Application d'une méthodologie de co-design à la définition et à l'implémentation d'une chaîne SLAM opérationnelle

François Brenot

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Institut National Polytechnique de Toulouse (INP Toulouse)

Présentée et soutenue le *25/11/2016* **par :**

FRANÇOIS BRENOT

Application d'une méthodologie de co-design à la définition et à l'implémentation d'une chaîne SLAM opérationnelle

---

JURY

PHILIPPE FILLATREAU	Maître de conférences	Président du Jury
JONATHAN PIAT	Maître de conférences	Membre du Jury
MICHEL DEVY	Directeur de recherche	Membre du Jury
ALBERTO IZAGUIRRE	Professor	Membre du Jury
FRANÇOIS BERRY	Maître de conférences	Membre du Jury
DOMINIQUE GINHAC	Professeur des Universités	Membre du Jury

École doctorale et spécialité :
EDSYS : Robotique 4200046
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directeur(s) de Thèse :
Philippe FILLATREAU et *Jonathan PIAT*
Rapporteurs :
François BERRY et *Alberto IZAGUIRRE*

Résumé

Dans le domaine de la détection et du suivi d'obstacles pour les systèmes ADAS (Advanced Driver Assistance System) basés vision, il est nécessaire d'assurer la localisation à court terme du véhicule. Le SLAM (Simultaneous Localization and Mapping) basé vision propose de résoudre ce problème en combinant l'estimation de l'état du véhicule (pose dans un repère local et vitesses) et une modélisation incrémentale de l'environnement. Ce dernier est perçu par l'extraction de caractéristiques locales (points d'intérêt) dans une séquence d'images et leur suivi au cours du temps pour permettre la construction incrémentale d'une carte d'amers. Cette tâche de perception engendre une importante charge de calcul qui affecte très significativement la latence et la cadence du système.

Les méthodologies de co-design permettent de concevoir une architecture mixte de calcul pour l"exécution d'une application particulière. Dans ce type d'architecture, l'utilisation d'accélérateurs matériels permet d'améliorer significativement les performances (temps d'exécution, encombrement, consommation). Le ZynQ (Xilinx) propose une architecture de prototypage mixte comprenant un processeur dual-core associé à des ressources matérielles configurables.

L'objectif de cette thèse est donc de proposer une implémentation co-design d'un SLAM basé vision par la conception d'accélérateurs pour les opérations de vision afin de satisfaire les contraintes en performances des systèmes ADAS embarqués.

La première contribution des travaux est la conception de cette chaîne complète 3D EKF-SLAM à l'aide une approche co-design. Nous avons défini et validé, selon notre méthodologie de conception, le choix d'une architecture Hardware-in-the-loop (HIL) afin de valider les différentes itérations de conception.

La seconde contribution est l'intégration de modules matériels dédiés pour accélérer les traitements de perception visuelle de cette chaîne (détection, description et mise en correspondance de points d'intérêt).

Abstract

In the field of obstacle detection and tracking for vision-based ADAS (Advanced Driver Assistance System), it is necessary to perform short-term vehicle localisation. Vision based SLAM (Simultaneous Localization and Mapping) solves this problem by combining the vehicle state estimation (local pose and speeds) and an incremental modelling of the environment. The environment is perceived by extracting features (interest points) in a sequence of images and tracking them over time to allow an incremental landmarks map construction. The perception step leads to an important computational load which affects very significantly the system latency and throughput.

Co-design methodologies allow to design a mixed processing architecture optimized for a specific application In this type of architecture, the use of hardware accelerators allows for great performance increase (throughput, memory size, power consumption). The ZynQ (Xilinx) provides a prototyping mixed-architecture including a dual-core microprocessor combined with configurable hardware resources.

The goal of this thesis is to propose a co-design implementation of a vision-based SLAM processing chain involving hardware accelerators for image processing in order to meet the constraints of an embedded ADAS system.

The first contribution is the design of a complete 3D EKF-SLAM processing chain thanks to a co-design approach. We defined and validated, according to the followed co-design approach, the choice of a Hardware-in-the-loop (HIL) architecture to validate the different design iterations.

The second contribution is the integration of dedicated hardware modules to accelerate the visual perception computations of this processing chain (features detection, description and matching).

Remerciements

A Philippe Fillatreau et Jonathan Piat, mes directeurs de thèse qui m'ont soutenu durant toute la durée de cette thèse. Ils m'ont formé à la robotique, au traitement d'images et m'ont permis d'améliorer ma méthodologie de travail. Tant au point de vu technique que humain j'ai pu compter sur eux.

A mes parents, re-lecteurs sans faille (même pendant leurs vacances bien méritées), qui m'ont toujours soutenu malgré la distance.

A mes soeurs, cousins, cousines, tantes, oncles, grand-mères et les petit(es) derniers/ières de la famille.

A Simon et Nelly qui ont su m'apporter leur aide dans la dernière ligne droite et m'ont permis de m'évader le temps de réunions hebdomadaires. Ma prochaine aventure commence avec vous.

A Lucie, Alice et Anaïs mes colocataires.

A mes amis et amies.

Abréviations

ABS : Anti-lock Braking System
ADAS : Advanced Driver Assistance Systems
AHP : Anchored Homogeneous Point
ALU : Arithmetic Logic Unit
ASIC : Application-Specific Integrated Circuit
BDF : Boolean Data Flow
BRAM : Block Random Access Memory
BRIEF : Binary Robust Independent Elementary Features
BRISK : Binary Robust Invariant Scalable Keypoints
CNRS : Centre Nationnal de Recherche Scientifique
CPU : Central Processing Unit
CLE : Contrat de recherche Laboratoires-Entreprises
DAG : Directed Acyclic Graph
DARPA : Defense Advanced Research Projects Agency
DICTA : Development of an Integrated Camera for Transport Applications
DIDS : Décision et Interaction Dynamiques pour les Systèmes
DSP : Digital Signal Processor
DTSO : Delta Technologies Sud-Ouest
ECS : Electronic Stability Control
EKF : Extendded Kalman Filter
ESC : Electronic Stability Control
FAST : Features From Accelerated Segment Test
FIFO : First In First Out
FPGA : Field-Programmable Gate Array
GPGPU : General-Purpose computing on Graphics Processing Units
GPS : Global Positioning System
GPU : Graphics Processing Unit
GSM : Global System for Mobile communications
HIL : Hardware-In-the-Loop
HLS High Level Synthesis
HW : HardWare
IDP : Inverse-depth Landmark Parametrization
IMU : Inertial Measurement Unit
IP : Intellectual Property
LAAS : Laboratoire d'Analyse et d'Architecture des Systèmes
LGP : Laboratoire Génie de Production
LIDAR : Light Detection And Ranging
LUT : Loo-Up Table
NMS : Non Maxima Suppression
ORB : Oriented FAST and Rotated BRIEF
PE : Processing Element
PLL : Phase Locked Loop
PSDF : Parametrized Synchronous Data Flow

QVGA : Quarter Video Graphics Array
RAM : Random Access Memory
RANSAC : RANdom SAmple Consensus
RAP : Robotique, Action et Perception
REMODE : REal-time probabilistic, MOnocular, DEnse reconstruction
ROI : Region Of Interest
RT-SLAM : Real Time Simultaneous Localization and Mapping
RTL : Register-Transfer Level
SDF : Synchronous Data Flow
SIFT : Scale-Invariant Feature Transform
SIMD : Single Instruction, Multiple Data
SLAM : Simultaneous Localization and Mapping
SoC : System on a Chip
SURF : Speeded-Up Robust Features
SVO : Semi-direct Visual Odometry
SW : SoftWare
V2V : Vehicle to Vehicle
VGA : Video Graphics Array
VHDL : VHSIC Hardware Description Language
VHSIC : Very High Speed Integrated Circuit

Table des matières

Introduction Générale

Les ADAS (Advanced Driver Assistance Systems, ou Systèmes Avancés d'Assistance au Conducteur), apparus dans les années 2000, sont aujourd'hui de plus en plus présents dans les voitures de série. Ils mettent en œuvre des capteurs (caméras, lidar, autres) ou des concepts (extraction et analyse de caractéristiques à partir des signaux ou images acquis) validés notamment par la recherche dans le domaine de la robotique mobile autonome depuis les années 1980. Cependant, ils implantent encore le plus souvent des fonctions spécifiques et indépendantes (telles que la vérification des distances de sécurité, la détection de franchissement de lignes, la surveillance de l'hypovigilance du conducteur, etc). Des fonctionnalités telles que la localisation fine du véhicule et la cartographie éparse de son environnement permettraient d'ouvrir la voie vers la détection, l'identification ou le suivi d'obstacles (notamment grâce à la réduction de la complexité des traitements associés), et vers la navigation autonome de véhicules routiers à grande vitesse.

Les fonctionnalités de localisation et de modélisation ou cartographie de l'environnement sont des problématiques abordées par la communauté de la robotique mobile, d'abord séparément (à partir des années 1980) puis de manière conjointe et unifiée (à partir des années 1990 puis 2000). Les moyens de perception utilisés se sont basés sur des capteurs divers : ultrasons, télémètres laser puis vision à partir des années 2000. L'émergence des caméras bas-coût et de résolution adaptée permettent leur mise en œuvre dans des systèmes de stéréovision, monoculaires ou de vision panoramique.

La problématique du SLAM (Simultaneous Localization And Mapping/ Localisation et Cartographie simultanées) a été désormais largement explorée par la communauté scientifique, et de nombreuses solutions ont été proposées en milieu académique. Cependant, il existe très peu de chaînes opérationnelles embarquées compatibles avec les contraintes (temps d'exécution, consommation, encombrement) d'un ADAS. Ceci s'explique en partie par la complexité liée au traitement des données images, due au volume de données à traiter (les applications de vision impliquent le traitement d'un grand nombre de pixels) et à la complexité intrinsèque des algorithmes de traitement d'images mis en jeu.

La communauté de la vision ou de la perception pour la robotique a commencé à s'intéresser il y a une quinzaine d'années à la conception d'architectures matérielles spécifiques, embarquées et temps réel, pour le traitement d'images avancé. De nombreux progrès restent à faire pour le SLAM basé vision, chaîne complexe de traitement d'images et de filtrage numérique.

Lorsqu'on envisage l'intégration de ce type de chaîne de traitement, il est nécessaire d'explorer les multiples possibilités de ressources d'exécution pour tirer parti de leurs caractéristiques respectives. Cette exploration doit mener le concepteur à la formulation d'une architecture de calcul hétérogène (ou non) mêlant des unités de calcul génériques à des accélérateurs spécifiques. La distribution du calcul sur ces deux familles d'opérateurs nécessite d'aborder la conception de l'architecture dans une démarche conjointe dite de co-design.

Notre travail s'intéresse à la définition d'une architecture hétérogène pour l'intégration d'une chaîne complète de SLAM monoculaire compatible avec les contraintes d'un ADAS. Pour cela, nous avons étudié la mise en œuvre d'une méthodologie de co-design appropriée et le développement d'accélérateurs spécifiques pour les fonctions de traitement d'images avancées utilisées.

Le mémoire est organisé comme suit.

La première partie présente le contexte et la motivation des travaux, l'état de l'art des domaines scientifiques concernant notre étude et notre positionnement par rapport à celui-ci. Le chapitre 1 présente tout d'abord le contexte de nos travaux qui concernent les ADAS, la localisation et la cartographie de l'environnement pour la robotique mobile autonome et les systèmes temps réel embarqués. Le chapitre se poursuit par l'évocation du projet coopératif laboratoires-entreprises dans le contexte duquel nos travaux se sont déroulés. Nous y présentons ensuite les objectifs de nos travaux, annonçons ses principales contributions scientifiques et détaillons l'organisation du document. Le chapitre 2 présente un état de l'art des divers domaines relatifs à nos travaux : SLAM, SoC cibles, et méthodologies de co-design. En conclusion, nous synthétisons nos hypothèses de travail et notre positionnement méthodologique par rapport à cet état de l'art.

La deuxième partie présente nos principales contributions pour l'intégration d'une chaîne SLAM embarquée. Le chapitre 3 décrit les grandes lignes de l'application de la méthodologie de co-design retenue à cette intégration originale. Le chapitre 4 se focalise sur la partie logicielle et son optimisation et évoque les possibilités des accélérations matérielles pour le filtrage et la mise à jour de l'état du système (localisation du robot et carte d'amers dans l'environnement). Le chapitre 5 présente les accélérateurs matériels originaux de fonctionnalités de traitement d'images que nous avons proposés et validés. Ces contributions sont illustrées dans chacun de ces chapitres par les résultats obtenus en laboratoire et en conditions opérationnelles.

Enfin, ce mémoire se termine par une synthèse des travaux et une présentation des perspectives qu'ils offrent.

Première partie

Contexte - État de l'art

Dans ce chapitre nous allons présenter le contexte de nos travaux. Nous définirons dans un premier temps les systèmes ADAS dans la section 1.1. Dans la section 1.2, nous introduirons le concept de localisation pour la navigation autonome et définirons la notion de temps réel dans les systèmes embarqués dans la section 1.3. Ensuite nous présenterons le projet DICTA en section 1.4 puis les objectifs des travaux en section 1.5. Enfin, nous terminerons en présentant les contributions scientifiques de notre travail en section 1.6 ainsi que les publications relatives à ces travaux en section 1.7.

Chapitre 1

Contexte des travaux

1.1 Les systèmes ADAS

Les ADAS (Advanced Driver Assistance Systems) sont des systèmes développés pour améliorer la sécurité et la conduite de véhicules. Ces fonctionnalités ADAS peuvent être basées sur des systèmes de vision par caméra et/ou des technologies de capteurs infrarouges, LIDAR [1] ou encore ultrasons. Ils peuvent aussi permettre la mise en réseau des voitures et/ou la communication véhicule à véhicule (V2V [2]).

