sur une architecture hétérogène à base de FPGA. Nous avons défini un modèle d'implantation CPU-FPGA dans lequel le FPGA se charge de la construction du système linéaire. Celle-ci est caractérisée par un grand nombre de calculs trigonométriques. Nous avons remarqué que l'implantation des routines trigonométriques sur FPGA (à l'aide de LUTs) peut conduire à une dégradation de la précision de localisation et de cartographie. La trajectoire obtenue finit par diverger. Il a été, donc, judicieux d'utiliser le HPS pour effectuer ce type de calculs. Cela implique encore plus de transferts de données en comparaison avec le modèle CPU/GPU. L'étude a fait l'objet de l'utilisation de deux outils de synthèse de haut niveau : OpenCL et LegUp. Celle-ci nous a permis de vérifier certaines propriétés relatives aux pipelines ainsi qu'à la duplication de briques calculatoires sans avoir à décrire l'architecture par un langage HDL.

AOCL (Altera Offline CompiLer) produit des architectures pipelinées permettant de maximiser le débit du flux de données en sortie. Indépendamment de la fréquence machine, nous avons constaté que la brique implantée sur FPGA en utilisant OpenCL permettent une accélération allant jusqu'à 29 fois et 17 fois par rapport, respectivement, aux implantations homogènes mono cœur et multi cœur sur le HPS. Bien que ce type d'architectures accélère relativement le calcul, le temps de transfert de données entre le FPGA et le HPS reste très contraignant. Cela remet en cause le gain apporté par l'accélération sur FPGA.

LegUp a été utilisé pour observer l'impact d'utiliser plusieurs briques parallèles implantées sur FPGA. Nous avons constaté que, contrairement à l'architecture pipelinée, la duplication de briques calculatoires n'améliorait pas d'une manière significative les performances. En augmentant le nombre de briques parallèle, le goulet d'étranglement sera au niveau des accès mémoire. Les briques implantées sur FPGA interagissent énormément avec la mémoire.

Finalement, nous pouvons dire que le modèle CPU-FPGA, avec son instanciation sur une architecture hétérogène bas coût, n'améliore pas les performances globales. Cela est dû principalement au transfert de données prohibitif. D'un autre côté, la densité d'interaction avec la mémoire du GraphSLAM rend la duplication des briques calculatoires inefficace. Il est à souligner que AOCL atténue considérablement le goulet d'étranglement en utilisant des pipelines personnalisés et profonds.

Conclusions et perspectives

Conclusions

L'objectif principal de cette thèse était de réduire la complexité du SLAM par optimisation de graphe. Cette complexité est due à plusieurs paramètres inhérents à l'algorithme lui-même et à la caractéristique de lissage. Pour atténuer cette complexité, nous avons étudié deux aspects : l'organisation des données en mémoire et la répartition des blocs fonctionnels de calcul sur des architectures parallèles hétérogènes dans un contexte d'adéquation algorithme-architecture. Dans cette étude, nous donnons des recommandations sur le choix d'un modèle d'implantation adéquat aux algorithmes de SLAM basés graphe, en particulier le SLAM par optimisation de graphe (GraphSLAM). Les modèles d'implantation sont couplés à une organisation efficace des données en mémoire. Ce travail a donc donné lieu à deux contributions principales.

Le GraphSLAM nécessite une masse de données importante qu'il faut manipuler et explorer, tout au long du processus de calcul. L'organisation des données en mémoire et la stratégie d'y accéder impactent considérablement les performances temporelles. Le défi dans une structure de données orientée graphe est de pouvoir garder trace des connexions constituant le graphe. L'objectif est de faciliter l'accès aux éléments du graphe en limitant en même temps la consommation d'espace mémoire. Nous avons démontré dans cette thèse l'apport qu'une structure de données adéquate peut donner pour accélérer les traitements.

Nous avons proposé deux structures de données : IDS (*Index-based Data Structure*) et LS (*LightSLAM*). IDS utilise une matrice bidimensionnelle pour référencer les éléments non nuls du graphe. Elle offre un compromis entre l'accès direct aux données et l'utilisation d'espace mémoire. L'index a une complexité de stockage mémoire de $O(N^2)$, où N est le nombre de noeuds dans le graphe. IDS ne peut être utilisée pour les graphes contenant un grand nombre de noeuds (plus de 4000 noeuds). Bien qu'elle donne de meilleures performances sur les petits graphes en comparaison avec les implantations de l'état de l'art (g^2o et iSAM), il ne convient pas de l'utiliser pour les graphes basés amers. Cela est dû à la gestion contraignante de l'index quand le nombre de noeuds est très grand.

Pour pallier aux problèmes de IDS, nous avons proposé une seconde représentation mémoire (LS). Celle-ci fournit une structure de données très compacte en termes d'espace mémoire. D'où le nom *Light SLAM* (SLAM léger). LS exploite le caractère épars et incrémental du problème de SLAM. Nous ne stockons en mémoire que les éléments non nuls du graphe et du système linéaire associé. Chaque noeud garde trace de son voisinage. La complexité de stockage mémoire de LS est $O(NV)$, où V est le degré maximal des noeuds ($V \ll N$). La mise à jour de LS s'effectue d'une manière incrémentale ; c'est-à-dire à chaque fois que l'on met à jour le graphe. Bien que la structure de celui-ci évolue avec le déplacement du robot, la mise à jour de LS ne modifie que les noeuds concernés.

Les évaluations temporelles ont montré l'efficacité de LS. Le gain par rapport aux implantations de l'état de l'art (g^2o et iSAM) est très important en mode incrémental. LS est jusqu'à trois fois plus rapide que les implémentations de l'état de l'art (g^2o et iSAM). Cela revient principalement à la mise à jour incrémentale qui permet d'éviter la recherche des voisnages des noeuds à chaque correction de la carte et de la trajectoire.

Par ailleurs, nous avons investigué l'apport des architectures hétérogènes dans l'atténuation de la complexité de calcul du GraphSLAM. Notre étude a porté sur des architectures hétérogènes embarquées. Celles-ci ont la particularité d'intégrer différents types de calculateurs sur la même puce. Notre étude s'est basée sur une analyse incrémentale du temps de calcul ; c'est-à-dire avec l'évolution de la taille du graphe. Dans un contexte d'adéquation algorithme-architecture, nous avons défini deux modèles pour l'implantation du GraphSLAM : CPU/GPU et CPU/FPGA. Notre travail étend les études d'implantation du GraphSLAM de l'état de l'art [Choudhary et al., 2010, Wu et al., 2011, Rodriguez-Losada et al., 2013, Ratter et al., 2013] par :

- L'introduction de l'analyse incrémentale du temps de calcul. Cela permet de comprendre et prévoir le comportement du temps sur l'architecture sélectionnée.
- L'utilisation de la métrique *temps par élément de graphe* (TPE) dans l'adéquation algorithme-architecture.
- L'étude d'une implantation du GraphSLAM sur FPGA en utilisant OpenCL. Cette étude constitue donc un premier travail dont les leçons tirés peuvent être exploitées pour étendre et améliorer le travail effectué.
- L'utilisation de relativement un grand nombre de jeux de données publiques dans le but de fournir, à la communauté scientifique, une évaluation du GraphSLAM sur des architectures embarquées.

Le modèle CPU/GPU vise à exploiter un processeur graphique. Les tâches sont réparties sur les deux processeurs (CPU et GPU) en vue d'accélérer les traitements par rapport au modèle homogène. Celui-ci utilise uniquement le CPU comme calculateur. Sur l'architecture d'évaluation utilisée (Tegra K1), le modèle CPU/GPU permettait un facteur deux d'accélération par rapport à la version mono cœur du modèle homogène. Notre étude a montré, par ailleurs, que les charges des blocs fonctionnels sont variables. Elle dépendent de la structure de l'environnement modélisée par le graphe. Plus celui-ci est épars, plus le modèle CPU/GPU est avantageux en comparaison avec le modèle homogène.

Le modèle CPU/FPGA utilise un FPGA comme un coprocesseur. De par notre expérience avec la plateforme d'évaluation choisie (DE1-SoC), nous avons mis en avant plusieurs paramètres concernant l'implantation du GraphSLAM sur une architecture hétérogène basée FPGA :

- De la même manière que sur un GPU, la construction du système linéaire se prête bien pour une parallélisation sur FPGA
- La construction du système linéaire fait appel au calcul trigonométrique. L'implantation de routines trigonométriques sur FPGA en utilisant des LUTs conduit à des imprécisions de calcul susceptibles de faire diverger la localisation. L'utilisation du processeur embarqué (HPS) s'avère, dans ce cas, avantageuse pour plus de performances en termes de précision de localisation et de cartographie.
- La masse de données importante manipulée dans le GraphSLAM limite l'utilisation de la mémoire on-chip sur FPGA.
- Bien que le FPGA accélère relativement la construction du système, le transfert de données entre le FPGA et le CPU constitue le premier facteur limitant de cette solution.
- Les interactions avec la mémoire sont très denses. Par conséquent, la duplication d'une brique

calculatoire sur FPGA n'améliore pas significativement le temps de calcul. Le goulot d'étranglement va se produire sur les accès mémoire.

Par ailleurs, notre étude a fait l'objet de l'utilisation d'outils de synthèse de haut niveau. Celle-ci est en pleine expansion particulièrement avec l'adoption du standard OpenCL pour concevoir des systèmes à base d'architectures hétérogènes intégrant un CPU et une fabrique FPGA sur la même puce. Nous avons utilisé AOCL (*Altera Offline CompiLer*) pour instancier une architecture du modèle CPU/FPGA. Le paradigme de parallélisme dans AOCL est basé principalement sur l'utilisation de pipelines. Nous avons démontré à travers notre expérience, dans le contexte du GraphSLAM, que cette stratégie de parallélisme appliquée par AOCL permet d'accélérer considérablement les traitements. A fréquences égales, AOCL a permis d'accélérer la brique implantée sur FPGA d'un facteur 29 et 17 par rapport, respectivement, à la version mono-coeur et multi-coeur de l'exécution HPS. Cependant, ce gain n'est pas sans prix. AOCL produit des architectures très consommatrices en ressources logiques. Cela ne nous a malheureusement pas permis d'étendre l'étude à l'ensemble des blocs fonctionnels pour insuffisance de ressources. Nous pensons, cependant, que le transfert de données restera toujours le facteur limitant dans ce modèle d'implantation.

En résumé, nous pouvons dire que le modèle CPU/FPGA défini n'améliore pas le temps de calcul par rapport au modèle homogène en dépit de l'accélération de la construction du système. Le transfert de données reste contraignant. Cela met en avant l'avantage des architectures hétérogènes à base de GPU quant à l'accélération du GraphSLAM. En effet, le temps de transfert de données entre le CPU et le GPU peut être considérablement réduit grâce à l'utilisation d'une mémoire unifiée.

Perspectives

Plusieurs pistes sont envisageables afin d'étendre le travail présenté dans ce manuscrit et réduire encore la complexité du GraphSLAM :

Le modèle CPU/GPU défini implique la résolution du système linéaire sur CPU. Celle-ci constitue avec la construction du système la charge la plus importante pour le GraphSLAM. Le gain apporté par la parallélisation est très variable et dépend de la structure du système linéaire. En fonction de la taille de l'environnement exploré, le GraphSLAM peut manipuler relativement de larges systèmes linéaires (nombre d'arêtes : 20k, 30k, 60k,...). Les solveurs épars (Cholesky) sur GPU n'ont de gain clair que si le graphe est très large [Rodriguez-Losada et al., 2013, Wu et al., 2011]. Nous envisageons d'étudier la possibilité de déporter une partie de la résolution du système linéaire sur GPU afin d'améliorer encore les performances [Chen and Chen, 2015].

Dans le modèle CPU/GPU, il nous a été difficile de faire coopérer les deux processeurs (CPU et GPU) sur un même bloc fonctionnel. L'utilisation de la fonctionnalité zéro-copie est pénalisée par le fait que les données ne sont pas mises dans le cache ; ce qui dégrade les performances. Il serait intéressant d'investiguer une solution basée sur le transfert de données. Chaque processeur serait en charge de traiter une partie des données.

Notre stratégie consiste à utiliser des plateformes FPGA non complexes ou bas coût. Cependant, les fortes contraintes du GraphSLAM et des outils de synthèse de haut niveau (AOCL et LegUp) n'ont malheureusement pas permis d'aller au bout d'une implantation hétérogène (sur le DE1-SoC). Dans celle-ci, le bloc de construction du système linéaire serait entièrement implanté sur FPGA. Nous comptons, donc, étendre l'étude à d'autres plateformes plus performantes. Cela permettra également d'investiguer d'autres solutions basées sur l'utilisation de processeurs softcores et de blocs de mémoire on-chip dans l'objectif d'éviter autant que possible le transfert de données.

Dans ce travail, nous avons également développé un système de perception basé sur une caméra monoculaire. La partie front-end du système est en charge de l'extraction des amers et la mise en correspondance. L'initialisation des amers est effectuée en utilisant une technique de triangulation [Kitt et al., 2010]. Cette partie a été intégrée pour valider fonctionnellement l'instance élaborée du GraphSLAM. Nous souhaitons évaluer le système (Front-end et Back-end) sur un véhicule.

Pour aller plus dans la réduction de la complexité de calcul, il serait intéressant de combiner les différentes approches. La sous-cartographie et la technique de fenêtre glissante sont d'une immense importance pour faire du SLAM sur de longues périodes de temps (*longlife SLAM*) [Sibley et al., 2010, Strasdat et al., 2011, Zhao et al., 2014, Cheng et al., 2015]. L'élagage permet également de limiter la croissance rapide du graphe [Kretzschmar et al., 2010, Kretzschmar and Stachniss, 2012, Wang et al., 2013, Choudhary et al., 2015]. L'éparsification peut réduire considérablement le temps de résolution du système linéaire.

Annexes

Annexe A

Formulation probabiliste du GraphSLAM

Le GraphSLAM est basé sur une hypothèse Gaussienne. Une contrainte introduite par la mesure z_t^j entre une pose x_t et un amer m_j est de la forme :

$$(z_t^j - h(x_t, m_j))^T Q_t^{-1} (z_t^j - h(x_t, m_j)) \quad (\text{A.1})$$

où h est le modèle d'observation. De même, une contrainte de mouvement entre x_{t-1} et x_t est de la forme :

$$(x_t - g(u_t, x_{t-1}))^T R_t^{-1} (x_t - g(u_t, x_{t-1})) \quad (\text{A.2})$$

où g est le modèle de mouvement. Q_t et R_t sont les matrices de variances-covariances du bruit sur, respectivement, la mesure extéroceptive et proprioceptive.

Le GraphSLAM étant une méthode de lissage, l'objectif est donc d'estimer la densité de probabilité à posteriori de la trajectoire et de la carte, soit :

$$p(x_{0:t}, m | z_{1:t}, u_{1:t}) = \eta p(z_t | x_{0:t}, m, z_{1:t-1}, u_{1:t}) p(y_{0:t}, m | z_{1:t-1}, u_{1:t}) \quad (\text{A.3})$$

où η est la constante de normalisation. Les observations des amers sont indépendantes entre elles et ne dépendent que de la position courante du robot. Par conséquent, la première probabilité du membre droit de l'équation (Eq. A.3) peut être réduite comme suit :

$$p(z_t | x_{0:t}, m, z_{1:t-1}, u_{1:t}) = p(z_t | x_t, m) \quad (\text{A.4})$$

De la même façon, la seconde probabilité peut être factorisée en partitionnant $x_{0:t}$ en x_t et $x_{0:t-1}$; on obtient :

$$p(x_{0:t}, m | z_{1:t-1}, u_{1:t}) = p(x_t | x_{1:t-1}, m, z_{1:t-1}, u_{1:t}) p(x_{0:t-1}, m | z_{1:t-1}, u_{1:t}) \quad (\text{A.5})$$

La position courante du robot ne dépend que de la position précédente (chaîne de Markov d'ordre 1). L'équation (Eq. A.5) devient :

$$p(x_{0:t}, m | z_{1:t-1}, u_{1:t}) = p(x_t | x_{t-1}, u_t) p(x_{0:t-1}, m | z_{1:t-1}, u_{1:t-1})$$

En remplaçant ces expressions dans l'équation (Eq. A.3), on aura la définition récursive de la densité à posteriori du GraphSLAM :

$$p(x_{0:t}, m | z_{1:t}, u_{1:t}) = \eta p(z_t | x_t) p(x_t | x_{t-1}, u_t) p(x_{0:t-1}, m | z_{1:t-1}, u_{1:t-1}) \quad (\text{A.6})$$

On peut alors montrer par récurrence que :

$$p(x_{0:t}, m | z_{1:t}, u_{1:t}) = \eta p(x_0) \prod_{i=1}^t p(x_i | x_{i-1}, u_i) p(z_i | x_i) \quad (\text{A.7})$$

$$= \eta p(x_0) \prod_{i=1}^t [p(x_i | x_{i-1}, u_i) \prod_{j \in z_i} p(z_i^j | x_i, m)] \quad (\text{A.8})$$

Sachant que z_i^j est la j-ième mesure dans le vecteur des mesures z_i .

On applique désormais le logarithme sur l'équation (Eq. A.8), on obtient :

$$\log p(x_{0:t}, m | z_{1:t}, u_{1:t}) = \text{const.} + \log p(x_0) + \sum_t [\log p(x_t | x_{t-1}, u_t) + \sum_j \log p(z_i^j | x_i, m)] \quad (\text{A.9})$$

Les bruits appliqués sur le modèle d'observation et de mouvement sont supposés additifs, centrés et gaussiens. Soient Q_t et R_t les matrices de variances-covariances caractérisant le bruit sur, respectivement, la mesure d'observation (h) et de déplacement (g). Par conséquent :

$$p(x_t | x_{t-1}, u_t) = \eta \exp \left\{ -\frac{1}{2} (x_t - g(u_t, x_{t-1}))^T R_t^{-1} (x_t - g(u_t, x_{t-1})) \right\} \quad (\text{A.10})$$

$$p(z_i^j | x_i, m_i) = \eta \exp \left\{ -\frac{1}{2} (z_i^j - h(x_i, m_i))^T Q_t^{-1} (z_i^j - h(x_i, m_i)) \right\} \quad (\text{A.11})$$

La densité de la position initiale $p(x_0)$ peut être exprimée à l'aide d'une distribution Gaussienne, centrée, de matrice d'information Ω_0 :

$$p(x_0) = \eta \exp \left\{ -\frac{1}{2} x_0^T \Omega_0 x_0 \right\} \quad (\text{A.12})$$

Dans notre cas, on considère que la position initiale est $(0, 0, 0)$. Donc, Ω_0 est une matrice diagonale de très larges valeurs exprimant la certitude sur la position initiale. On obtient :

$$-\log p(x_{0:t}, m | z_{1:t}, u_{1:t}) = \text{const.} + \frac{1}{2} [x_0^T \Omega_0 x_0 \quad (\text{A.13})$$

$$+ \sum_{i=1}^t (x_i - g(u_i, x_{i-1}))^T R_i^{-1} (x_i - g(u_i, x_{i-1})) \quad (\text{A.14})$$

$$+ \sum_{i=1}^t \sum_j (z_i^j - h(x_i, m_i))^T Q_t^{-1} (z_i^j - h(x_i, m_i))] \quad (\text{A.15})$$

L'équation (Eq. A.15) caractérise la densité de probabilité à posteriori dans le GraphSLAM.