Il existe trois types d'applications :

— Les **véhicules particuliers**, véhicules de transport de personnes pouvant être utilisés à des fins privées ou professionnelles et comportant au minimum 4 roues et au maximum 9 places assises ;

— Les **transports publics** incluant tous les transports de personnes ou de marchandises, à l'exception des transports qu'organisent, pour leur propre compte, des personnes publiques ou privées ;

— Les **engins industriels** incluant par exemple les camions (tracteurs, porteurs, fourgons, bennes, malaxeurs, citernes, frigorifiques, bétaillères, pompiers, secours ou encore de voiries).

Ces applications partagent certaines contraintes mais chaque système a ses spécificités liées au contexte. Par exemple, les engins industriels comme les tracteurs autonomes évoluent dans un champ (donc en extérieur) sans interaction avec d'autres véhicules et sans règles de conduite comme le code de la route. Dans les applications de transports publics, on retrouve des applications sur des sites propres [3]. Sur les voies dédiées aux bus par exemple, la présence de voitures est exceptionnelle mais celle des piétons est plus fréquente. Dans le domaine de la voiture autonome pour les particuliers, on retrouve entre autres l'interaction avec d'autres véhicules, piétons et cyclistes. C'est de loin le domaine d'application le plus contraignant (le problème de la navigation autonome d'un véhicule en ville n'est toujours pas résolu à ce jour).

Des équipements de sécurité ont été implantés sur les véhicules du commerce dès 1958 et sont désormais standards dans les voitures : les ceintures de sécurité (obligatoires), l'Antilock Braking System (ABS), les airbags, l'Electronic Stability Control (ESC). Les ADAS ont étés implantés plus tardivement sur les véhicules du commerce. Les premiers sont apparus dans les années 2000. Ces différents systèmes ont pour but, d'une part, d'améliorer la conduite et la sécurité, mais aussi de rendre un jour les véhicules complètement autonomes.

1. LIDAR : LIght Detection And Ranging, technologie qui mesure les distances en illuminant la cible avec un laser (laser scanning ou 3D scanning).

2. V2V : Vehicle to Vehicle

3. Voies ou espaces réservés aux transports collectifs (le concept du site propre est défini dans l'arrêté du 21 septembre 1993 relatif à la terminologie des transports : Emprise affectée exclusivement à l'exploitation de lignes de transport)

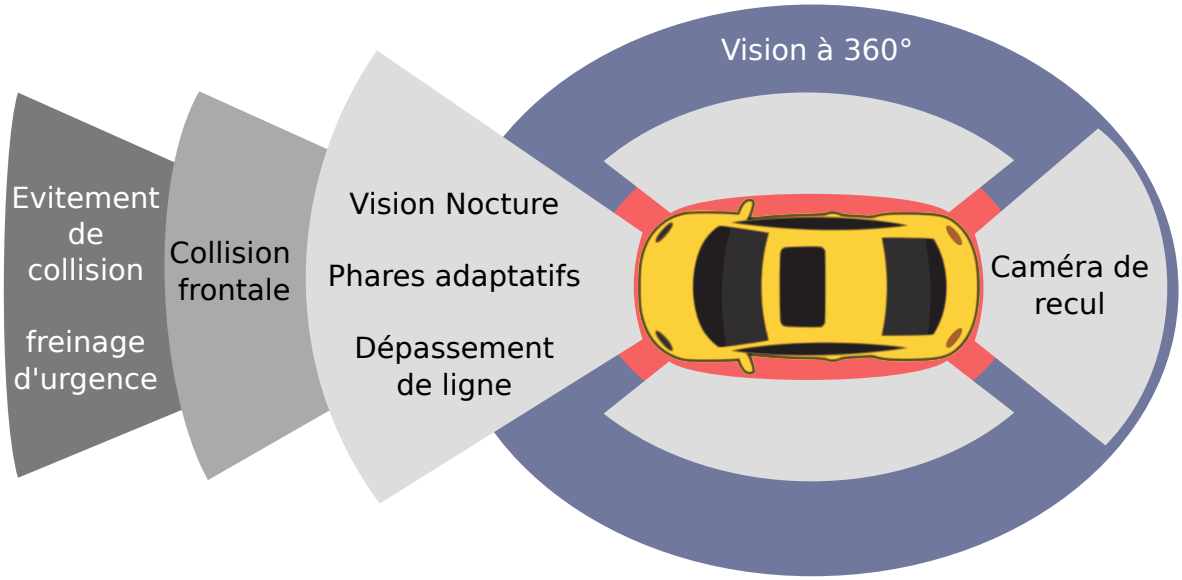

FIGURE 1.1 – Différents types de systèmes ADAS - Source : Texas Instruments®.

Un classement standard des ADAS répartit ces systèmes selon trois groupes : l'**aide à la conduite**, l'**avertissement** et l'**assistance à la conduite** (les exemples cités sont repérés dans la figure 1.1). Nous allons définir ces catégories en citant des exemples d'applications présents sur les véhicules de série.

Les équipements d'**aide à la conduite** permettent de faciliter la conduite. Parmi ces équipements on retrouve dès 2000 la vision nocturne, les caméras de recul (aide au parking) en 2002, les phares adaptatifs (diminution de la luminosité à la croisée d'un autre véhicule) en 2006 ou encore les systèmes de vision à 360 degrés en 2007.

Les systèmes d'**avertissement** préviennent d'un danger en le signalant par des sons, vibrations [4] ou encore des signaux lumineux. Ces équipements sont apparus dès 2000 avec les systèmes d'avertissement de collision frontale. Le radar de recul est apparu en 2002 et les systèmes d'avertissement de dépassement de ligne ont été implantés en 2005.

La dernière catégorie, l'**assistance à la conduite**, est plus complexe à implanter car elle requière une grande capacité de calculs nécessitant généralement la fusion de données provenant de capteurs différents. Cette famille englobe les systèmes prenant le contrôle d"une ou plusieurs fonctions du véhicule (accélération, freinage, contrôle de la trajectoire, etc). Elle gère des informations de haut niveau et se base systématiquement sur les systèmes d'avertissement (ils créent des alarmes qui sont intégrées dans le système d'assistance à la conduite). On y retrouve l'évitement de collision implanté en 2008, le stationnement automatique en 2006, le maintien du véhicule entre les lignes si le clignotant n'est pas activé en 2014 ou encore l'évitement de piéton en 2014.

Dans [Savatier, 2015] l'auteur propose un classement des ADAS d'assistance à la conduite en fonction du niveau d'automatisation du système (de 0 pour un simple avertissement à 5 pour une automatisation complète du véhicule). Le tableau 1.1 définit ce classement.

Le développement de la voiture autonome est freiné par 3 facteurs. Tout d'abord, l'in-

4. Vibrations du volant ou du siège

	Niveau d'automatisation		Précisions
L'homme supervise la machine	0	Pas d'automatisation	Le conducteur effectue toutes les tâches de conduite avec l'aide éventuelle de systèmes d'avertissement.
	1	Assistance à la conduite	Le système d'assistance agit soit sur la direction soit sur la vitesse en utilisant des informations sur la situation de conduite ; le conducteur effectue toutes les autres tâches.
	2	Automatisation partielle	Le système d'assistance agit à la fois sur la direction et sur la vitesse en utilisant des informations de la situation de conduite ; le conducteur effectue toutes les autres tâches.
La machine supervise l'homme	3	Automatisation sous conditions	Dans certaines situations de conduite bien définies, le système d'assistance agit sur l'ensemble des tâches (direction, accélération et freinage) de conduite mais le conducteur doit rester en mesure de reprendre le contrôle.
	4	Automatisation élevée	Dans certaines situations de conduite bien définies, le système d'assistance agit sur l'ensemble des tâches de conduite sans action du conducteur.
	5	Automatisation complète	Le système d'assistance agit sur l'ensemble des tâches de conduite sans action du conducteur et en toute situation.

TABLE 1.1 – Degrés d'automatisation des systèmes ADAS

dustrie doit faire face à l'**opinion générale**. En effet, comme pour la robotique, l'idée de se faire superviser par un système automatisé n'est pas bien perçue par la majeure partie des utilisateurs. Le deuxième frein à ce développement est la question de la **responsabilité** en cas d'accident. Ce tableau 1.1, présenté dans [Savatier, 2015], montre une séparation entre les systèmes compatibles avec les lois en vigueur (l'homme supervise la machine) légalement et les systèmes incompatibles avec celles-ci (la machine supervise l'homme). En effet lors d'un accident impliquant une voiture dont le niveau d'automatisation est de 3, 4 ou 5 de nouvelles questions de responsabilité sont soulevées. Qui est responsable ? Le fabriquant de la voiture, l'humain présent dans la voiture (qui aurait dû prendre ou ne pas prendre le contrôle de la voiture), le fabricant des capteurs, etc ? L'industrie doit alors trouver des solutions et les pays doivent procéder à des modifications de lois pour pouvoir voir un jour des véhicules complètement automatisés sur les routes publiques. Enfin, le développement de la voiture autonome est freiné par la **cohabitation humain-robot**, celle entre les modèles de voitures standards et les véhicules automatisés. En effet, les modèles conduits par l'humain sont susceptibles d'avoir des comportements qui ne rentrent pas dans un champ d'actions identifiées et utilisées pour développer les systèmes autonomes. Les véhicules entièrement automatisés doivent donc être capables de gérer des situations imprévues tout en satisfaisant des contraintes de fiabilité et sécurité strictes.

Les travaux de cette thèse s'inscrivent dans la catégorie des systèmes ADAS d'**aide à la conduite** et plus précisément la localisation fine d'un véhicule. En effet, le système ne prend pas le contrôle du véhicule, il renvoit juste une information. Le système que nous proposons est applicable sur n'importe quel type de mobile. On peut alors étendre notre problématique à la localisation d'un **robot ou d'un véhicule**. La localisation est une connaissance pré-requise à tout système autonome qui doit effectuer un déplacement. Enfin, pour définir une application de référence, nous nous sommes fixés le but de créer un prototype applicable à l'automobile. Ce choix nous permet, par exemple, de bien cibler les contraintes (en fréquence de traitement, consommation ou encore en encombrement) du système (présentées dans la section 1.3).

1.1.1 Mobileye

Une caméra dans laquelle est ajoutée une puce électronique permettant d'acquérir, de traiter de l'information et de communiquer avec les systèmes environnants est appelée *smart camera*. On en trouve de différents types aussi bien dans l'industrie que dans la recherche [Birem and Berry, 2014]. Les applications sont nombreuses et on peut citer par exemple [Sérot et al., 2013] et [Lapray et al., 2014].

Parmi les systèmes ADAS utilisant la vision pour détecter des événements, la société Mobileye® [5] a mis sur le marché une caméra capable d'analyser la route et d'envoyer des signaux d'avertissement (voir figure 1.2).

Pour cela, la société Mobileye® a designé un SoC [6] (EyeQ) dédié spécifiquement aux traitements d'images. Mobileye® a pu réduire les coûts de fabrication en produisant à grande échelle tout en garantissant une capacité de calcul optimisée pour l'application.

Le système Mobileye® est capable de :

— Prévenir en cas de collision frontale ;
— Alerter en cas de collision de piéton imminente ;
— Avertir lors de dépassements de ligne ;
— Indiquer les limites de vitesse ;
— Contrôler intelligemment les feux de route du véhicule.

Ce capteur est disponible au grand public et est implanté, entre autres, dans les voitures de la société Tesla®. Mobileye® a aussi des partenariats avec les plus grands constructeurs d'automobiles : Volkswagen®, BMW®, GM®, Hyudai®, Opel®, Peugeot®, Citroen®, Honda®, etc.