L'objectif originel de maximiser la densité de probabilité à posteriori (Eq. A.3) peut être ramené à un problème de minimisation. En effet, le logarithme de densité à posteriori permet de définir une fonction de coût F . En omettant les valeurs constantes, F est donnée par :

$$F = \sum_{i=1}^t (x_i - g(u_i, x_{i-1}))^T R_i^{-1} (x_i - g(u_i, x_{i-1})) + \sum_{i=1}^t \sum_j (z_i^j - h(x_i, m_i))^T Q_t^{-1} (z_i^j - h(x_i, m_i)) \quad (\text{A.16})$$

Annexe B

Présentation de la méthode de Cholesky

La méthode de Cholesky¹ consiste en la décomposition d'une matrice carrée symétrique définie positive A en un produit d'une matrice triangulaire inférieure L et sa transposée L^T . Soit A une matrice symétrique définie positive de taille $n \times n$. Alors, A peut être factorisée sous cette forme :

$$A = L L^T \quad (\text{B.1})$$

Le système linéaire $A x = b$ peut donc s'écrire :

$$L L^T x = b \quad (\text{B.2})$$

Ce système peut donc être traité par la résolution successive de deux systèmes triangulaires simples : $L y = b$ et $L^T x = y$. Il existe une variante de cette méthode qui tire profit du caractère épars du système et minimise le nombre de calculs. La résolution du système est effectuée en 4 étapes :

- Renumérotation
- Factorisation symbolique
- Factorisation numérique
- Résolution

B.1 Renumérotation

L'élimination d'une variable au niveau d'une ligne, lors de la factorisation, peut provoquer un remplissage qui caractérise le fait qu'un élément nul devient non nul dans L . Pour cette raison, la matrice L peut ne pas garder la même structure creuse que A ; ce qui réduit les performances en termes de calcul. Cette étape dans la factorisation permet de minimiser le remplissage dans L . Formellement, on cherche une matrice de permutation P et on applique la factorisation sur la nouvelle matrice PAP^T . Cependant, trouver une bonne permutation est un problème complexe qui nécessite le recours à des

1. Les algorithmes et les descriptions donnés dans cette annexe sont une adaptation de [Erhel et al., 1993].

heuristiques. Les algorithmes de renumérotation les plus utilisés reposent sur des principes de la théorie des graphes.

Soit A une matrice symétrique d'ordre n . Le graphe $G = (X, E)$ associé A est un graphe non orienté tel que :

$X = \{1, 2, \dots, n\}$, où chaque sommet représente une ligne.

$$E = \{(i, j) \in X^2 \mid i > j \text{ et } A_{ij} \neq 0\}$$

Pour minimiser le remplissage, il existe plusieurs algorithmes. Les plus populaires sont l'algorithme de degré minimum et l'algorithme de Dissection emboîtée.

Algorithme du degré minimum : Cet algorithme est basé sur le fait que, si $\{x_1, \dots, x_{i-1}\}$ sont numérotés, il faut supprimer le nœud du degré minimum dans le graphe G_{i-1} . Cela permet de réduire le nombre d'éléments non nuls dans la colonne i . Dans le cas où plusieurs sommets ont le même degré, on choisit arbitrairement un nœud.

1. $(G_0, E_0) = G(X, E)$

Pour $i = 1$ à $n - 1$ faire

2. Trouver $x \in X_{i-1}$ de degré minimum

3. Affecter le numéro i à x

4. Construire le nouveau graphe G_i à partir de G_{i-1} en supprimant i

La méthode de Dissection emboîtée : Cette méthode suppose que la matrice A est irréductible. Cela revient à dire que le graphe est connexe. Son principe est de diviser pour régner. Partant du graphe initial G , on cherche un ensemble de sommets S , appelé séparateur. La suppression de celui-ci divise le graphe G en au moins deux sous-graphes disjoints G_1 et G_2 . Si les sommets des deux sous-graphes sont numérotés avant ceux du séparateur, la factorisation d'un sous-graphe est indépendante de l'autre. Cette décomposition peut être appliquée récursivement sur G_1 et G_2 . Les éléments nuls sont, de la même manière, préservés dans les sous-matrices obtenues. Grâce à cette indépendance dans le traitement, les deux sous-graphes peuvent être factorisés en parallèle.

B.2 Factorisation symbolique

Le processus complet conduit à la construction d'un arbre d'élimination. Avant d'effectuer les calculs, une phase préliminaire, basée sur cet arbre d'élimination, définit la structure creuse de L .

Définitions : On appelle *structure* d'une ligne i (Res. colonne j) de A l'ensemble des indices des éléments non nuls de la ligne i (Res. colonne j) dans le triangle inférieur de A :

$$\text{Struct}(A_{i*}) = \{k < i / a_{ik} \neq 0\}$$

$$\text{Struct}(A_{*j}) = \{k > j / a_{kj} \neq 0\}$$

On appelle *premier* d'une colonne j de structure non vide, l'indice du premier élément non nul dans la colonne j , au dessous de la diagonale.

$$P(j) = \begin{cases} \min\{i \in \text{Struct}(L_{*j})\} \text{ si } \text{Struct}(L_{*j}) \neq 0 \\ j \quad \text{sinon.} \end{cases}$$

L'arbre d'élimination $T(A)$ de A est défini comme suit :

- L'ensemble de ses nœuds est l'ensemble des indices des colonnes de L
- i est le père de j (avec $i > j \leftrightarrow i = P(j)$)

L'algorithme 8 permet de construire un tel arbre. L'arbre d'élimination permet de définir la structure creuse du facteur de Cholesky L et de préparer ainsi la structure de données nécessaire pour le stockage. L'algorithme 9 explicite la manière dont la structure de L est calculée.

Algorithme 8 : Construction de l'arbre d'élimination

```

1 for  $i \leftarrow 1$  to  $n$  do
2    $\text{parent}[i] \leftarrow 0$  ;
3   for  $x_k \in \text{Adj}(x_i) \& k < i$  do
4      $r \leftarrow k$ ;
5     while  $\text{parent}[r] \neq 0 \& \text{parent}[r] \neq i$  do
6        $r \leftarrow \text{parent}[r]$ ;
7       if ( $\text{parent}[r] = 0$ ) then
8          $\text{parent}[r] \leftarrow i$ ;
9       end
10      end
11    end
12 end

```

B.3 Factorisation numérique

Durant cette phase, on calcule les éléments du facteur de Cholesky L . La factorisation s'appuie sur la structure de L . Les éléments nuls ne sont donc inclus dans les calculs. Il existe deux versions de l'algorithme de factorisation numérique : Fan-In 10 et Fan-out 11. Dans le Fan-In, la boucle externe modifie, à chaque itération, la même colonne. Par contre, dans le Fan-out, plusieurs colonnes sont modifiées.

Algorithme 9 : Calcul de la structure de L

```

1 for  $j \leftarrow 1$  to  $n$  do
2   |  $R_j \leftarrow \emptyset$  ;
3 end
4 for  $i \leftarrow 1$  to  $n$  do
5   |  $S \leftarrow Struct(A_{*j})$ ;
6   | for  $i \in R_j$  do
7     |   |  $S \leftarrow S \cup Struct(L_{*i}) - \{j\}$ ;
8   | end
9   |  $Struct(L_{*j}) \leftarrow S$ ;
10  | if  $Struct(L_{*j}) \neq \emptyset$  then
11    |   |  $p(j) \leftarrow \min(i \in Struct(L_{*j}))$ ;
12    |   |  $R_{p(j)} \leftarrow R_{p(j)} \cup \{j\}$ ;
13  | end
14 end

```

Algorithme 10 : Algorithme de factorisation Fan-In

```

1 for  $j \leftarrow 1$  to  $n$  do
2   | for  $k \in Struct(L_{j*})$  do
3     |   | for  $i \leftarrow j$  to  $n$  do
4       |     |  $A[i, j] \leftarrow A[i, j] - A[i, k] * A[j, k]$ 
5     |   | end
6     |   |  $A[j, j] \leftarrow \sqrt{A[j, j]}$  ;
7     |   | for  $i \leftarrow j + 1$  to  $n$  do
8       |     |  $A[i, j] = A[i, j] / A[j, j]$  ;
9     |   | end
10    | end
11 end

```

Algorithme 11 : algorithme de factorisation Fan-Out

```

1 for  $k \leftarrow 1$  to  $n$  do
2   |  $A[k, k] \leftarrow \sqrt{A[k, k]}$  ;
3   | for  $i \leftarrow k + 1$  to  $n$  do
4     |   |  $A[i, k] \leftarrow A[i, k] / A[k, k]$  ;
5   | end
6   | for  $j \in Struct(L_{*k})$  do
7     |   | for  $i \leftarrow j$  to  $n$  do
8       |     |  $A[i, j] \leftarrow A[i, j] - A[i, k] * A[j, k]$  ;
9     |   | end
10    | end
11 end

```

B.4 Résolution

Dans cette étape, on résout, d'abord, le système $L y = b$ (descente). L étant triangulaire inférieure, la résolution revient à calculer les éléments de la solution y un à un en substituant à chaque fois les éléments calculés par leurs valeurs. Une fois le premier système résolu, on effectue la même opération en remontée pour retrouver la solution x du système initial.

Annexe C

Validation fonctionnelle

Avant d'apporter tout type d'optimisations, nous avons validé l'aspect fonctionnel de l'algorithme GraphSLAM en utilisant des jeux de données synthétiques et réelles. La figure C.1 montre les résultats de localisation et cartographie. La trajectoire obtenue par intégration odométrique conduit à une grande dérive par rapport à la trajectoire référence. Le GraphSLAM permet de corriger la localisation et la carte. En effet, la trajectoire et la carte recouvrent pratiquement celles de la réalité du terrain en dépit

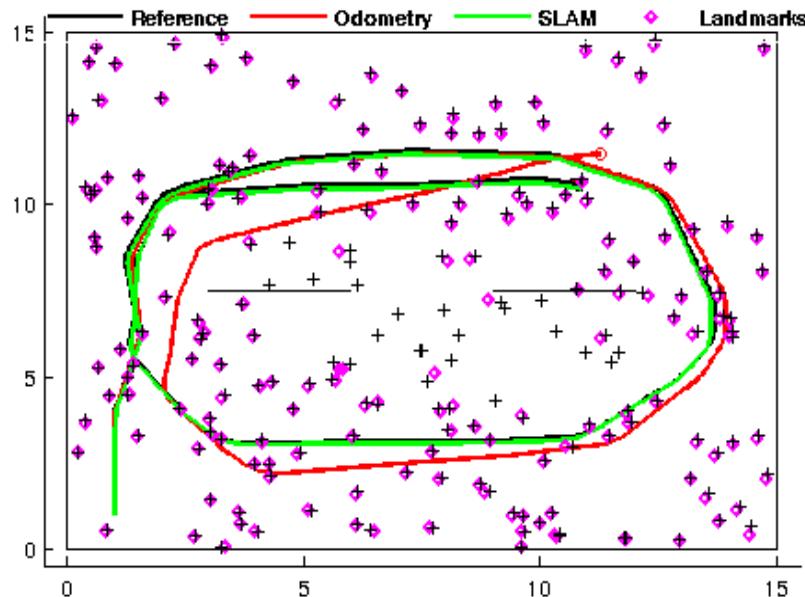


FIGURE C.1 – Validation sur un jeu de données simulé.

Comme jeu de données réelles, nous avons utilisé celui de IPDS PAVIN-Faust [Korrapati et al. \[3 06\]](#). Le jeu de données concerne un véhicule qui circule sur un circuit extérieur. Le véhicule est équipé de plusieurs capteurs. Nous avons utilisé les données des odomètres et la caméra frontale. La figure C.2 présente les résultats obtenus en utilisant notre implantation du GraphSLAM et l'intégration odométrique. Les trajectoires reconstruites sont comparées à celle fournie par un GPS-RTK (*Real Time Kinematic*). A la fin de la séquence utilisée, le véhicule effectue deux boucles. La trajectoire

estimée par le GraphSLAM est relativement proche de celle fournie par le GPS. L'odométrie donne une large dérive. En effet, on peut facilement distinguer les boucles vers la fin de la trajectoire.

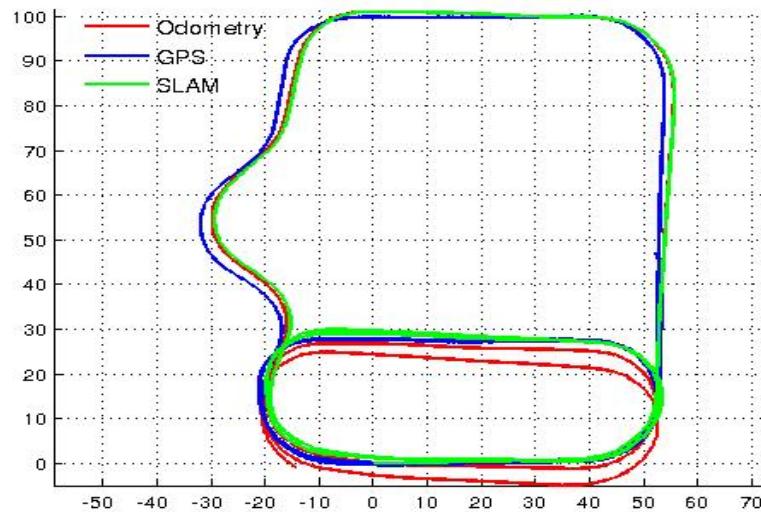


FIGURE C.2 – Validation sur le jeu de données Pavin-Faust [Dine et al., 2015a](#).

Bibliographie

- Agarwal, P., Grisetti, G., Tipaldi, G. D., Spinello, L., Burgard, W., and Stachniss, C. (2014). Experimental analysis of dynamic covariance scaling for robust map optimization under bad initial estimates. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3626–3631. IEEE. 35
- Agarwal, P. and Olson, E. (2012). Variable reordering strategies for slam. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3844–3850. IEEE. 38
- Agarwal, P., Tipaldi, G. D., Spinello, L., Stachniss, C., and Burgard, W. (2013). Robust map optimization using dynamic covariance scaling. In *2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 62–69. IEEE. 35
- Agrawal, M., Konolige, K., and Iocchi, L. (2005). Real-time detection of independent motion using stereo. In *WACV/MOTIONS'05 Volume 1. Seventh IEEE Workshops on Application of Computer Vision*, volume 2, pages 207–214. IEEE. 12
- Alahi, A., Ortiz, R., and Vandergheynst, P. (2012). Freak : Fast retina keypoint. In *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on*, pages 510–517. Ieee. 11
- Altera (2016). Altera sdk for opencl programming guide. 119
- Backes, L., Rico, A., and Franke, B. (2015). Experiences in speeding up computer vision applications on mobile computing platforms. In *Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation, SAMOS XV, Samos, Greece*. 48
- Bailey, T. and Durrant-Whyte, H. (2006). Simultaneous localization and mapping (slam) : part ii. *IEEE Robotics Automation Magazine*, 13(3) :108–117. 13
- Bailey, T., Nieto, J., and Nebot, E. (2006). Consistency of the fastslam algorithm. In *IEEE International Conference on Robotics and Automation*, pages 424–429. IEEE. 15, 16
- Bay, H., Ess, A., Tuytelaars, T., and Van Gool, L. (2008). Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3) :346–359. 11
- Bell, C. G. and Newell, A. (1971). *Computer structures : Readings and examples*, volume 2. McGraw-Hill New York. 52

- Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation. 70, 97
- Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, pages 18 :1–18 :11, New York, NY, USA. ACM. 70, 97
- Biber, P., Andreasson, H., Duckett, T., and Schilling, A. (2004). 3d modeling of indoor environments by a mobile robot with a laser scanner and panoramic camera. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 3430–3435. IEEE. 11
- Bollaert, T. (2008). Catapult synthesis : a practical introduction to interactive c synthesis. In *High-Level Synthesis*, pages 29–52. Springer. 55
- Bonato, V., de Holanda, J. A., and Marques, E. (2006). An embedded multi-camera system for simultaneous localization and mapping. In *Reconfigurable Computing : Architectures and Applications*, pages 109–114. Springer. 51
- Bosse, M., Newman, P., Leonard, J., Soika, M., Feiten, W., and Teller, S. (2003). An atlas framework for scalable mapping. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 2, pages 1899–1906. IEEE. 15
- Boulos, V. B. (2012). *Programming model for the implementation of 2D-3D image processing applications on a hybrid CPU-GPU cluster*. PhD thesis, Université de Grenoble. 43
- Brillu, R., Pillement, S., Lemonnier, F., Millet, P., Bernot, M., and Falzon, F. (2013). Algorithm-architecture adequacy, an application to the phase diversity algorithm. 43
- Cadence (2011). NVIDIA Tegra K1. http://www.nvidia.com/content/pdf/tegra_white_papers/tegra-k1-whitepaper.pdf/. 55
- Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). Brief : Binary robust independent elementary features. In *Proceedings of the 11th European Conference on Computer Vision : Part IV, ECCV'10*, pages 778–792, Berlin, Heidelberg. Springer-Verlag. 11
- Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. (2011). Legup : high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM. viii, 55, 56, 58
- Carlevaris-Bianco, N., Kaess, M., and Eustice, R. M. (2014). Generic node removal for factor-graph slam. *IEEE Transactions on Robotics*, 30(6) :1371–1385. 36
- Carlone, L. (2013). A convergence analysis for pose graph optimization via gauss-newton methods. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 965–972. IEEE.