1.1.2 Projets de voitures autonomes (Self-Driving Car Projects)

Nous allons, dans cette section, décrire différents projets ADAS de voitures autonomes. Plusieurs de ces projets ont permis de développer les technologies liées aux véhicules totalement autonomes. Entre autres, le concours DARPA (Defense Advanced Research Projects Agency) Robotics Challenge, met en concurrence des équipes sur différents objectifs - [DARPA, 2012]. Ce challenge a été créé pour dynamiser les développements dans le secteur de la robotique, afin d'implanter des technologies innovantes d'abord dans des applications militaires, puis dans celles du grand public notamment pour faire de l'assistance à la conduite. Le concours DARPA a évolué au cours des différentes éditions :

5. http ://www.mobileye.com
6. System-On-Chip

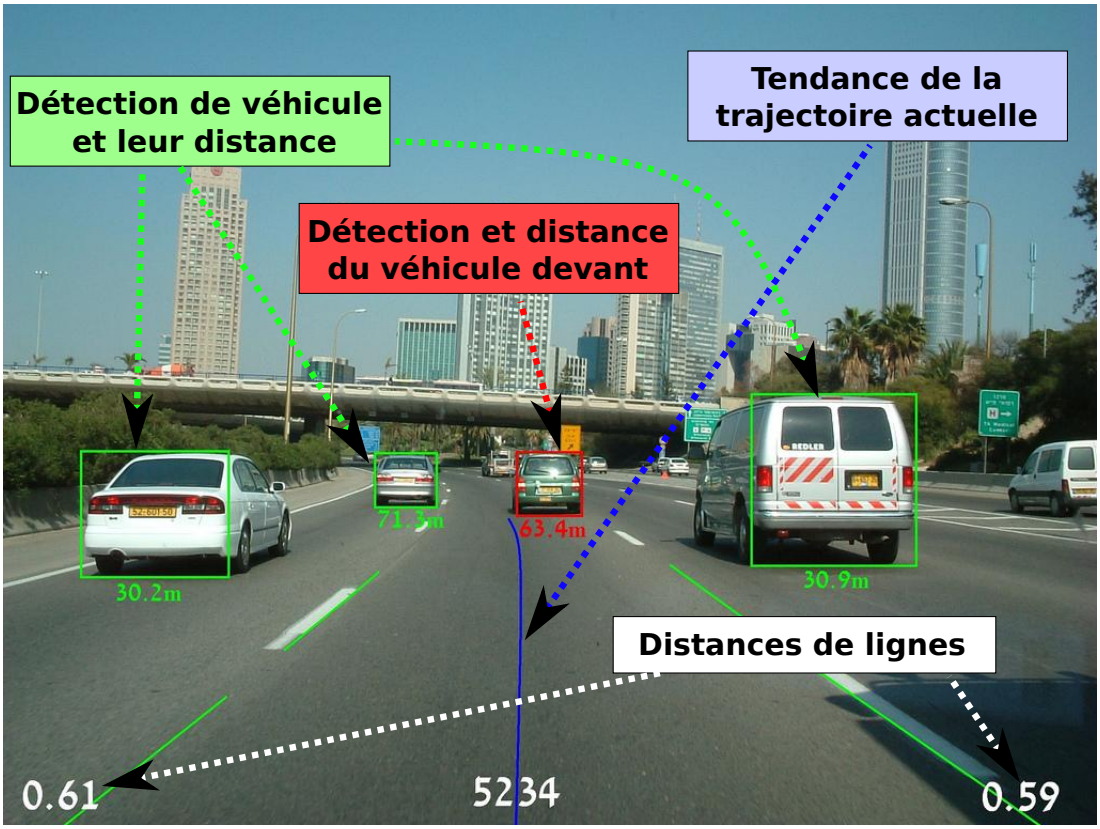

FIGURE 1.2 – Détection et analyse des événements dans la caméra Mobileye - Source : Mobileye®

— Création du concours en 2005, nommé le DARPA **Grand Challenge**. Lors de cet événement, l'objectif commun à tous les participants était d'effectuer un circuit prédéfini dans le désert avec un robot mobile autonome sur roues ;
— En 2007 ce même concours devient le DARPA **Urban Challenge** et se déroule en milieu urbain avec l'ajout d'épreuves comme des manœuvres, la circulation avec une densité de voitures élevée, etc ;
— Enfin, il devient le DARPA **Robotics Challenge** depuis 2012 et ajoute de nouvelles catégories, comme celle des robots humanoïdes. Ces derniers doivent par exemple effectuer des déplacements sur des terrains accidentés (rochers, escaliers, poutre, etc) ou encore accomplir des tâches comme l'ouverture d'une porte ou d'un sas.

Dans la catégorie originelle, celle des robots mobiles autonomes, depuis 2012 les objectifs sont les suivants :

— Se déplacer de manière autonome et effectuer un circuit en **moins de 10 heures** ;
— Utiliser le système de positionnement GPS et d'autres **capteurs disponibles pour les applications civiles** ;
— Fonctionner de manière **entièrement autonome**, sans intervention humaine. Le véhicule ne peut se servir que des informations qu'il perçoit et doit être capable de prendre ses propres décisions ;
— Ne pas toucher de manière intentionnelle un autre véhicule de la compétition.

On retrouve, parmi les compétiteurs, les plus grands laboratoires de robotique du monde.

L'industrie a contribué aussi à dynamiser la robotique en développant des prototypes de voitures autonomes. Parmi les principaux acteurs, on peut citer le projet de voiture autonome de Google® : Google Self-Driving Car Project [7]. Ce projet, démarré dans les années 2000, a pour but de créer une voiture complètement autonome. La société Google® a élaboré un premier prototype en 2009 basé sur une Toyota® Prius. Leur véhicule autonome fut le premier prototype roulant sur voies publiques : les autoroutes de Californie. Cet État des État-Unis fut d'ailleurs le premier à mettre en place des lois pour permettre de faire circuler les voitures autonomes sur les routes publiques. A l'origine du programme, en 2012, il y avait six Toyota®, une Audi® et trois Lexus® (figure 1.3) en circulation. On comptabilise à ce jour 2.4 millions de kilomètres parcourus par les véhicules autonomes de Google® sur les différentes autoroutes des États de Californie, Michigan, Floride et du Nevada.

C'est en 2014 que Google® fait évoluer son projet en dévoilant un nouveau prototype de voiture électrique, visible en figure 1.4. Cette voiture a été élaborée pour faire un apprentissage de la conduite en ville en utilisant des techniques de deep learning (utilisant par exemple des réseaux de neurones [8]). Le véhicule analyse la conduite de l'humain sur une longue période (plusieurs mois/années) en mettant en correspondance les actions du conducteur avec ce qu'il perçoit grâce à ses capteurs. Une fois entraîné, le véhicule est capable de reproduire le comportement de conduite de l'humain et de prendre des décisions en fonction des informations de ses capteurs.

7. https ://www.google.com/selfdrivingcar/

8. Les réseaux de neurones artificiels sont des réseaux fortement connectés de processeurs élémentaires fonctionnant en parallèle. Chaque processeur élémentaire calcule une sortie unique sur la base des informations qu'il reçoit. Toute structure hiérarchique de réseaux est évidemment un réseau.

FIGURE 1.3 – Voiture autonome de Google sur une base de Lexus RX450h

FIGURE 1.4 – Prototype de voiture électrique 100% Google : Google Self-Driving Car Project

Il existe beaucoup d'autres projets comme celui-ci tels que le prototype Mercedes® F 015 ou encore les voitures Tesla®.

La plupart de ces prototypes de véhicules sont basés sur la technologie LIDAR ou scanner laser 3D. On peut voir ce capteur sur les toits des voitures, figures 1.3 et 1.1. Ces capteurs constituent un des points critiques de commercialisation de ce type de véhicule au vu de leurs prix élevés (75000 USD).

1.2 La localisation pour la navigation autonome

La **navigation** de systèmes mobiles autonomes requiert trois fonctionnalités fondamentales :

— la **localisation** du mobile ;
— la **planification de la trajectoire** (connue sous le terme *path planning* dans la littérature) ;
— la **construction d'une carte** et l'**interprétation** de celle-ci.

La **construction de carte** désigne le fait de représenter le monde dans le repère du véhicule. La carte peut être locale ou globale. Dans le premier cas, on ne mémorise qu'une portion limitée de la carte autour du robot. Dans le deuxième cas, on construit la carte de manière incrémentale au fil des déplacements du mobile. Contrairement à la carte locale, la carte globale représente une grosse charge calculatoire (la carte s'étend et s'enrichit au fur et à mesure du parcours) mais a l'avantage de permettre de faire de la "fermeture de boucle" (ce processus permet de ré-observer des zones de la carte pour une meilleure correction des mesures). Cette nouvelle acquisition permet de corriger l'erreur des positions intégrée à la première observation. La figure 1.5 illustre cette méthode de fermeture de boucle.

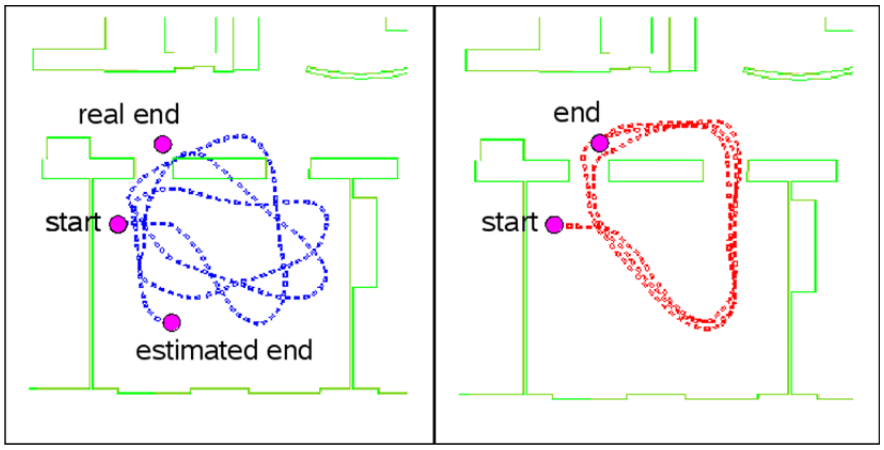

FIGURE 1.5 – Exemple de deux acquisitions SLAM du même parcours. A gauche, le résultat d'un SLAM sans fermeture de boucle. A droite, la position du robot est corrigée grâce à la méthode de fermeture de boucle.

La **planification de trajectoire** correspond au choix d'un chemin pour relier un point A à un point B en prenant en compte les obstacles et l'environnement. Cette trajectoire est remise en cause à chaque nouvel événement (détection d'un obstacle ou changement dans la scène).

La **localisation** permet de connaître la position d'un système dans son environnement. Cette dernière peut être relative si les capteurs utilisés mesurent un déplacement à partir d'une position initiale (IMUS, odomètres, etc). La position mesurée est absolue si les capteurs mesurent une position dans un repère global (le GPS par exemple).

Le choix des capteurs se fait en partie en fonction de l'environnement que l'on souhaite explorer, milieu extérieur ou intérieur. Dans le cadre des capteurs mesurant une position absolue, en milieu **extérieur**, le GPS fournit des données avec une précision de l'ordre de 8 mètres. Il peut être utilisé en mode différentiel (avec un 2eme GPS fixe) pour avoir une bonne connaissance du bruit dans la mesure et permettre de le retrancher dans les acquisitions du GPS embarqué. Ce système appelé aussi GPS centimétrique augmente grandement sa précision mais limite le champ d'action du mobile car plus le GPS du mobile est éloigné du GPS fixe, plus la connaissance du bruit est approximative. On peut aussi citer la triangulation d'ondes utilisées pour la communication comme le signal WIFI ou le GSM. Ce type de mesure est fortement bruité et offre une précision de localisation de l'ordre de 50 mètres en ville.

En milieu **intérieur**, les environnements sont généralement moins vastes mais les murs et plafonds rendent le signal GPS inutilisable. Il existe des capteurs de type motion capture qui offrent une mesure avec une précision de l'ordre du millimètre. Ce type de système implique l'instrumentation du milieu et du sujet dont on mesure les déplacements (installation de caméras infrarouges, port de marqueurs), présente un coût non négligeable et demande à être ré-étalonné régulièrement.

La navigation autonome nécessite une précision de localisation la plus fine possible car les erreurs de localisation sont intégrées dans la carte. Au vu de leur précision et de leur contraintes, aucun de ces capteurs ne peut être utilisé seul dans le cadre d'une application non contrainte (pouvant combiner des déplacements extérieurs et intérieurs). La solution consiste à multiplier les acquisitions à l'aide de différents capteurs et à fusionner les mesures obtenues.

Nos travaux s'inscrivent dans cette problématique de recherche de solution précise de **localisation** et de **construction de carte**, en utilisant un algorithme de SLAM (Simultaneous Localization and Mapping) (cf. section 2.1).

1.3 Le temps réel dans les systèmes embarqués

Une définition du temps réel, devenue référence dans le domaine, a été présentée dans [Stankovic, 1988]. Elle peut se traduire comme suit : "Le comportement correct d'un système dépend, non seulement des résultats logiques des traitements, mais aussi du temps nécessaire pour produire les résultats". Pour les auteurs la notion de **temps de réponse** est aussi importante que l'**exactitude de la réponse** du système. Les contraintes temps réel d'un système dépendent du contexte applicatif. Par exemple, pour un système ADAS de détection de piéton, le temps de traitement doit être élevé afin d'avoir le temps de réagir en cas de besoin. Le temps de réponse doit être de l'ordre de la milliseconde. En revanche, la navigation de sonde spatiale peut nécessiter un recalcul de trajectoire tous les mois. De manière générale, au vu de la complexité des environnements et de la vitesse des déplacements/mouvements, les besoins temps réels de la robotique se situent plus souvent entre la micro seconde et la seconde.

Un système temps réel peut se classer en fonction du niveau de criticité de son échéance (appelé *deadline* dans la littérature ce terme renvoi au temps de réponse limite à ne pas dépasser pour ne pas engendrer un défaut dans le système). Nous allons lister ce classement

et illustrer chaque définition par un exemple. Ce dernier permettra de comprendre les diverses conséquences d'un dépassement de deadline d'une fonction calculant les mouvements d'un robot :

- Les systèmes temps réel durs ou critiques sont ceux qui engendrent une défaillance du système si la deadline est dépassée. *Exemple : A un instant t, un robot bipède calcule trop lentement le mouvement que doit effectuer sa jambe, il perd l'équilibre et tombe. Si le système ne sait pas se relever en autonomie, la tâche en cours est annulée.*
- Les systèmes temps réel mous ou souples sont ceux qui tolèrent le dépassement de deadline. Cependant, ce débordement est susceptible de dégrader le fonctionnement du système. *A un instant t, un robot bipède calcule trop lentement le mouvement que doit effectuer sa jambe, il perd l'équilibre et tombe. Si le système peut se relever en autonomie, la tâche en cours est interrompue et poursuivie une fois le robot à nouveau sur ses deux jambes.*
- Les systèmes temps réel ouverts représentent ceux qui ne dégradent que légèrement le fonctionnement du système en cas de dépassement de date limite. *Exemple : Si le robot est équipé de six pattes, un retard sur la commande d'un de ses membres n'engendre pas de perte d'équilibre. La tâche de déplacement n'est pas interrompue.* [Izagirre et al., 2006] et [Saenz de argandona et al., 2007] présentent un autre exemple d'application temps réel qui ne dégrade que légèrement le fonctionnement du système en cas de dépassement de deadline puisque que le système n'intervient que sur le contrôle qualité de produits dans un processus d'hydroformage.