- Carlone, L., Aragues, R., Castellanos, J. A., and Bona, B. (2012). A linear approximation for graph-based simultaneous localization and mapping. *Robotics : Science and Systems VII*, pages 41–48. 35
- Carlone, L., Aragues, R., Castellanos, J. A., and Bona, B. (2014). A fast and accurate approximation for planar pose graph optimization. *The International Journal of Robotics Research*, page 0278364914523689. 35
- Carlone, L. and Censi, A. (2014). From angular manifolds to the integer lattice : Guaranteed orientation estimation with application to pose graph optimization. *IEEE Transactions on Robotics*, 30(2) :475–492. 35
- Carlone, L., Tron, R., Daniilidis, K., and Dellaert, F. (2015). Initialization techniques for 3d slam : a survey on rotation estimation and its use in pose graph optimization. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4597–4604. IEEE. 35
- Carrasco, P. L. N., Bonin-Font, F., and Codina, G. O. (2016). Stereo graph-slam for autonomous underwater vehicles. In *Intelligent Autonomous Systems 13*, pages 351–360. Springer. 12
- Casseau, E. and Le Gal, B. (2009). High-level synthesis for the design of fpga-based signal processing systems. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*, pages 25–32. IEEE. 53
- Castellanos, J. A., Montiel, J., Neira, J., and Tardós, J. D. (1999). The spmap : A probabilistic framework for simultaneous localization and map building. *Robotics and Automation, IEEE Transactions on*, 15(5) :948–952. 13
- Celik, K., Chung, S.-J., and Somani, A. (2008). Mono-vision corner slam for indoor navigation. In *Electro/Information Technology, 2008. EIT 2008. IEEE International Conference on*, pages 343–348. IEEE. 12
- Chen, J. and Chen, Z. (2015). Cholesky factorization on heterogeneous cpu and gpu systems. In *2015 Ninth International Conference on Frontier of Computer Science and Technology*, pages 19–26. 98, 135
- Cheng, J., Kim, J., Shao, J., and Zhang, W. (2015). Robust linear pose graph-based slam. *Robotics and Autonomous Systems*, 72 :71–82. 37, 136
- Choi, S., Joung, J. H., Yu, W., and Cho, J.-I. (2011). What does ground tell us ? monocular visual odometry under planar motion constraint. In *Control, Automation and Systems (ICCAS), 2011 11th International Conference on*, pages 1480–1485. IEEE. 12
- Choudhary, S., Gupta, S., and Narayanan, P. (2010). Practical time bundle adjustment for 3d reconstruction on the gpu. In *European Conference on Computer Vision*, pages 423–435. Springer. 48, 93, 134

- Choudhary, S., Indelman, V., Christensen, H. I., and Dellaert, F. (2015). Information-based reduced landmark slam. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4620–4627. IEEE. 36, 136
- Civera, J., Davison, A. J., and Montiel, J. M. (2006). Unified inverse depth parametrization for monocular slam. In *In Proceedings of Robotics : Science and Systems*. Citeseer. 12
- Clemente, L. A., Davison, A. J., Reid, I. D., Neira, J., and Tardós, J. D. (2007). Mapping large loops with a single hand-held camera. In *Robotics : Science and Systems*, volume 2, page 2. 37
- Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. (2011a). High-level synthesis for fpgas : From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4) :473–491. 53
- Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. (2011b). High-level synthesis for fpgas : From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4) :473–491. 54
- Corp., A. (2009). Nios ii c2h compiler user guide. *Altera Corp.* 55
- Coussy, P., Chavet, C., Bomel, P., Heller, D., Senn, E., and Martin, E. (2008). Gaut : A high-level synthesis tool for dsp applications. In *High-Level Synthesis*, pages 147–169. Springer. 54
- Coussy, P., Gajski, D. D., Meredith, M., and Takach, A. (2009). An introduction to high-level synthesis. *IEEE Design and Test of Computers*, (4) :8–17. 53, 54
- Coussy, P. and Takach, A. (2008). Special issue on high-level synthesis. *IEEE Design and Test of Computers*, (5) :393. 53
- Cruz, S., Munoz, D. M., Conde, M., Llanos, C. H., and Borges, G. A. (2013). Fpga implementation of a sequential extended kalman filter algorithm applied to mobile robotics localization problem. In *Circuits and Systems (LASCAS), 2013 IEEE Fourth Latin American Symposium on*, pages 1–4. IEEE. 52
- Czajkowski, T. S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., and Singh, D. P. (2012). From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534. IEEE. 58
- Davis, T. A. (2006). *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 98
- Davison, A. J. (2003). Real-time simultaneous localisation and mapping with a single camera. In *IEEE International Conference on Computer Vision - Volume 2*, ICCV '03, pages 1403–, Washington, DC, USA. IEEE Computer Society. 12, 13
- Davison, A. J., Reid, I. D., Molton, N. D., and Stasse, O. (2007). Monoslam : Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6) :1052–1067. 12

- de la Escalera, A., Izquierdo, E., Martín, D., Musleh, B., García, F., and Armingol, J. M. (2016). Stereo visual odometry in urban environments based on detecting ground features. *Robotics and Autonomous Systems*, 80 :1–10. 12
- Dellaert, F., Carlson, J., Ila, V., Ni, K., and Thorpe, C. (2010). Subgraph-preconditioned conjugate gradients for large scale slam. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2566–2571. 33
- Dellaert, F. and Kaess, M. (2006). Square root sam : Simultaneous localization and mapping via square root information smoothing. *The International Journal of Robotics Research*, 25(12) :1181–1203. 31, 33, 38
- DeSouza, G. N. and Kak, A. C. (2002). Vision for mobile robot navigation : A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(2) :237–267. 11, 12
- Di Marco, M., Garulli, A., Lacroix, S., and Vicino, A. (2001). Set membership localization and mapping for autonomous navigation. *International Journal of robust and nonlinear control*, 11(7) :709–734. 19
- Dine, A., Elouardi, A., Bouaziz, S., and Vincke, B. (2016). Graph-based slam : Efficient algorithmic optimizations and mapping on a gpu-based heterogeneous architecture. In *IEEE Robotics and Automation Magazine (RAM)*, pages 00–00. IEEE. 48
- Dine, A., Elouardi, A., Vincke, B., and Bouaziz, S. (2014). Efficient implementation of the graph-based slam on an omap processor. In *13th International Conference on Control Automation Robotics Vision (ICARCV)*, pages 1935–1940. 67, 98
- Dine, A., Elouardi, A., Vincke, B., and Bouaziz, S. (2015a). Graph-based slam embedded implementation on low-cost architectures : A practical approach. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4612–4619. x, 147
- Dine, A., Elouardi, A., Vincke, B., and Bouaziz, S. (2015b). Speeding up graph-based slam algorithm : a gpu-based heterogeneous architecture study. In *IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 72–73. IEEE. 48
- Dissanayake, G., Williams, S. B., Durrant-Whyte, H., and Bailey, T. (2002). Map management for efficient simultaneous localization and mapping (slam). *Autonomous Robots*, 12(3) :267–286. 36
- Dissanayake, M. W. M. G., Newman, P., Clark, S., Durrant-whyte, H. F., and Csorba, M. (2001). A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotics and Automation*, 17 :229–241. 3, 9, 10, 13
- Dong-Si, T.-C. and Mourikis, A. I. (2012). Estimator initialization in vision-aided inertial navigation with unknown camera-imu calibration. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1064–1071. IEEE. 35

- Duckett, T., Marsland, S., and Shapiro, J. (2000). Learning globally consistent maps by relaxation. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 4, pages 3841–3846. IEEE. 33
- Duff, I. S., Grimes, R. G., and Lewis, J. G. (1989). Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1) :1–14. 31
- Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localization and mapping : part i. *IEEE Robotics Automation Magazine*, 13(2) :99–110. viii, 9, 13
- Durrant-Whyte, H., Majumder, S., Thrun, S., De Battista, M., and Scheding, S. (2003). A bayesian algorithm for simultaneous localisation and map building. In *Robotics Research*, pages 49–60. Springer. 13
- Eade, E., Fong, P., and Munich, M. (2010). Monocular graph slam with complexity reduction. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3017–3024. 36
- Edwards, S. A. (2006). The challenges of synthesizing hardware from c-like languages. *IEEE Design and Test of Computers*, 23(5) :375–386. 54
- Elliott, J. P. (2012). *Understanding behavioral synthesis : A practical guide to high-level design*. Springer Science and Business Media. 54
- Endres, F., Hess, J., Engelhard, N., Sturm, J., Cremers, D., and Burgard, W. (2012). An evaluation of the rgb-d slam system. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1691–1696. IEEE. 12
- Engel, J., Schöps, T., and Cremers, D. (2014). Lsd-slam : Large-scale direct monocular slam. In *Computer Vision–ECCV 2014*, pages 834–849. Springer. 12, 48
- Engelhard, N., Endres, F., Hess, J., Sturm, J., and Burgard, W. (2011). Real-time 3d visual slam with a hand-held rgb-d camera. In *Proc. of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Västerås, Sweden*, volume 180. 12
- Erhel, J., Hahad, M., and Priol, T. (1993). *Factorisation parallèle de Cholesky pour matrices creuses sur une mémoire virtuelle partagée*. IRISA. 34, 141
- Estrada, C., Neira, J., and Tardos, J. D. (2005). Hierarchical slam : Real-time accurate mapping of large environments. *IEEE Transactions on Robotics*, 21(4) :588–596. 37
- Feist, T. (2012). Vivado design suite. *White Paper*, 5. 55
- Fletcher, R. (1987). *Practical Methods of Optimization ; (2Nd Ed.)*. Wiley-Interscience, New York, NY, USA. 22
- Frese, U. (2005). A proof for the approximate sparsity of slam information matrices. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 329–335. IEEE. 30

- Fu, S., Liu, H.-y., Gao, L.-f., and Gai, Y.-x. (2007). Slam for mobile robots using laser range finder and monocular vision. In *Mechatronics and Machine Vision in Practice, 2007. M2VIP 2007. 14th International Conference on*, pages 91–96. IEEE. 11
- Gajski, D. D., Dutt, N. D., Wu, A. C., and Lin, S. Y. (2012). *High—Level Synthesis : Introduction to Chip and System Design*. Springer Science and Business Media. 54
- Gallegos, G., Meilland, M., Rives, P., and Comport, A. I. (2010). Appearance-based slam relying on a hybrid laser/omnidirectional sensor. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3005–3010. 11
- Geiger, A., Ziegler, J., and Stiller, C. (2011). Stereoscan : Dense 3d reconstruction in real-time. In *Intelligent Vehicles Symposium (IV)*. 12
- Gould, N. I. M., Scott, J. A., and Hu, Y. (2007). A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw.*, 33(2). 98
- Grisetti, G., Kümmerle, R., and Ni, K. (2012). Robust optimization of factor graphs by using condensed measurements. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 581–588. IEEE. 37
- Grisetti, G., Kümmerle, R., Stachniss, C., and Burgard, W. (2010). A tutorial on graph-based SLAM. *IEEE Transactions on Intelligent Transportation Systems Magazine*, 2 :31–43. 10, 21, 22, 23, 24, 77
- Gu, M., Guo, K., Wang, W., Wang, Y., and Yang, H. (2015). An fpga-based real-time simultaneous localization and mapping system. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 200–203. IEEE. 52
- Guo, Z., Buyukkurt, B., Cortes, J., Mitra, A., and Najjar, W. (2008). A compiler intermediate representation for reconfigurable fabrics. *International Journal of Parallel Programming*, 36(5) :493–520. 55
- Gupta, A., Karypis, G., and Kumar, V. (1997). Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel Distrib. Syst.*, 8(5) :502–520. 98
- Gupta, S., Dutt, N., Gupta, R., and Nicolau, A. (2003). Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE. 55
- Gutmann, J.-S. and Konolige, K. (1999). 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. 32
- Hahnel, D., Burgard, W., Fox, D., and Thrun, S. (2003). An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Intelligent*

- Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 206–211. IEEE. 11
- Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151. 11
- Henry, P., Krainin, M., Herbst, E., Ren, X., and Fox, D. (2010). Rgb-d mapping : Using depth cameras for dense 3d modeling of indoor environments. In *In the 12th International Symposium on Experimental Robotics (ISER)*. Citeseer. 12
- Herath, H., Kodagoda, S., and Dissanayake, G. (2007). *Stereo vision based SLAM : Issues and solutions*. ITECH. 12
- Hill, K., Craciun, S., George, A., and Lam, H. (2015). Comparative analysis of opencl vs. hdl with image-processing kernels on stratis-v fpga. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 189–193. IEEE. 59, 61
- Holz, D. and Behnke, S. (2016). Mapping with micro aerial vehicles by registration of sparse 3d laser scans. In *Intelligent Autonomous Systems 13*, pages 1583–1599. Springer. 11
- Howard, A., Matarić, M. J., and Sukhatme, G. (2001). Relaxation on a mesh : a formalism for generalized localization. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 2, pages 1055–1060. IEEE. 33
- HSL (2014). A collection of fortran codes for large-scale scientific computation. <http://www.hsl.rl.ac.uk>. 98
- Huang, G. P., Mourikis, A. I., and Roumeliotis, S. I. (2011). An observability-constrained sliding window filter for slam. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 65–72. IEEE. 37
- Huang, S. and Dissanayake, G. (2007). Convergence and consistency analysis for extended kalman filter based slam. *Robotics, IEEE Transactions on*, 23(5) :1036–1049. 15
- Hwang, S.-Y. and Song, J.-B. (2011). Monocular vision-based slam in indoor environment using corner, lamp, and door features from upward-looking camera. *Industrial Electronics, IEEE Transactions on*, 58(10) :4804–4812. 12
- Ila, V., Porta, J. M., and Andrade-Cetto, J. (2010). Information-based compact pose slam. *IEEE Transactions on Robotics*, 26(1) :78–93. 36
- Jaaskelainen, P. O., de La Lama, C. S., Huerta, P., and Takala, J. H. (2010). Opencl-based design methodology for application-specific processors. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 223–230. IEEE. 58
- Jaulin, L. (2006). Localization of an underwater robot using interval constraint propagation. In *Principles and Practice of Constraint Programming-CP 2006*, pages 244–255. Springer. 11, 19

- Jaulin, L. (2009). A nonlinear set membership approach for the localization and map building of underwater robots. *IEEE Transactions on Robotics*, 25(1) :88–98. 19
- Julier, S. J. and Uhlmann, J. K. (2001). A counter example to the theory of simultaneous localization and map building. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 4, pages 4238–4243. IEEE. 15
- Kaess, M., Johannsson, H., Roberts, R., Ila, V., Leonard, J., and Dellaert, F. (2012). iSAM2 : Incremental smoothing and mapping using the Bayes tree. *Intl. J. of Robotics Research (IJRR)*, 31 :217–236. 22, 24, 33, 37, 39, 64, 121
- Kaess, M., Ranganathan, A., and Dellaert, F. (2008). iSAM : Incremental smoothing and mapping. *IEEE Trans. on Robotics, TRO*, 24(6) :1365–1378. 24, 33, 37, 39, 64, 121
- Karakaya, S., Ocak, H., Küçükyıldız, G., and Kilinç, O. (2015). A hybrid indoor localization system based on infra-red imaging and odometry*. In *Proceedings of the International Conference on Image Processing, Computer Vision, and Pattern Recognition (IPCV)*, page 224. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 11
- Kitt, B., Geiger, A., and Lategahn, H. (2010). Visual odometry based on stereo image sequences with ransac-based outlier rejection scheme. In *Intelligent Vehicles Symposium (IV)*. 12, 136
- Kleeman, L. (2003). Advanced sonar and odometry error modeling for simultaneous localisation and map building. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 1, pages 699–704. IEEE. 11
- Klein, G. and Murray, D. (2007). Parallel tracking and mapping for small ar workspaces. In *6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234. 12, 37
- Kneip, L., Weiss, S., and Siegwart, R. (2011). Deterministic initialization of metric state estimation filters for loosely-coupled monocular vision-inertial systems. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2235–2241. IEEE. 35
- Koch, D. and Torresen, J. (2011). Fpgasort : A high performance sorting architecture exploiting runtime reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54. ACM. 50
- Konolige, K. (2004). Large-scale map-making. In *Proceedings of the National Conference on Artificial Intelligence*, pages 457–463. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999. 33
- Konolige, K. and Agrawal, M. (2007). Frame-frame matching for realtime consistent visual mapping. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 2803–2810. IEEE. 31, 38
- Konolige, K. and Agrawal, M. (2008). Frameslam : From bundle adjustment to real-time visual mapping. *Robotics, IEEE Transactions on*, 24(5) :1066–1077. 31, 36, 37, 38, 47

- Konolige, K. and Bowman, J. (2009). Towards lifelong visual maps. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1156–1163. IEEE. 36
- Konolige, K., Bowman, J., Chen, J., Mihelich, P., Calonder, M., Lepetit, V., and Fua, P. (2010a). View-based maps. *The International Journal of Robotics Research*. 37
- Konolige, K., Grisetti, G., Kummerle, R., Burgard, W., Limketkai, B., and Vincent, R. (2010b). Efficient sparse pose adjustment for 2d mapping. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 22–29. 24
- Korrapati, H., Courbon, J., Alizon, S., and Marmoiton, F. (2013-06). The Institut Pascal Data Sets : un jeu de données en extérieur, multicapteurs et datées avec réalité terrain, données d'étalonnage et outils logiciels. In *Orasis, Congrès des jeunes chercheurs en vision par ordinateur*, Cluny, France. 146
- Kretzschmar, H., Grisetti, G., and Stachniss, C. (2010). Lifelong map learning for graph-based slam in static environments. *KI-Künstliche Intelligenz*, 24(3) :199–206. 36, 136
- Kretzschmar, H. and Stachniss, C. (2012). Information-theoretic compression of pose graphs for laser-based slam. *The International Journal of Robotics Research*, 31(11) :1219–1230. 36, 136
- Kretzschmar, H., Stachniss, C., and Grisetti, G. (2011). Efficient information-theoretic graph pruning for graph-based slam with laser range finders. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 865–871. IEEE. 36
- Kummerle, R., Grisetti, G., Strasdat, H., Konolige, K., and Burgard, W. (2011). G2o : A general framework for graph optimization. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3607–3613. 22, 23, 31, 33, 38, 39, 64, 79, 120
- Lambert, A., Gruyer, D., Vincke, B., and Seignez, E. (2009). Consistent outdoor vehicle localization by bounded-error state estimation. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1211–1216. IEEE. 19
- Lattner, C. (2008). LLVM and clang : Next generation compiler technology. In *The BSD Conference*, pages 1–2. 54
- Lattner, C. and Adve, V. (2004). LLVM : A compilation framework for lifelong program analysis and transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE. 54
- Le Bars, F., Bertholom, A., Sliwka, J., and Jaulin, L. (2010). Interval slam for underwater robots ; a new experiment. In *NOLCOS 2010*, page XX. 19
- Le Gal, B. and Casseau, E. (2011). Word-length aware dsp hardware design flow based on high-level synthesis. *Journal of Signal Processing Systems*, 62(3) :341–357. 54
- Lee, K. Y., Holley, J., Bailey, M., and Bright, W. (1985). A high-level design language for programmable logic devices. *VLSI Design*, pages 50–62. 53