Dans le contexte des systèmes ADAS, cette contrainte temps réel est très forte puisque la **sécurité de l'humain** est souvent en jeu. Cependant, dans le domaine de la recherche, le but étant de valider le fonctionnement d'un concept et non de le commercialiser, la définition de cette contrainte n'est généralement pas renseignée. Traditionnellement, dans la littérature du traitement d'images, on considère qu'un système est temps réel si sa fréquence de fonctionnement est au dessus de 25 Hz. Ce choix est, d'une part, souvent aiguillé par la persistance rétinienne de l'humain (les démonstrations sont fluides à l'oeil). D'autre part, la fréquence des caméras de type global shutter [9] à bas coût sont souvent de cet ordre de grandeur.

Il est pourtant intéressant de faire la démarche d'**identifier les besoins** du système pour ensuite statuer sur une fréquence de traitemenent en adéquation avec le contexte de l'application. Si nous définissons la contrainte temps réel de notre application ADAS à 30 Hz (correspondant à une capacité de traitement de 30 images/s). En sachant qu'un véhicule motorisé peut atteindre la vitesse de 130 km/h (soit 36,11 m/s) sur autoroute, le système sera capable de traiter une image tous les 1.2 mètres. Pour un système de localisation par vision, fixer la contrainte temps réel à 30Hz est convenable puisqu'à cette fréquence, le changement de scène entre deux acquisitions d'images est limité (Les changements sont suffisamment réduits pour permettre de faire du suivi de points dans l'image).

A ce paramètre temps réel s'ajoute la notion primordiale de **latence**. La contrainte de latence et celle du temps de traitement sont indissociables l'une de l'autre. Prenons l'exemple d'une application de détection de piéton par vision illustrée par la figure 1.6. En entrée, la chaîne de traitement fait l'acquisition de l'image $img1$ à $t1$. Dans cet exemple, on considère que

9. La totalité des pixels sont échantillonnés au même instant. Contrairement au rolling shutter où les pixels sont échantillonnés ligne par ligne (caméra de téléphone portable ou webcam).

le système nécessite $2t$ pour traiter l'image. L'information de présence d'un piéton dans l'image $img1$ sera donc fournie à $t3$ (information transmise : il n'y a pas de piéton). Le système a une latence de $2t$, soit deux images. La fréquence de traitement est de $frequence\_acquisition/2$ puisque le système est trop lent pour pouvoir traiter toutes les images (une image sur deux est perdue). Si maintenant, le système est équipé de deux unités de traitement permettant d'effectuer deux fois la même détection en parallèle l'architecture sera alors capable de traiter toutes les images (les images paires sont traitées par le processeur 1 et les impaires par le second). La fréquence de traitement du système devient égale à $frequence\_acquisition$. En revanche, la latence reste égale à $2t$ puisque les processeurs nécessitent toujours un temps de traitement de $2t$ avant de pouvoir fournir une réponse.

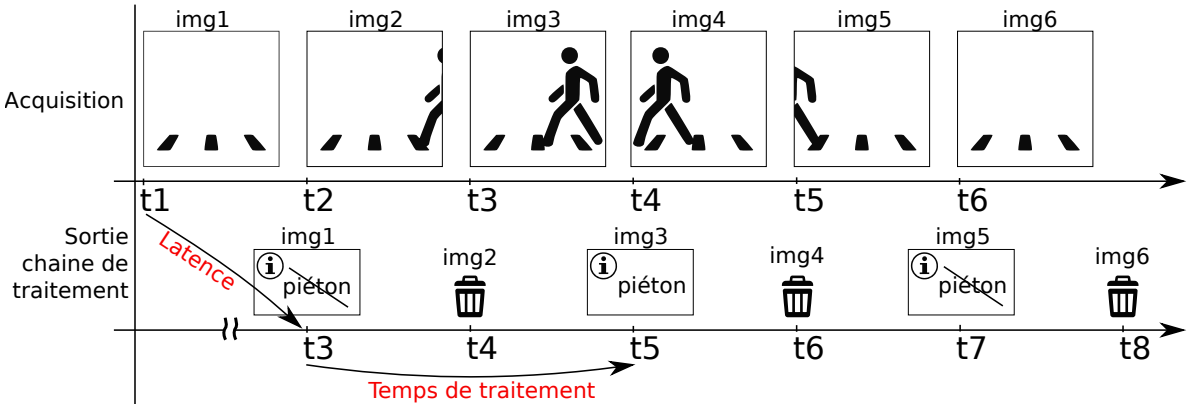

FIGURE 1.6 – Latence et temps de calcul

1.4 Projet DICTA

Les travaux de cette thèse s'inscrivent dans le cadre d'un **projet région Midi-Pyrénées** de type CLE (Contrat de recherche Laboratoires-Entreprises). Il est issu d'un partenariat entre une entreprise et deux laboratoires. Baptisé DICTA [10], il concerne la conception et la réalisation d'un capteur visuel intégré (appelé **smart camera**) pour des applications de transport terrestre. Ces capteurs permettent d'acquérir et d'analyser des données qui sont ensuite exploitées soit dans un système d'aide à la conduite de véhicules (ADAS embarqué sur une automobile, un bus, un camion, etc), soit dans un véhicule autonome (navettes électriques autonomes pour site urbain, robots, etc).

1.4.1 Les partenaires du projet DICTA

Le premier partenaire est l'entreprise **Delta Technologies Sud-Ouest** (DTSO [11]), spécialisée dans la conception et la fabrication de systèmes électroniques embarqués. Son champ d'expertise couvre les secteurs de l'**électronique embarquée**, les problématiques **temps réel**, l'informatique industrielle, l'**image** et la **vision**, la modélisation et simulation ou encore les réseaux. Les applications visées concernent l'aéronautique, le spatial, l'**automobile**, l'industrie, l'énergie et la télécommunication. Cette PME fait partie du groupe Systerel Imagine

10. DICTA : Development of an Integrated Camera for Transport Applications
11. http ://www.dtso.fr/

depuis 2 ans, groupe qui conçoit et met en œuvre depuis plus de 10 ans des solutions innovantes dans le domaine des systèmes critiques temps réel ou de sécurité.

Le **Laboratoire Génie de Production** (LGP [12]) est un laboratoire pluridisciplinaire dont les recherches s'inscrivent dans le thème général de la conception intégrée dans un cadre multi-échelle, considéré tant au plan multiphysique des processus analysés qu'à celui des niveaux d'engagement des ressources matérielles, logicielles et humaines (compétences et connaissances). Il est divisé en 4 équipes

— IMF : Interfaces et Matériaux Fonctionnels ;
— M2SP : Mécanique des Matériaux, des Structures et Procédés ;
— **DIDS : Décision et Interaction Dynamiques pour les Systèmes** ;
— SDC : Systèmes Décisionnels et Cognitifs.

L'équipe **DIDS**, impliquée dans ce projet, travaille sur la modélisation, la simulation, le pilotage, la conduite réactive et la supervision de systèmes dynamiques complexes. Ces travaux sont organisés en deux thèmes :

— L'interaction et le pilotage décisionnels ;
— La décision dynamique distribuée.

Le **Laboratoire d'Analyse et d'Architecture des Systèmes** (LAAS [13]) est une unité propre du CNRS (Centre Nationnal de Recherche Scientifique) rattachée à l'INstitut des Sciences de l'Ingénierie et des Systèmes (INSIS) et à l'INstitut des Sciences de l'Information et de leurs Interactions (INS2I). Depuis 2006, le laboratoire est l'un des 34 Instituts Carnot [14]. Il est fondé sur quatre champs disciplinaires : l'informatique, la robotique, l'automatique et les micro et nano systèmes. Au sein de ces disciplines, les thématiques de recherche du LAAS s'articulent autour de 8 départements scientifiques animant l'activité des 22 équipes de recherche, unités de base de la recherche du laboratoire, cf. figure 1.7.

La figure 1.7 présente les différentes équipes de recherche du LAAS :

— TSF : Tolérance aux fautes et Sûreté de Fonctionnement informatique ;
— VERTICS : Vérification de Systèmes Temporisés Critiques ;
— ISI : Ingénierie Système et Intégration.
— SARA : Services et Architectures pour les Réseaux Avancés ;
— CDA : Calcul Distribué et Asynchronisme.
— GEPETTO : Mouvement des Systèmes Anthropomorphes ;
— **RAP : Robotique, Action et Perception** ;
— RIS : Robotique et InteractionS.
— DISCO : DIagnostic, Supervision et COnduite ;
— MAC : Méthodes et Algorithmes en Commande ;
— ROC : Recherche Opérationnelle/Optimisation Combinatoire/Contraintes.
— MINC : MIcro et Nanosystèmes pour les Communications sans fils ;
— MOST : Microondes et Opto-microondes pour Systèmes de Télécommunications ;
— OSE : Optoélectronique pour les Systèmes Embarqués ;
— PHOTO : Photonique.
— N2IS : Nano Ingénierie et Intégration des Systèmes.

12. http ://www.enit.fr/fr/recherche/le-laboratoire-genie-de-production-1.html
13. https ://www.laas.fr/public/
14. le label Carnot a vocation à développer la recherche partenariale entre entreprises et laboratoires

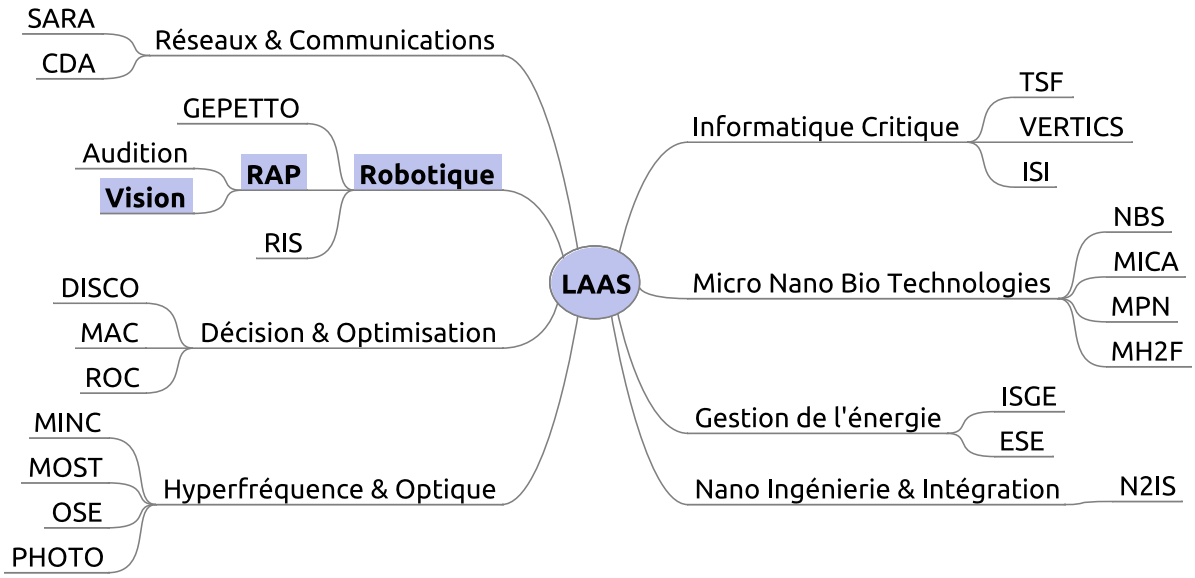

FIGURE 1.7 – Organigramme du LAAS

— NBS : Nanobiosystèmes ;
— MICA : Microsystèmes d'Analyse ;
— MPN : Matériaux et Procédés pour la Nanoélectronique ;
— MH2F : Micro et nanosystèmes HyperFréquences Fluidiques.
— ESE : Energie et Systèmes Embarqués ;
— ISGE : Intégration de Systèmes de Gestion de l'Energie.

Les recherches menées par l'équipe **Robotique, Action et Perception** (RAP) s'inscrivent principalement dans les domaines de la perception, du mouvement référencé capteurs, et des capteurs intégrés en robotique. Les contributions concernent la conception de fonctions, leur **prototypage**, ainsi que leur implémentation sur des robots et leur évaluation en conditions réelles. La plupart d'entre elles sont orientées vers la robotique en interaction avec les humains (analyse de scènes peuplées d'humains ; navigation coordonnée humain-robot ; cobotique) ou la robotique ubiquiste et l'intelligence ambiante (fusion de données en provenance de capteurs embarqués et déportés ; conception de capteurs intégrés communicants). Le traitement du signal sous-tend de nombreux développements et fait également l'objet de travaux séparés en amont.

1.4.2 Les objectifs du projet DICTA

Basé sur un transfert technologique dans le cadre d'un partenariat entreprises/laboratoires, le projet DICTA visait des contributions dans trois domaines :

— Les traitements qui, s'agissant d'une application critique pour l'homme, doivent satisfaire des contraintes critiques de robustesse, performance temps réel, fiabilité ;
— L'architecture matérielle de l'unité de traitement qui doit être adaptée aux traitements : en particulier, des opérations simples et répétitives peuvent être implantées matériellement, tandis que des calculs ou des traitements combinatoires complexes doivent rester implantés en logiciel ;

— La méthodologie de co-design qui permet de partitionner un traitement en sous-fonctions, les unes implantées matériellement, les autres logiciellement : on parle de co-design.

Le projet DICTA a pour but de partager des savoirs et expériences entre DTSO, le LAAS et le LGP. Ce partage s'est fait selon quatre axes :

— Transfert technologique entre la recherche et l'industrie ;
— Trouver une méthodologie de co-design applicable à plusieurs développements de type co-design (présenté en section 2.3) ;
— Veille technologique ;
— Élaboration d'une plateforme commune aux partenaires du projet.

DTSO souhaite diversifier son activité de conception de caméras dites intelligentes (ou smart-caméras) embarquant des fonctionnalités avancées de traitement d'images. DTSO souhaite ajouter de la valeur à ces produits en intégrant des fonctionnalités avancées de type ADAS. Les laboratoires **LAAS et LGP**, spécialistes du traitement de l'image, se proposent d'effectuer un **transfert technologique** sous forme d'IP [15] afin de valoriser leurs recherches dans ce secteur.

Tout au long de ces travaux, nous avons assuré une **veille technologique** dans différents domaines comme les méthodologies de co-design, les algorithmes de localisation, les plateformes disponibles sur étagère, etc. Nous avons tenu un rôle de consultant auprès de DTSO afin de les accompagner dans leur volonté d'intégrer des algorithmes de traitement dans leur caméra. Nous avons aussi fourni des études préliminaires et l'état de l'art pour différentes applications (implantation co-design de détection de lignes, de compression d'images sans perte, etc). Enfin, nous avons livré un IP de détection de points d'intérêt FAST (cf. section 2.1.5) et un IP descripteur de points d'intérêt BRIEF (cf. section 2.1.6). Nous avons intégré l'accélérateurs FAST dans une chaîne de démonstration effectuant une estimation du flux optique [16] par la méthode Lucas-Kanade [Kanade, 1981] sur une architecture mixte (Zedboard®, cf. section 3.2.4). Dans cette architecture, nous nous sommes basé sur la fonction C : *cvCalcOpticalFlowPyrLK()* de la bibliothèque OpenCV [17] pour résoudre les équations du flux optique que nous avons alimentée par l'accélérateur de détection de points d'intérêt FAST.

Enfin, le LAAS et le LGP souhaitent avoir une **plateforme de développement commune**. Ceci afin de faciliter la conception et les applications futures. Elle doit être adaptée à notre application mais à la fois suffisamment générique. De son côté, DTSO souhaite faire évoluer sa caméra en changeant l'unité de traitement. Nous avons donc exploré les solutions existantes en prenant en compte les capacités de traitement, la consommation, la communauté d'utilisateurs, l'encombrement du composant, les périphériques disponibles, les évolutions possibles, etc.

1.5 Objectifs des travaux

La figure 1.8 récapitule les objectifs et sous-objectifs de cette thèse. Tout d'abord, nous avions deux objectifs, ceux propres au projet DICTA précédemment évoqués (cf. section 1.4.2)

15. Intellectual Property.
16. Le flux optique (ou défilement visuel) est le mouvement apparent des objets, surfaces et contours d'une scène visuelle, causé par le mouvement relatif entre un observateur (l'œil ou une caméra) et la scène.
17. Bibliothèque de référence dédiée au traitement d'images.

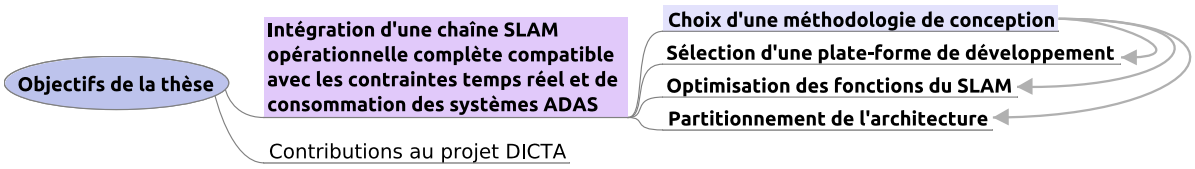

FIGURE 1.8 – Objectifs de cette thèse.

et le défi majeur d'implantation d'une chaine 3D EKF SLAM sur une architecture embarquée.

Comme illustré dans le diagramme de la figure 1.8, quatre sous-objectifs découlent de l'objectif principal d'implantation d'une chaîne de localisation et cartographie temps réel sur architecture embarquée.

Le premier objectif sous-jacent était de **définir une plateforme de traitement**. Le but était de choisir cette dernière dès le début du projet afin d'optimiser les développements en fonction des ressources disponibles. L'enjeu était de taille puisque les performances temps réel (la fréquence de traitement maximale) sont fortement impactées par ce choix.

Le deuxième objectif secondaire était d'identifier et **optimiser les fonctions du SLAM**. L'algorithme de SLAM requiert de grosses capacités de calcul et les architectures embarquées disposent de ressources limitées par rapport à des ordinateurs standards (type PC de travail avec, par exemple, un micropresseur intel® i5 et 8Go de mémoire RAM). Il était donc nécessaire d'accélérer les fonctions les plus demandeuses en temps de traitement pour permettre à l'architecture cible d'exécuter le SLAM et de satisfaire les contraintes temps réel.

Le troisième sous-objectif, fortement lié aux deux précédents, était d'effectuer un **partitionnement de l'application**. En effet, afin d'optimiser les calculs, nous devions proposer une répartition des traitements en fonction des ressources disponibles (cf. section 2.3.3).

Enfin, pour optimiser les développements, nous nous étions fixé comme objectif de **choisir une méthodologie de co-design**. Cette méthodologie est la clé de voûte permettant de réaliser l'implantation de la chaine 3D EKF-SLAM sur architecture embarquée. En effet, du choix de la plateforme, en passant par la modélisation et le partitionnement du système, la méthodologie de co-design permet de guider le développement :

1. Elle permet d'aider à choisir une plateforme de traitement grâce à l'identification des contraintes du système (cf. section 3.1.2) ;
2. Elle aide à modéliser le système afin d'identifier des traitements parallélisables (cf. section 3.2 ;
3. Elle participe au choix du partitionnement en contribuant à identifier les spécifications matérielles (cf. section 3.1.3).

1.6 Contributions

Nos recherches nous ont amené à contribuer dans le domaine de la **vision** au service de la robotique et plus particulièrement dans celui des sytèmes embarqué ADAS. Nous avons développé le **premier prototype d'une chaîne 3D EKF SLAM monoculaire embarquée complète sur une architecture mixte** en utilisant une **méthodologie de co-design**. Elle est capable de maitenir une vitesse de traitement **constante** de **24.9 images/s** et la **latence** totale du système est de **une image** tout en consommant moins de **5W**. La plateforme que

nous avons sélectionnée est équipée d'une unité de traitement de type micro-processeur et d'une unité matérielle de type FPGA (Field-Programmable Gate Array). Nous avons d'une part **optimisé** la partie logicielle spécifiquement pour le microprocesseur cible (ARM - cortex A9), d'autre part, nous avons proposé un **partitionnement front-end** [18]**/back-end** [19] : une partie traitant les données pixel de bas niveau et la deuxième, le back-end, se chargeant des données de haut niveau. Nous avons **accéléré** les fonctions de la partie purement vision du SLAM sur FPGA (traitement bas niveau). Pour cela, plusieurs IPs matériels ont été développés :

- Une architecture optimisée de **détection** de points d'intérêt basée sur l'algorithme FAST [20] [Brenot et al., 2015] ;
- Une architecture optimisée de **description** de points d'intérêt basé sur l'algorithme BRIEF [21] [Brenot et al., 2016] ;
- Une architecture de division horizontale et verticale de l'image (**tesselation**) [Brenot et al., 2016] ;
- Une architecture originale de **mise en correspondance d'amers** dans des images successives [Brenot et al., 2016] ;
- Une architecture originale de **suppression de non-maxima locaux** (NMS [22]) versatile [Brenot et al., 2015].

En parallèle de cette thèse, une fonction matérielle d'accélération des produits matriciels a été réalisée par Daniel Törtei/Tertei [Tertei et al., 2014], doctorant au LAAS qui a contribué aussi à l'intégration de la chaîne SLAM.

1.7 Publications

1.7.1 Conférences Internationales avec comité de lecture

F. BRENOT, P. FILLATREAU et J. PIAT "FPGA based accelerator for visual features detection" IEEE workshop on Electronics, Control, Measurement, Signals, Liberec, République Tchèque 2015

F. BRENOT, J. PIAT et P. FILLATREAU "FPGA based hardware acceleration of a BRIEF correlator module for a monocular SLAM application" présenté sous forme de poster à ICDSC 2016, International Conference On Distributed Smart Camera, Paris, France, 2016.

18. Partie ayant pour rôle de lire et de mettre en forme les signaux.
19. Terme désignant un étage de sortie d'un logiciel devant produire un résultat.
20. Features from Accelerated Segment Test
21. Binary Robust Independent Elementary Features
22. Non Maxima Suppression

Chapitre 2

Du SLAM au SLAM embarqué

2.1 SLAM

Les travaux de cette thèse portent sur l'intégration d'une chaîne complète de localisation et de cartographie simultanées (SLAM [1]) sur une architecture mixte matérielle/logicielle dédiée. Dans ce chapitre, nous allons expliquer le principe général du SLAM (section 2.1.1). Nous dresserons un état de l'art des différentes approches de SLAM existantes. Puis, nous présenterons notre méthode basée sur un Filtre de Kalman Etendu (ou EKF [2]) développée au laboratoire LAAS-CNRS (section 2.1.1) et la positionnerons par rapport aux solutions existantes. Enfin, nous exposerons la méthodologie de co-design et l'architecture utilisée afin d'implanter un EKF-SLAM sur une architecture embarquée.

2.1.1 Principe général du SLAM

Dans un environnement connu ou inconnu, l'être humain est capable d'établir une carte et de s'y localiser. Pour ce faire, il utilise toutes les informations disponibles comme son déplacement ou encore ses sens. En robotique, la fonction de localisation et cartographie simultanées est appelée SLAM qui est l'acronyme anglais de Simultaneous Localisation and Mapping. Comme pour l'humain, le principe de ce dernier est de localiser précisément un robot dans son environnement tout en construisant une carte de cet environnement à partir de ses capteurs proprioceptifs et extéroceptifs.

Les capteurs proprioceptifs effectuent une mesure de l'état du robot (vitesse, orientation ...). On peut trouver dans cette catégorie les capteurs de type odométriques ou inertiels. Par opposition, les capteurs dit extéroceptifs permettent de percevoir l'environnement du robot. On trouve dans cette catégorie les SONAR, LIDAR ou caméras.

Dans l'exemple présenté en figure 2.1, à chaque instant, le robot (représenté par un triangle) perçoit des amers (caractéristiques invariantes de l'environnement, notés L_n). Au fur et à mesure qu'ils sont découverts, ces amers sont intégrés à la carte construite par le robot. Le fait de ré-observer ces amers au cours du déplacement du robot permet de déterminer son déplacement dans la carte mais aussi d'affiner la position des amers représentés dans la carte.

D'après [Roussillon, 2013], toute approche de SLAM métrique basée sur des amers implique l'implantation des grandes fonctions suivantes :

- Détection d'amers sur la base des données observées par les capteurs extéroceptifs du robot et estimation de leur position relative au robot.
- Estimation des déplacements du robot dans un repère du monde.
- Association de données : mise en correspondance des amers détectés depuis des positions différentes.

1. Simulatneous Localisation and Mapping.
2. Acronyme de l'anglais Extended Kalman Filter.

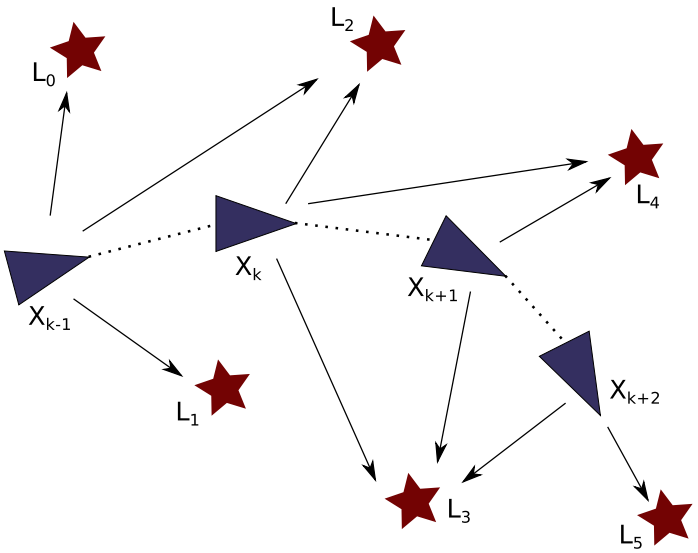

FIGURE 2.1 – Illustration du principe du SLAM. A chaque instant le robot (représenté par une flèche) perçoit des amers (L_n pour la traduction anglaise de landmarks, symbolisés par des étoiles) grâce à ses capteurs. Si ces amers ont été déjà observés auparavant, le robot effectue une ré-observation de ces points. D'après ces données, il peut affiner sa position dans la carte et celle des amers.

— Estimation : mise à jour de la position des amers et de la localisation du robot.

La variabilité de réalisation de ces grandes fonctions est très grande et a été largement explorée par la communauté.

— Un amer est un élément de l'environnement auquel une position géométrique peut être associée. Il peut être représenté dans la carte de l'environnement par différentes primitives géométriques (points, segments de droites, plans, objets quelconques ou autres primitives géométriques plus complexes). Notons que la notion de descripteurs d'amers (signatures caractérisant chaque amer calculées avec leur voisinage respectif) joue un rôle important dans le processus de mise en correspondance d'amers (cf. section 2.1.6).
— Tous les moyens qui permettent d'estimer les déplacements du robot sont à priori exploitables pour une solution de SLAM.

La conception de nouvelles techniques de SLAM est un domaine de recherche en constante progression et on retrouve une multitudes d'approches différentes. Parmi elles, on peut distinguer deux familles distinctes, les SLAM par optimisation et les SLAM par filtrage. Nos travaux portent sur un SLAM par filtrage basé sur un filtre étendu de Kalman. Nous allons faire un état de l'art des méthodes existantes.