- Lemaire, T., Berger, C., Jung, I.-K., and Lacroix, S. (2007). Vision-based slam : Stereo and monocular approaches. *International Journal of Computer Vision*, 74(3) :343–364. 12
- Leonard, J. J. and Durrant-Whyte, H. F. (1991). Simultaneous map building and localization for an autonomous mobile robot. In *Intelligent Robots and Systems '91. 'Intelligence for Mechanical Systems, Proceedings IROS '91. IEEE/RSJ International Workshop on*, pages 1442–1447 vol.3. 9
- Leutenegger, S., Chli, M., and Siegwart, R. Y. (2011). Brisk : Binary robust invariant scalable keypoints. In *2011 International conference on computer vision*, pages 2548–2555. IEEE. 11
- Lin, M., Lebedev, I., and Wawrzynek, J. (2010). Openrcl : low-power high-performance computing with reconfigurable devices. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 458–463. IEEE. 58
- Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla : A unified graphics and computing architecture. *IEEE micro*, (2) :39–55. 44
- Lourakis, M. I. and Argyros, A. A. (2005). Is levenberg-marquardt the most efficient optimization algorithm for implementing bundle adjustment ? In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1526–1531. IEEE. 22, 47
- Lourakis, M. I. and Argyros, A. A. (2009). Sba : A software package for generic sparse bundle adjustment. *ACM Transactions on Mathematical Software (TOMS)*, 36(1) :2. 48
- Lowe, D. (1999). Object recognition from local scale-invariant features. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999.*, volume 2, pages 1150–1157 vol.2. 11
- Ltaief, H., Tomov, S., Nath, R., and Dongarra, J. (2010). Hybrid multicore cholesky factorization with multiple gpu accelerators. *IEEE Transaction on Parallel and Distributed Systems*. 48
- Lu, F. and Milios, E. (1997). Globally consistent range scan alignment for environment mapping. *AUTONOMOUS ROBOTS*, 4 :333–349. 10, 16, 32
- Madsen, K., Nielsen, H. B., and Tingleff, O. (2004). Methods for non-linear least squares problems. 22
- Martin, G. and Smith, G. (2009). High-level synthesis : Past, present, and future. *IEEE Design and Test of Computers*, (4) :18–25. 53
- Martinelli, A. (2012). Vision and imu data fusion : Closed-form solutions for attitude, speed, absolute scale, and bias determination. *IEEE Transactions on Robotics*, 28(1) :44–60. 35
- Mazuran, M., Burgard, W., and Tipaldi, G. D. (2016). Nonlinear factor recovery for long-term slam. *The International Journal of Robotics Research*, 35(1-3) :50–72. 36
- Mazuran, M., Tipaldi, G. D., Spinello, L., and Burgard, W. (2014). Nonlinear graph sparsification for slam. In *Proc. Robot. : Sci. and Syst. Conf*, pages 1–8. 36

- Melbouci, K., Collette, S. N., Gay-Bellile, V., Ait, O., Aider, M. C., and Dhome, M. (2015). Ajustement de faisceaux du slam revisité en utilisant un capteur rgb-d. In *Journées francophones des jeunes chercheurs en vision par ordinateur*. 12
- Michel, P., Chestnutt, J., Kagami, S., Nishiwaki, K., Kuffner, J., and Kanade, T. (2007). Gpu-accelerated real-time 3d tracking for humanoid locomotion and stair climbing. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 463–469. IEEE. 47
- Mingas, G., Tsardoulias, E., and Petrou, L. (2012). An fpga implementation of the smg-slam algorithm. *Microprocessors and Microsystems*, 36(3) :190–204. 52
- Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. (2002). FastSLAM : A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada. AAAI. 3, 10, 15
- Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. (2003). Fastslam 2.0 : An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *In Proc. of the Int. Conf. on Artificial Intelligence (IJCAI)*, pages 1151–1156. 16
- Moore, R. E. and Bierbaum, F. (1979). *Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (Siam Studies in Applied Mathematics, 2.)*. Soc for Industrial and Applied Math. 18
- Moreno, F.-A., Blanco, J.-L., and Gonzalez-Jimenez, J. (2015). A constant-time slam back-end in the continuum between global mapping and submapping : application to visual stereo slam. *The International Journal of Robotics Research*, page 0278364915619238. 12
- Mur-Artal, R., Montiel, J. M. M., and Tardós, J. D. (2015). ORB-SLAM : A versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5) :1147–1163. 40
- Nardi, L., Bodin, B., Zia, M. Z., Mawer, J., Nisbet, A., Kelly, P. H. J., Davison, A. J., Luján, M., O’Boyle, M. F. P., Riley, G., Topham, N., and Furber, S. (2015). Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. 48
- Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., and Fitzgibbon, A. (2011). Kinectfusion : Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE. 48
- Ni, K. and Dellaert, F. (2010). Multi-level submap based slam using nested dissection. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2558–2565. 37
- Ni, K., Steedly, D., and Dellaert, F. (2007). Out-of-core bundle adjustment for large-scale 3d reconstruction. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8. IEEE. 37

- Nikolic, J., Rehder, J., Burri, M., Gohl, P., Leutenegger, S., Furgale, P. T., and Siegwart, R. (2014). A synchronized visual-inertial sensor system with fpga pre-processing for accurate real-time slam. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 431–437. IEEE. 52
- Njiki, M. (2013). *Architecture matérielle pour la reconstruction temps réel d'images par focalisation en tout point (FTP)*. PhD thesis, Institut d'Electronique Fondamentale, Paris-Sud University. viii, 49
- Nourani-Vatani, N. and Borges, P. V. K. (2011). Correlation-based visual odometry for ground vehicles. *Journal of Field Robotics*, 28(5) :742–768. 12
- Nvidia (2013). C-to-Silicon Compiler High-Level Synthesis . http://www.cadence.com/r1/Resources/datasheets/C2Silicon_ds.pdf. viii, 45, 46
- Oh, T., Kim, H., Jung, K., and Myung, H. (2015). Graph-based slam approach for environments with laser scan ambiguity. In *12th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 139–141. IEEE. 11
- Owaida, M., Bellas, N., Daloukas, K., and Antonopoulos, C. D. (2011). Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE. 58
- Papakonstantinou, A., Gururaj, K., Stratton, J. A., Chen, D., Cong, J., and Hwu, W.-M. W. (2009). Fcuda : Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pages 35–42. IEEE. 58
- Paull, L., Huang, G., and Leonard, J. J. (2016). A unified resource-constrained framework for graph slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1346–1353. IEEE. 36
- Paz, L. M., Jensfelt, P., Tardos, J. D., and Neira, J. (2007). Ekf slam updates in $O(n)$ with divide and conquer slam. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1657–1663. IEEE. 15
- Pinies, P. and Tardos, J. D. (2008). Large-scale slam building conditionally independent local maps : Application to monocular vision. *IEEE Transactions on Robotics*, 24(5) :1094–1106. 37
- Polok, L., Ila, V., Solony, M., Smrz, P., and Zemcik, P. (2013a). Incremental block cholesky factorization for nonlinear least squares in robotics. In *Proceedings of Robotics : Science and Systems*, Berlin, Germany. 33, 38, 79, 82
- Polok, L., Solony, M., Ila, V., Smrz, P., and Zemcik, P. (2013b). Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2263–2269. 33, 66
- Powell, M. J. (1970). A hybrid method for nonlinear equations. *Numerical methods for nonlinear algebraic equations*, 7 :87–114. 22

- Ratter, A., Sammut, C., and McGill, M. (2013). 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. IEEE. 48, 93, 134
- Richter-Gottfried, F., Ditter, A., and Fey, D. (2015). Opencl 2.0 for fpgas using oclacc. *arXiv preprint arXiv:1508.07977*. 59
- Rodriguez-Losada, D., San Segundo, P., Hernando, M., de la Puente, P., and Valero-Gomez, A. (2013). Gpu-mapping : Robotic map building with graphical multiprocessors. *IEEE Robotics Automation Magazine*, 20(2) :40–51. 48, 93, 134, 135
- Rosen, D. M., Kaess, M., and Leonard, J. J. (2012). An incremental trust-region method for robust online sparse least-squares estimation. In *2012 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1262–1269. IEEE. 22, 38
- Rosen, D. M., Kaess, M., and Leonard, J. J. (2014). Rise : An incremental trust-region method for robust online sparse least-squares estimation. *IEEE Transactions on Robotics*, 30(5) :1091–1108. 38
- Rosten, E., Porter, R., and Drummond, T. (2010). Faster and better : A machine learning approach to corner detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1) :105–119. 11
- Royer, E., Lhuillier, M., Dhome, M., and Chateau, T. (2005). Localization in urban environments : monocular vision compared to a differential gps sensor. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 114–121. IEEE. 12
- Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). Orb : An efficient alternative to sift or surf. In *IEEE International Conference on Computer Vision (ICCV)*, pages 2564–2571. 11
- Salas-Moreno, R., Newcombe, R., Strasdat, H., Kelly, P., and Davison, A. (2013). Slam++ : Simultaneous localisation and mapping at the level of objects. In *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1352–1359. 11
- Scaramuzza, D. (2011). 1-point-ransac structure from motion for vehicle-mounted cameras by exploiting non-holonomic constraints. *International journal of computer vision*, 95(1) :74–85. 12
- Seignez, E., Kieffer, M., Lambert, A., Walter, E., and Maurin, T. (2005). Experimental vehicle localisation by bounded-error state estimation using interval analysis. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 1084–1089. IEEE. 19
- Seignez, E., Kieffer, M., Lambert, A., Walter, E., and Maurin, T. (2009). Real-time bounded-error state estimation for vehicle tracking. *The International Journal of Robotics Research*, 28(1) :34–48. 19

- Shen, S., Michael, N., and Kumar, V. (2011). Autonomous multi-floor indoor navigation with a computationally constrained mav. In *Robotics and automation (ICRA), 2011 IEEE international conference on*, pages 20–25. IEEE. 11
- Sibley, G., Matthies, L., and Sukhatme, G. (2010). Sliding window filter with application to planetary landing. *Journal of Field Robotics*, 27(5) :587–608. 37, 136
- Sibley, G., Sukhatme, G. S., and Matthies, L. (2007). Constant time sliding window filter slam as a basis for metric visual perception. In *Proc. IEEE International Conference on Robotics and Automation Workshop*, pages 10–14. 31, 38
- Sileshi, B., Ferrer, C., and Oliver, J. (2015). *Accelerating Techniques for Particle Filter Implementations on FPGA*, pages 19–37. Elsevier Inc. 52
- Sileshi, B., Oliver, J., Toledo, R., Gonçalves, J., and Costa, P. (2016a). On the behaviour of low cost laser scanners in hw/sw particle filter {SLAM} applications. *Robotics and Autonomous Systems*, 80 :11 – 23. 52
- Sileshi, B. G., Oliver, J., Toledo, R., Goncalves, J., and Costa, P. (2016b). Particle filter slam on fpga : A case study on robot@ factory competition. In *Robot 2015 : Second Iberian Robotics Conference*, pages 411–423. Springer. 52
- Sliwka, J., Bars, F. L., Reynet, O., and Jaulin, L. (2011). Using interval methods in the context of robust localization of underwater robots. In *Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS)*, pages 1–6. IEEE. 11, 19
- Smith, R., Self, M., and Cheeseman, P. (1990). Estimating uncertain spatial relationships in robotics. In *Autonomous robot vehicles*, pages 167–193. Springer. 10, 13
- Smith, R. C. and Cheeseman, P. (1986). On the representation and estimation of spatial uncertainty. *The international journal of Robotics Research*, 5(4) :56–68. 10, 13
- Snavely, N., Seitz, S. M., and Szeliski, R. (2008). Skeletal graphs for efficient structure from motion. In *CVPR*, volume 1, page 2. 37
- Sola, J., Monin, A., Devy, M., and Lemaire, T. (2005). Undelayed initialization in bearing only slam. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2499–2504. IEEE. 12
- Solutions, A. D. (2008). Handel-c language reference manual. 55
- Sorel, Y. (1994). Massively parallel computing systems with real time constraints : the “algorithm architecture adequation” methodology. In *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, pages 44–53. IEEE. 43
- Strasdat, H., Davison, A. J., Montiel, J., and Konolige, K. (2011). Double window optimisation for constant time visual slam. In *2011 International Conference on Computer Vision*, pages 2352–2359. IEEE. 37, 136

- synopsys (2016). SynphonyC-Compiler. <https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx>. 55
- Takezawa, S., Herath, D. C., and Dissanayake, G. (2004). Slam in indoor environments with stereo vision. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1866–1871. IEEE. 12
- Tertei, D. T., Piat, J., and Devy, M. (2016). Fpga design of ekf block accelerator for 3d visual slam. *Computers and Electrical Engineering*. 52
- Thomas, D. B., Howes, L., and Luk, W. (2009). A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM. 50
- Thrun, S. and Liu, Y. (2005). Multi-robot slam with sparse extended information filters. In *Robotics Research. The Eleventh International Symposium*, pages 254–266. Springer. 33
- Thrun, S., Liu, Y., Koller, D., Ng, A. Y., Ghahramani, Z., and Durrant-Whyte, H. (2004). Simultaneous localization and mapping with sparse extended information filters. *The International Journal of Robotics Research*, 23(7-8) :693–716. 15, 33
- Thrun, S. and Montemerlo, M. (2006). The graph slam algorithm with applications to large-scale mapping of urban structures. *I. J. Robotic Res.*, 25(5-6) :403–429. viii, 3, 10, 22, 23, 24, 30, 31, 38
- Toronto, U. (2015). Legup documentation : Realease 4.0. 55
- Triggs, B., McLauchlan, P. F., Hartley, R. I., and Fitzgibbon, A. W. (1999). Bundle adjustment—a modern synthesis. In *Vision algorithms : theory and practice*, pages 298–372. Springer. 47
- Triggs, B., McLauchlan, P. F., Hartley, R. I., and Fitzgibbon, A. W. (2000). Bundle adjustment - a modern synthesis. In *International Workshop on Vision Algorithms : Theory and Practice*, ICCV '99, pages 298–372, London, UK, UK. Springer-Verlag. 79
- Tripp, J. L., Gokhale, M. B., and Peterson, K. D. (2007). Trident : From high-level language to hardware circuitry. *Computer*, (3) :28–37. 55
- Ventura, J., Arth, C., Reitmayr, G., and Schmalstieg, D. (2014). Global localization from monocular slam on a mobile phone. *IEEE transactions on visualization and computer graphics*, 20(4) :531–539. 12
- Villarreal, J., Park, A., Najjar, W., and Halstead, R. (2010). Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134. IEEE. 55
- Vincke, B. (2012). *Architectures pour des systèmes de localisation et de cartographie simultanées*. PhD thesis, Institut d'Electronique Fondamentale, Paris-Sud University. x, 19, 103, 104

- Vincke, B. and Lambert, A. (2011). Experimental comparison of bounded-error state estimation and constraints propagation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4724–4729. 19
- Walcott-Bryant, A., Kaess, M., Johannsson, H., and Leonard, J. J. (2012). Dynamic pose graph slam : Long-term mapping in low dynamic environments. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1871–1878. IEEE. 36
- Wan, E. A. and Van Der Merwe, R. (2000). The unscented kalman filter for nonlinear estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158. Ieee. 15
- Wang, Y., Xiong, R., Li, Q., and Huang, S. (2013). Kullback-leibler divergence based graph pruning in robotic feature mapping. In *Mobile Robots (ECMR), 2013 European Conference on*, pages 32–37. IEEE. 36, 136
- Warne, D. J., Kelson, N. A., and Hayward, R. F. (2014). Comparison of high level fpga hardware design for solving tri-diagonal linear systems. *Procedia Computer Science*, 29 :95–101. 59
- Wu, C., Agarwal, S., Curless, B., and Seitz, S. (2011). Multicore bundle adjustment. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3057–3064. 48, 93, 134, 135
- Xilinx (2014). Sdaccel development environment. <http://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf>. 59
- Zhang, G., Lee, J. H., Lim, J., and Suh, I. H. (2015). Building a 3-d line-based map using stereo slam. *IEEE Transactions on Robotics*, 31(6) :1364–1377. 12
- Zhang, H. and Martin, F. (2013). Cuda accelerated robot localization and mapping. In *2013 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE. 47
- Zhang, L., Meng, X.-J., and Chen, Y.-W. (2009). Convergence and consistency analysis for fastslam. In *IEEE Intelligent Vehicles Symposium*, pages 447–452. IEEE. 16
- Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., and Cong, J. (2008). *AutoPilot : A Platform-Based ESL Synthesis System*, pages 99–112. Springer Netherlands, Dordrecht. 55, 58
- Zhao, L., Huang, S., and Dissanayake, G. (2013). Linear slam : A linear solution to the feature-based and pose graph slam based on submap joining. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 24–30. 37
- Zhao, L., Huang, S., and Dissanayake, G. (2014). Linear monoslam : A linear approach to large-scale monocular slam problems. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1517–1523. IEEE. 37, 136
- Zhao, L., Huang, S., Yan, L., and Dissanayake, G. (2011). Parallax angle parametrization for monocular slam. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3117–3124. IEEE. 37

- Zia, M. Z., Nardi, L., Jack, A., Vespa, E., Bodin, B., Kelly, P. H., and Davison, A. J. (2016). Comparative design space exploration of dense and semi-dense slam. ICRA. 48