2.1.2 Le filtre de Kalman Etendu pour le SLAM

Le filtre de Kalman [Kalman, 1960] permet d'estimer l'état d'un système à partir de mesures incomplètes ou bruitées. Pour qu'il soit applicable, les équations permettant de modéliser le problème doivent être linéaires.

La solution de résolution du SLAM basée sur un filtrage de Kalman consiste à éliminer l'effet du bruit dans les mesures grâce à une connaissance de la dynamique de la cible. Pour cela

le filtre de Kalman linéarise les fonctions d'évolution et d'observation autour de l'estimation courante pour appliquer un filtre de Kalman linéaire. Kalman est un filtre récurssif, il travaille sur l'état précédent du système et les mesures en cours. L'une des applications références de SLAM par filtrage de Kalman a été réalisée par Smith et Cheeseman dans [Smith et al., 1991]. Les auteurs ont proposé d'utiliser un Filtre de Kalman Etendu ou EKF [3] (pour Extended Kalman Filter).

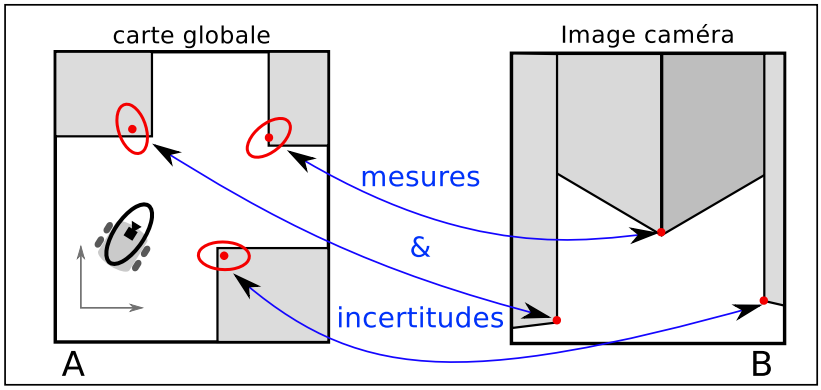

FIGURE 2.2 – Figure A, la carte globale du SLAM avec la position du robot, celle des amers et leurs incertitudes respectives. Figure B, la scène perçue par la caméra et les observations de points d'intérêt dans l'image.

Les étapes principales et générales du SLAM par filtrage EKF peuvent se décomposer en 4 phases. La figure 2.3 illustre ces étapes à l'aide d'un exemple simple de SLAM visuel 2D issu de [Davison, 2003] et illustré par la figure 2.2. Les figures A, B, C, D montrent l'évolution de la carte globale du SLAM où la position et l'orientation du robot sont représentées par un repère de coordonnées euclidiennes (en noir) et les obstacles par des rectangles gris.

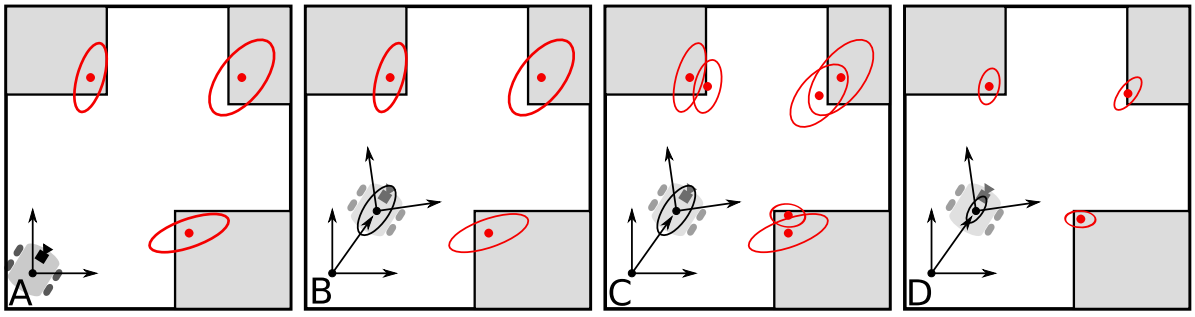

FIGURE 2.3 – Etapes d'un SLAM visuel

1. (A) Le robot observe trois nouveaux points d'intérêt (les coins des obstacles). Il les initialise dans la carte en tant qu'amers en estimant leurs positions dans le repère monde avec leurs incertitudes. Ces dernières sont représentées par des ellipses rouges dans la figure (la perception monoculaire ne permet pas d'estimer la profondeur de manière précise) ;

3. Dans le cas où les équations permettant de modéliser un problème ne sont pas linéaires, le filtre de Kalman Etendu permet de linéariser localement le problème afin d'appliquer les équations du filtre de Kalman classique.

2. (B) Le robot change de position et prédit sa nouvelle position sur la base de son modèle de mouvement et de données capteurs proprioceptifs ;
3. (C) Le robot prédit les nouvelles positions des amers, les transpose en position de points d'intérêt dans l'image, observe ses points d'intérêt et détermine la distance entre l'observation et la prédiction ;
4. (D) Une étape de filtrage permet de mettre à jour l'état du système (carte des amers et position/orientation du robot dans la scène) et d'affiner le modèle stochastique associé.

Nous allons maintenant présenter une formulation du SLAM EKF basée sur [Sola et al., 2007] et [Gonzalez, 2012]. Le EKF-SLAM est basé sur deux éléments principaux :
— le vecteur d'état X_k contenant les valeurs moyennes de l'état du système à l'instant k ;
— P la matrice de covariance de X_k à l'instant k représentant l'erreur d'estimation.

Vecteur d'état La **carte** SLAM consiste en un vecteur d'état :

$$\hat{X} = \begin{bmatrix} \hat{R} \\ \hat{M} \end{bmatrix} = \begin{bmatrix} \hat{R} \\ \hat{L}_1 \\ \vdots \\ \hat{L}_i \\ \vdots \\ \hat{L}_n \end{bmatrix} \tag{2.1}$$

Avec X le vecteur d'état du système (robot) qui contient 3 types d'information :
— les éléments du **modèle du mouvement** du robot (comme la vitesse linéaire et angulaire ou encore l'accélération) (contenus dans R) ;
— Sa **pose** c'est à dire sa position p et son orientation q définie sous la forme de quaternion (contenus dans R) ;
— et M contenant les positions de l'ensemble des **amers** (avec la notation L pour Landmark).

Dans le cas où les amers admettent une paramétrisation euclidienne, X est de taille $[1; 4 + 3n]$ où n est le nombre d'amers.

Matrice de covariance La matrice P est la matrice de covariance. Elle contient l'erreur d'estimation.

$$P = \begin{bmatrix} P_{RR} & P_{RM} \\ P_{MR} & P_{MM} \end{bmatrix} = \begin{bmatrix} P_{RR} & P_{RL_1} & \dots & P_{RL_i} & \dots & P_{RL_n} \\ P_{L_1R} & P_{L_1L_1} & \dots & P_{L_1L_i} & \dots & P_{L_1L_n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{L_iR} & P_{L_iL_1} & \dots & P_{L_iL_i} & \dots & P_{L_iL_n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{L_nR} & P_{L_nL_1} & \dots & P_{L_nL_i} & \dots & P_{L_nL_n} \end{bmatrix} \tag{2.2}$$

Cette matrice P est carrée et symétrique ce qui implique que $P_{RR} = P_{RR}^T$, $P_{RM} = P_{MR}^T$ et $P_{MM} = P_{MM}^T$. P_{RR} est l'incertitude sur la pose du système mobile, P_{RM} décrit les intercorrélations entre le robot et les amers et P_{MM} les intercorrélations entre amers, avec $P_{L_iL_i}$

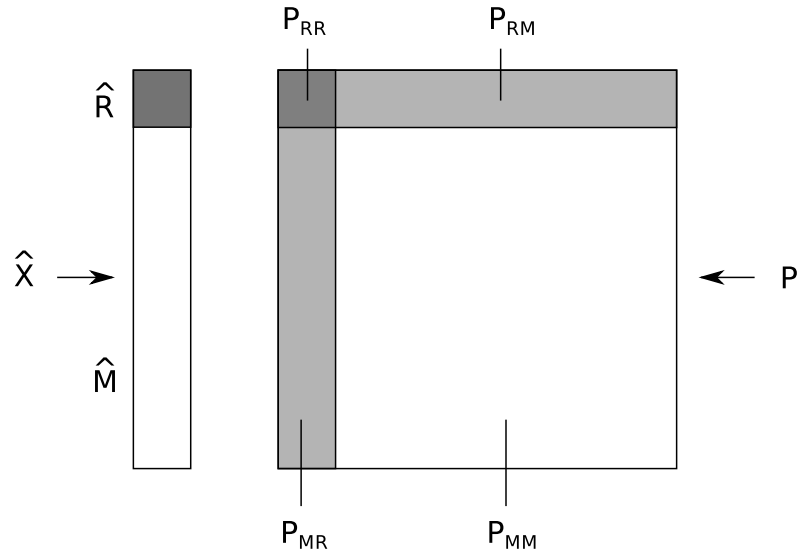

FIGURE 2.4 – Matrice de covariance P.

l'incertitude sur la pose de l'amer L_i. La figure de 2.4 illustre le contenu de la matrice de covariance P.

Le processus de filtrage se déroule en trois étapes :

1. on **prédit** le mouvement du robot selon une loi de commande connue à l'avance et/ou de données issues des capteurs embarqués ;
2. on effectue la **correction** de cette prédiction en exploitant les ré-observations des points d'intérêt correspondant aux amers déjà connus dans la carte ;
3. on **initialise**, si besoin, de nouveaux amers dans la carte.

Prédiction Dans cette étape, on prédit le mouvement du robot selon une loi de commande connue à l'avance et de données issues des capteurs embarqués. La prédiction de la pose du robot l'instant $k+1$ suit la fonction :

$$R^+ = f(R, u) \tag{2.3}$$

où u est le vecteur de contrôle supposé gaussien. u peut être donné par la commande ou être observé par des capteurs proprioceptifs.

D'après la formulation de l'EKF, l'étape de prédiction est définie par :

$$\hat{R}^+ = f(\hat{R}, \hat{u}) \tag{2.4}$$

$$P^+_{RR} = F_R.P_{RR}.F_R^T + F_u.U.F_u^T \tag{2.5}$$

$$P^+_{RM} = F_R.P_{RM} \tag{2.6}$$

$$P^+_{MM} = P_{MM} \tag{2.7}$$

où les matrices jacobiennes sont :

$$F_R = \frac{\partial f}{\partial R^T}|_{\hat{R},\hat{u}} \qquad F_u = \frac{\partial f}{\partial u^T}|_{\hat{R},\hat{u}} \tag{2.8}$$

Suite à cette étape de prédiction, une boucle de correction est appliquée au système. Le nombre d'itérations est égal au nombre i d'amers observés.

Correction L'étape de correction du filtre de Kalman se fait par l'application d'un terme additif, produit de l'*innovation* par le *gain*. Dans le cadre du SLAM EKF monoculaire, l'innovation est la différence entre l'observation d'un amer dans l'image et la prédiction de celle-ci. Cette différence est calculée dans le plan image. La mesure (y) d'un amer (i) se fait par la fonction d'observation :

$$y_i = h(R, C, L_i) + v \tag{2.9}$$

où v est un bruit blanc gaussien de moyenne nulle.

D'après la formulation de l'EKF, l'étape de correction est définie par les équations suivantes :

$$z_i = y_i - h(\hat{R}, C, \hat{L}_i) \tag{2.10}$$

$$Z_i = H_i.P.H_i^T + R \tag{2.11}$$

$$K_i = P.H_i^T - Z_i^{-1} \tag{2.12}$$

$$\hat{X}^+ = \hat{X} + K_i.\hat{z}_i \tag{2.13}$$

$$P^+ = P - K_i.Z_i.K_i^T \tag{2.14}$$

où K et la matrice de gain du filtre de Kalman, Z la matrice de covariance de l'innovation et la matrice jacobienne H_i est :

$$H_i = \frac{\partial h(\hat{R}, C, \hat{L}_i)}{\partial X^T}|_{\hat{X}} \tag{2.15}$$

Initialisation L'initialisation consiste en l'ajout d'un nouvel amer dans le vecteur d'état carte, tel que :

$$\hat{X} = \begin{bmatrix} X \\ L_{n+1} \end{bmatrix} \tag{2.16}$$

Afin de pouvoir initialiser un nouvel amer dans la carte, la fonction d'observation $h()$ doit être inversée. Posons $g()$ cette fonction tel que :

$$L_{n+1} = g(R, C, y_{n+1}) \tag{2.17}$$

Or, comme la profondeur de l'amer n'est pas observable en vision monoculaire, la fonction $h()$ n'est pas inversible. Cette profondeur doit être donnée à priori, et $g()$ se récrit :

$$L_{n+1} = g(R, C, y_{n+1}, c) \tag{2.18}$$

On peut maintenant définir la moyenne par :

$$\hat{L}_{n+1} = g(R, C, y_{n+1}, \hat{c}) \tag{2.19}$$

et les jacobiennes par :

$$G_R = \frac{\partial g(\hat{R}, C, \hat{y}_{n+1}, \hat{c})}{\partial c} \tag{2.20}$$

$$G_{y_{n+1}} = \frac{\partial g(\hat{R}, C, \hat{y}_{n+1}, \hat{c})}{\partial y_{n+1}} \tag{2.21}$$

$$G_c = \frac{\partial g(\hat{R}, C, y_{n+1}, \hat{c})}{\partial c} \tag{2.22}$$

La matrice de covariance P s'écrit :

$$P = \begin{bmatrix} P & P_{L_{n+1}X}^T \\ P_{L_{n+1}X} & P_{L_{n+1}L_{n+1}} \end{bmatrix} \tag{2.23}$$

avec

$$P_{L_{n+1}X} = G_R.P_{RX} \quad et \quad P_{L_{n+1}L_{n+1}} = G_R.P_{RR}.G_R^T + G_{y_{n+1}}.R.G_{y_{n+1}} + G_c.C.G_c^T \tag{2.24}$$

2.1.3 Etat de l'art des SLAM par vision embarqués

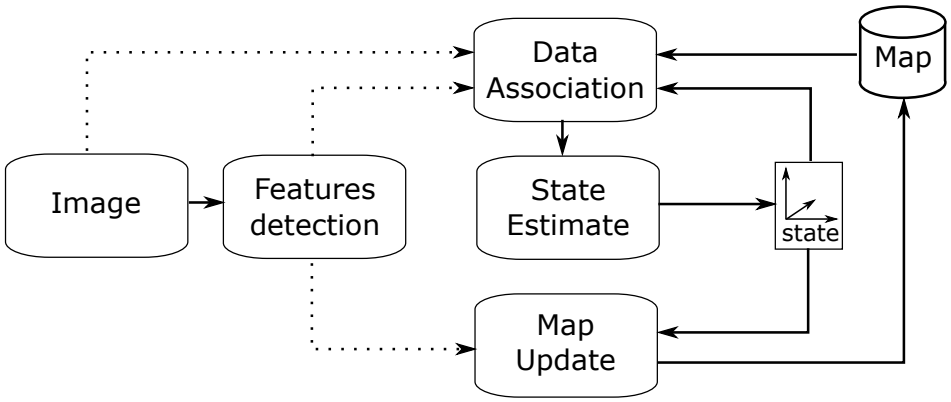

FIGURE 2.5 – Principe du SLAM visuel

SLAM vision Le SLAM basé vision utilise une caméra comme capteur extéroceptif pour acquérir des données sur l'environnement. On extrait dans les images des caractéristiques (par exemple et généralement des points d'intérêt ou coins - notion précisée dans la section 2.1.5). L'observation de ces points d'intérêt tout au long du déplacement de la caméra permet de construire une carte globale des amers de l'environnement et d'y estimer la position et l'orientation de la caméra /du robot (figure 2.5).

Ainsi, la mise en correspondance des points d'intérêt acquis dans des images successives permet d'estimer les positions des amers correspondants dans une carte globale. Ceci permet

d'estimer la position et l'orientation du robot dans cette carte. Le processus du SLAM peut donc être divisé en deux principales étapes.

La première (fonction de perception qui traite les données pixel, front-end) permet d'extraire les points d'intérêt (ou *features* dans la littérature) dans les images acquises. Elle permet aussi la mise en correspondance des points d'intérêt perçus dans les images successives.

La seconde (étape d'estimation) effectue la mise à jour de la position et de l'incertitude des amers correspondants (ou *landmarks* dans la littérature) et en déduit la position, l'orientation et l'incertitude du robot dans la carte construite incrémentalement.

SLAM vision embarqués La complexité du problème du SLAM entraîne une charge de calcul difficilement supportable par les architectures embarquées. De sucroît, l'utilisation de la vision dans la résolution de ce problème entraîne des calculs encore plus exigeants en terme de fréquence de traitement et de latence. De nos jours, avec l'évolution des moyens de traitement embarqués (amélioration de la consommation, de la capacité de calcul et de encombrement), on commence à voir apparaître des implantations de chaînes SLAM sur architecture embarquée (elles sont encore rares). Dans [Gonzalez, 2012], l'auteur fait une présentation des différentes implantations. C'est en 2007 que la première implantation d'un SLAM embarqué est apparue, [Abrate et al., 2007]. Elle met en oeuvre un EKF-SLAM, sur un robot Khepera en utilisant uniquement des capteurs infrarouges. La faible qualité de mesure des capteurs est compensée par le filtre de Kalman étendu. Le système manque de précision du fait des capteurs utilisés, de leurs précisions et de leurs portées.

Dans [Nikolic et al., 2014], les auteurs implantent une autre solution sur architecture embarquée. Cette dernière est équipée d'un ZynQ® contenant une unité de traitement logicielle (CPU ARM Cortex A9) et une matérielle (FPGA [4]). Les auteurs mettent en oeuvre seulement la détection de coins et la synchronisation des capteurs proprioceptifs (IMU) sur la partie matérielle tandis que toutes les autres tâches de traitement d'images et SLAM sont calculées en logiciel. Le SLAM est réalisé en utilisant une méthode d'optimisation (cf. section 2.3.1) et nécessite des traitements déportés sur un ordinateur. Le système présente des performances en fréquence de $20Hz$ et une erreur de trajectoire de 0.7% pour une distance parcourue de 700m.

Une implantation de la méthode SVO (Semi-direct Visual Odometry) basée sur l'**optimisation** est présentée dans [Faessler et al., 2016]. Dans cette solution, l'intérêt est porté sur le vol autonome d'un drône. Une plateforme embarquée exécute l'odométrie visuelle (SVO) et une station de traitement au sol reconstruit la carte 3D (REMODE ou REal-time, probabilistic, MOnocular, DEnse reconstruction). La partie odométrie est très légère en traitement mais nécessite une reconstruction très coûteuse (optimisation GPU obligatoire pour pouvoir satisfaire les contraintes temps réel) afin de maintenir une carte globale de l'environnement.

Dans [Bonato et al., 2008], les auteurs implantent le descripteur de points d'intérêt SIFT (Scale-Invariant Feature Transform) sur hardware (FPGA Altera Stratix II) afin de l'utiliser dans un EKF-SLAM avec quatre caméras QVGA (320 x 240 pixels). Leur application produit une carte 2D de l'environnement avec une paramétrisation des amers réduite (Euclidienne tridimensionnelle uniquement).

Dans [Botero et al., 2012], les auteurs proposent une implantation des fonctions de traitement de la vision du EKF-SLAM (détection de point de Harris et la mise en correspondance)

4. Field Programmable Gate-Array, cf. section 2.2.1

à fréquence pixel sur FPGA Virtex 5. Le système arrive à traiter les calculs à une fréquence de $25Hz$.

A notre connaissance, il n'existe actuellement pas d'implantation d'un 3D **EKF**-SLAM basé vision monoculaire sur un seul et même SoC [5] FPGA. La plupart des travaux présentent des architectures accélérant certaines fonctions sur architecture matérielle mais ne proposent pas de solution temps réel complète.

2.1.4 Chaînes SLAM vision embarquées opérationnelles au LAAS

RT-SLAM et C-SLAM sont les deux chaînes opérationnelles les plus récentes au LAAS. RT-SLAM a été implanté sur un robot (appelé *Mana* basé sur une plateforme Segway RMP400). Le processeur mobilisé pour cet algorithme est un Intel i7 cadencé à 3.3GHz. La consommation minimale pour ce type d'architecture est de 140W. L'encombrement et de l'autonomie ne permet pas son utilisation dans le cadre d'un système embarqué de type ADAS. C-SLAM, dérivé de RT-SLAM, est moins coûteux en calcul et plus orienté vers l'embarqué.

RT-SLAM Un EKF-SLAM vision temps réel, basé sur [Davison et al., 2007] a été développé et amélioré au LAAS [Roussillon et al., 2011]. Développé en C++, il est nommé RT-SLAM (pour Real Time-SLAM) et est basée sur la SLAM Toolbox [6]. La version logicielle est disponible à l'adresse : RT-SLAM [7]. C'est un algorithme 3D EKF-SLAM utilisant une caméra pour observer l'environnement et des capteurs inertiels pour estimer le mouvement. Il est présenté dans la thèse [Roussillon, 2013] par Cyril Roussillon.

La carte de RT-SLAM contient l'état du filtre, soit l'estimation de l'état du système et l'estimation de son incertitude. Le nombre d'amers contenus dans la carte de l'environnement est configurable. La paramétrisation des points d'intérêt est configurable (euler, IDP, AHP) et peut évoluer au cours du temps.

Une stratégie **Active Search** (détaillée en section 2.1.7) avec le détecteur de points d'intérêt Harris [Harris and Stephens, 1988] permet d'observer et de mettre en correspondance les points d'intérêt.

De nouveaux amers sont **initialisés** dans la carte en suivant l'algorithme de one-point RANSAC (random sample consensus) - [Fischler and Bolles, 1981]. Pour rappel, l'algorithme classique RANSAC (RANdom SAmple Consensus) fonctionne comme suit :

- Si un modèle nécessite n éléments pour être estimé (par exemple deux points sont nécessaires pour modéliser une droite), on choisit aléatoirement n éléments de l'ensemble et on estime le modèle correspondant.
- On teste ensuite la compatibilité de tous les autres éléments avec le modèle obtenu et on compte combien d'entre eux sont compatibles avec ce dernier.
- On réitère m fois l'opération et on choisit l'ensemble d'éléments compatibles le plus grand.

Cette méthode permet d'effectuer une correction groupée des amers initialement compatibles. Cette méthode permet d'accélérer les traitements et compense largement le sur-coût de calcul lié au test de RANSAC.

5. System on a Chip
6. https ://github.com/damarquezg/SLAMTB
7. https ://www.openrobots.org/wiki/rtslam/

L'implantation C++ de RT-SLAM atteint 60Hz (avec des images VGA, 640x480) sur un PC avec un microprocesseur Intel® i7 (en mono-thread) en maintenant 20 amers dans la carte.

C-SLAM Dans [Gonzalez et al., 2011], les auteurs de C-SLAM proposent une version simplifiée de l'algorithme RT-SLAM avec une vitesse de déplacement du robot constante (les données inertielles ne sont pas utilisées).

La paramétrisation des points d'intérêt est de type IDP (Inverse-Depth Point)- [Montiel, 2006] où les amers sont représentés par 6 paramètres contre 7 en paramétrisation AHP. D'après [Sola, 2007], cette **paramétrisation** IDP est moins robuste. Les auteurs présentent dans [Botero et al., 2012], 2 plateformes FPGA couplées (Virtex5 et Virtex6 - Xilinx) [8] embarquant C-SLAM qui atteint 24Hz (24 images/s avec une résolution 640×480) en maintenant 20 amers dans la carte de l'environnement (soit 20x24 corrections/seconde).

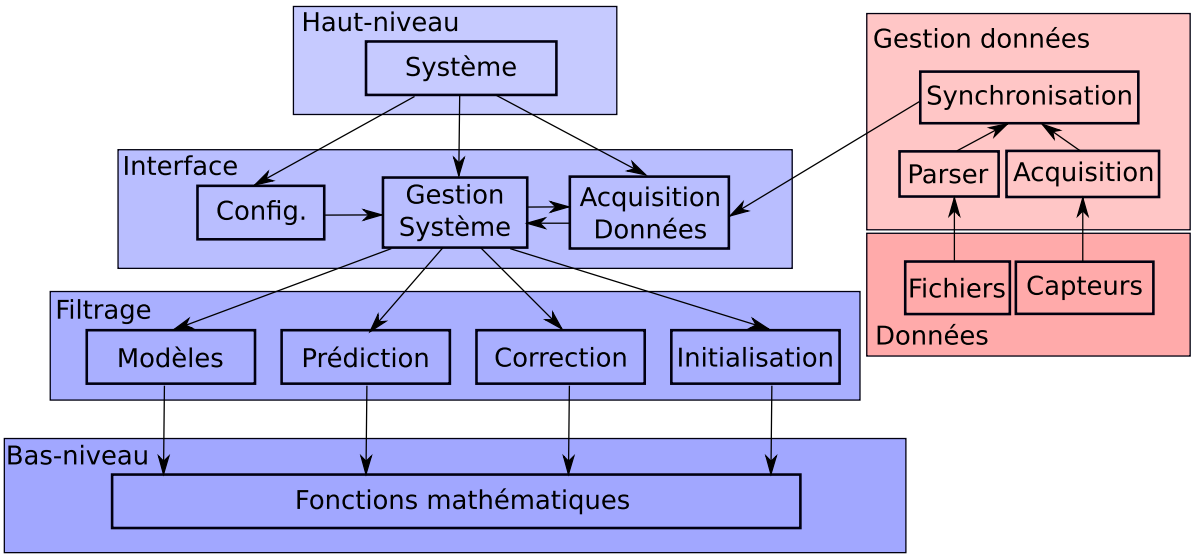

FIGURE 2.6 – Architecture de l'algorithme C-SLAM.

Même si fondamentalement l'algorithme reste identique à RT-SLAM, cette version du EKF-SLAM basée vision diffère de RT-SLAM au niveau de son implantation. Tout d'abord, elle est développée en C-ANSI à la place du C++ utilisé dans RT-SLAM. Toutes les allocations mémoires sont statiques dans la version C-SLAM (pour répondre aux contraintes DAL [9], requises au moment du développement). De plus, toutes les fonctions utilisant des bibliothèques externes comme Boost [10], Qt [11] et OpenCV [12] ont été supprimées et des fonctions de remplacement re-développées. En revanche, deux caractéristiques majeures de RT-SLAM ont été conservées :

8. http ://www.xilinx.com/content/xilinx/en/products/silicon-devices/fpga/
9. Design Assurance Level ou Niveau d'assurance du design déterminé à partir du processus d'évaluation de la sécurité et de l'analyse des risques en examinant les conditions de défaillance du système. Ces dernières sont classées en fonction de leur effet (Catastrophique, dangereux, majeur, mineur et sans effet).
10. Ensemble de bibliothèques C++ très riche. Boost offre par exemple des bibliothèques pour : les threads, les matrices, la gestion mémoire (smart pointers), la manipulation de chaîne de caractères, etc.
11. Bibliothèque dédiée au développement d'interfaces graphiques multi-plateformes.
12. Bibliothèque de référence dédiée au traitement d'images.

— La modularité car on souhaite pouvoir utiliser différentes sources de données pour C-SLAM notamment, garder la possibilité d'utiliser des données stockées sur disque dur ou directement issues des capteurs. De plus, cette modularité permet de substituer des fonctions sans avoir à modifier le reste du code. Ceci permet de tester plusieurs algorithmes et de choisir le plus adapté à nos besoins ou de déporter des tâches complètes sur d'autres unités de traitement ;
— L'architecture hiérarchique, illustrée dans la figure 2.6.

La figure 2.6 présente l'architecture du programme C-SLAM, on peut voir les parties principales : *Gestion de données* en rouge et *Système* en bleu.

La partie *Gestion de données* gère les données provenant soit directement des capteurs, soit de séquences d'images pré enregistrée et stockées en mémoire. Ce bloc regroupe, par exemple, une interface pour l'utilisation de fonctions OpenCV facilitant la relecture de séquences d'images stockées en mémoire.

La partie *Système* est divisée en quatre couches :
— Une partie *Haut-niveau* qui permet de définir le type d'entrées utilisées (capteurs, données simulées ou encore séquences stockées sur disque) et les capteurs utilisés (GPS, IMU, caméra) ;
— La section *Interfaces* effectue les appels aux fonctions principales du SLAM (prédiction, correction, etc) et charge le fichier de configuration. Ce dernier comprend le nombre d'amers considérés, la méthode de mise en correspondance des point (RANSAC ou recherche Active [13]), le type de paramétrisation (Euler, AHP, IDP) ou encore le type de descripteur/détecteur de points d'intérêt utilisé ;
— La partie *Filtrage* contient toutes opérations de mise à jour du filtre de Kalman ;
— La couche *Bas-niveau* contient les opérations mathématiques telles que les opérations vectorielles ou matricielles.

La figure 2.7 montre l'architecture niveau structurel de C-SLAM. Dans cette dernière, la boucle du SLAM débute au jeton (symbolisé en rouge) entre *landmarks initialization* et *prediction*. Dans cette dernière on retrouve les fonctions suivantes :
— *prediction* permet de prédire le vecteur d'état du robot pour l'itération courante en se basant sur le modèle de mouvement et les capteurs proprioceptifs (IMU, odométrie).
— *landmarks selection* sélectionne les amers de la carte à observer et calcule leur projection dans l'image courante.
— *correlation* permet de faire la corrélation entre les amers et leur ré-observation dans l'image courante. Cette fonction reçoit la position du point et la zone de recherche basée sur l'incertitude. Elle détermine alors l'écart entre la position prédite du point et sa position observée dans l'image.
— *correction* réalise l'étape de correction du filtre de Kalman. L'état du robot est corrigé et le modèle stochastique est mis à jour.
— *features detection* permet de faire l'acquisition des points d'intérêt dans l'image courante. L'image est découpée en portions d'images à l'aide d'une grille (tessellation cf. section 2.1.7). La fonction de détection fournit le meilleur un point d'intérêt de chaque case (celui qui à le plus grand score de détection).
— *landmarks initialization* intègre les points d'intérêt détectés comme nouveaux amers

13. ou Active Search.

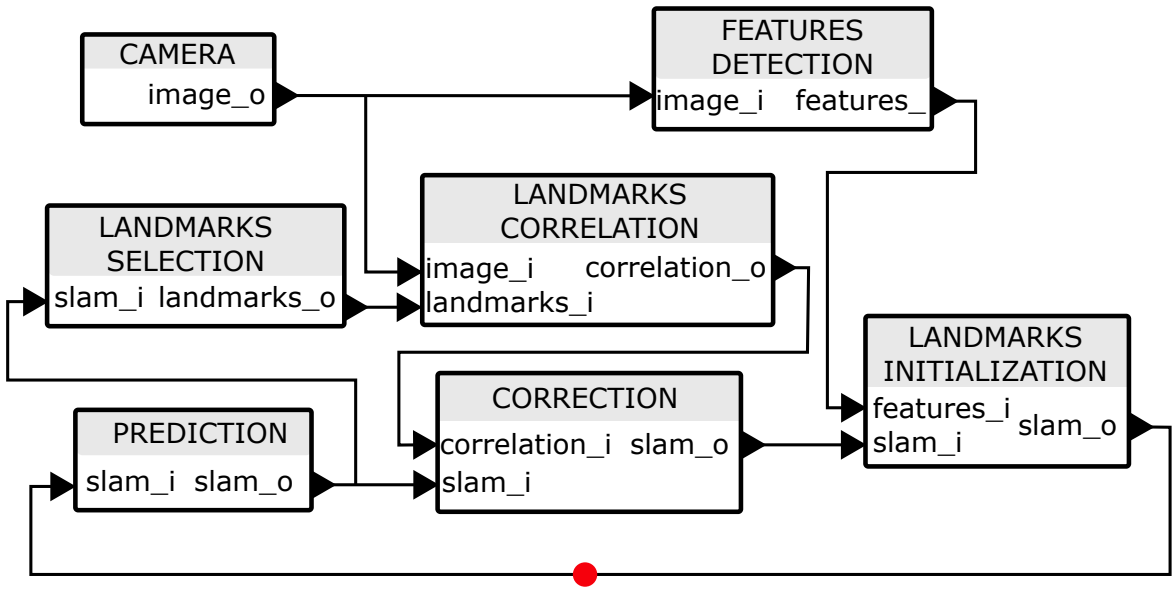

FIGURE 2.7 – Architecture structurelle de C-SLAM.

dans le vecteur d'état X. La taille de ce dernier étant limitée, aucune initialisation n'est réalisée si le nombre d'amers a atteint le maximum.

2.1.5 Détection de points d'intérêt

La détection de points d'intérêt (coins) est une étape préliminaire de la chaîne SLAM. Les points d'intérêt, dans une image, correspondent à des doubles discontinuités de la fonction d'intensité. Ce sont par exemple : les coins, les jonctions en T ou les points de fortes variations de texture. La détection de coins et de régions est devenue une méthode bien connue pour extraire des caractéristiques stables dans les images.

Un des algorithmes fréquenmment utilisés est Shi-Tomasi, [Shi and Tomasi, 1994], une amélioration du détecteur de Harris, [Harris and Stephens, 1988]. Le détecteur de Harris calcule les changements de gradients en x et en y dans une fenêtre autour du pixel traité. Un coin dans l'image correspond à un fort changement de gradient en x et en y. Un score est calculé et sa valeur correspond à la qualité de ce point d'intérêt (un angle droit obtient de meilleurs scores qu'un angle obtu). Cette méthode de détection est très efficace d'après [Mohammad Awrangjeb and Fraser, 2012], peu consommatrice en ressources de calculs. Ce détecteur est fréquement utilisé dans les SLAM en raison de son efficacité de détection, sa vitesse de calcul et sa bonne répétabilité. Le détecteur de Shi-Tomasi, [Shi and Tomasi, 1994] reprend le même principe mais change le calcul du score pour le rendre plus efficace.

FAST (Features from Accelerated Segment Test), [Rosten and Drummond, 2005], plus récent, est l'un des algorithmes les plus rapides, les plus simples et, d'après [Mohammad Awrangjeb and Fraser, 2012] aussi efficace que Shi-Tomasi ou Harris. Il est basé sur la comparaison des pixels d'un cercle centré sur le pixel traité. Sa rapidité vient du fait qu'il ne met en oeuvre que des opérations simples (des comparaisons et des additions).

Harris et FAST sont deux détecteurs rapides et efficaces mais qui sont sensibles à la rotation,

	Rapidité	Type de détecteur	Invariabilité au changement	Implantation HW
Harris	+ +	coin	/	+ +
FAST	+ + +	coin	/	+ + +
BRISK	+ +	coin	de rotation & d'échelle	-
ORB	+ +	coin	de rotation & d'échelle	-
SIFT	- -	région	de rotation, d'échelle & de netteté	- - -
SURF	-	région	de rotation, d'échelle & de netteté	- -

TABLE 2.1 – Récapitulatif des avantages et inconvénients des détecteurs de points d'intérêt. Avec en bleu les détecteurs de coins (ou points d'interêt) et en rouge les détecteurs de régions.

à l'échelle et au changement de netteté [14]. Le détecteur BRISK (Binary Robust Invariant Scalable Keypoints, [Leutenegger et al., 2011]), basé sur FAST, résout le problème de sensiblité à l'échelle par un sous-échantillonnage de l'image. ORB (Oriented FAST and Rotated BRIEF), [Rublee et al., 2011], est une alternative à SIFT (Scale-Invariant Feature Transform), [Lowe, 1999]. Il combine la méthode BRISK avec un descripteur de point d'intérêt BRIEF (cf. section 2.1.6) afin de résoudre le problème de sensiblité à la rotation et à l'échelle. Cependant, le détecteur SIFT [Lowe, 1999] dont la philosophie est très proche du détecteur SURF (Speeded-Up Robust Features - [Bay et al., 2006]) est invariant à la rotation, à l'échelle mais aussi au changement de netteté.

Le tableau 2.1, basé sur [Olaizola Goenaga, 2014], récapitule les détecteurs précédemment cités en fonction de leur rapidité d'exécution en logiciel, leur invariance et leur complexité d'implantation matérielle.Les quatre détecteurs BRISK, ORB, SIFT et SURF requièrent plus de ressources matérielles pour être implantés que les détecteurs élémentaires FAST et Harris. En effet, ces quatre détecteurs sont, tout d'abord, multi-échelle ce qui implique une duplication du processeur de détection de coin primaire - de type FAST par exemple - à plusieurs échelles dans l'image. De plus, ils intègrent tous le calcul d'un descripteur de points d'intérêt (cf. section 2.1.6) ce qui complexifie l'algorithme. En sachant que notre EKF-SLAM est basé sur le suivi d'amers ré-observé grâce à la Recherche Active (cf. 2.1.7), le détecteur de points d'intérêt n'a pas besoin d'être invariant à la rotation, l'échelle ou encore au changement de netteté. Harris et FAST font appel à des opérations peu coûteuses en ressources de calculs. Ils sont donc adaptés à notre application embarquée.

L'extraction des points d'intérêt consiste à traiter tous les pixels d'une image. Compte tenu de la quantité de données à traiter, il est intéressant d'implanter cette détection sur une architecture matérielle afin accélérer cette tâche. De cette façon, la fonction matérielle traite des données haut débit (les pixels) en entrée et fournit des données bas débit de plus haut niveau d'abstraction (les points d'intérêt de l'image). L'utilisation d'un FPGA comme unité de traitement hautement parallélisable permet d'extraire ces caractéristiques dans des images hautes résolutions et avec des performances de temps de traitement élevées et de latence faible. Dans [Birem and Berry, 2012], Birem et al. démontrent que malgré une optimisation intéressante, le détecteur de Harris, en raison de sa complexité de calcul, utilise beaucoup de

14. La diminution de la netteté de l'image engendrant un estompage les contours.

ressources matérielles et les délais de traitement des performances atteintes sont insuffisantes pour notre application SLAM.

2.1.6 Descripteur de points d'intérêt

Un point d'intérêt constitue une information assez pauvre. Pour rendre les appariements plus robustes, on enrichit ces points d'intérêt d'un descripteur décrivant leur voisinage.

Il existe de nombreux descripteurs et beaucoup d'entre eux sont devenus des références. Ces descripteurs peuvent être classés dans deux catégories, les descripteurs binaires et ceux basés sur les gradients.

Parmi les descripteurs binaires BRIEF, BRISK et ORB sont très utilisés. Ces descripteurs sont connus pour leur rapidité de calcul mais souffrent souvent d'un manque de robustesse. BRIEF, [Calonder et al., 2010], est basé sur des comparaisons de pixels d'une fenêtre autour du pixel à décrire. La taille de cette fenêtre peut être paramétrée (elle est en général de 9x9 ou 15x15). Le nombre de comparaisons minimum est situé entre 128 et 512 pour avoir une description du pixel suffisamment riche. On parle alors de BRIEF128, BRIEF256, etc. Ce descripteur est invariant au changement de netteté mais est très sensible à la rotation et à l'échelle. BRISK, présenté en 2011 dans [Leutenegger et al., 2011], est basé sur le même principe de comparaison que BRIEF. En revanche, il utilise les orientations de gradient dans les comparaisons de paires de pixels ce qui lui permet d'être plus robuste à la rotation. Ces comparaisons de pixels sont effectuées sur une fenêtre plus large que celle utilisée par BRIEF et la répartition des comparaisons est organisée en 2 catégories (comparaisons de longues distances et courtes distances). Cette répartition est, contrairement à BRIEF, non aléatoire, les comparaisons sont spatialement réparties sur des cercles concentriques au pixel décrit. Ces caractéristiques permettent de rendre l'algorithme plus robuste au changement d'échelle. ORB [Rublee et al., 2011], est la combinaison d'un détecteur FAST et d'un descripteur hybride utilisant les principes de BRIEF et de BRISK. Il hérite des qualités de BRISK (rapidité et invariabilité au changement de netteté, à la rotation et au changement d'échelle) mais perd en répétabilité.

La deuxième catégorie, les descripteurs basés sur les gradients, présente une très bonne répétabilité. SURF [Bay et al., 2006], recherche, dans un premier temps, l'orientation de l'angle du coin à décrire (ce qui lui permet d'être invariant à la rotation). Dans un deuxième temps, la fenêtre de description est divisée en une grille 4x4. Chaque case de cette grille est sous-divisée en une grille 5x5. Ces zones 5x5 sont alors utilisées pour calculer les orientations de gradient. Ces calculs sont faits à différentes échelles pour rendre le descripteur robuste au changement d'échelle. On comprend alors pourquoi cette deuxième catégorie est plus performante et pourquoi ces descripteurs nécessitent beaucoup plus de calculs. SIFT [Lowe, 2004] est, parmi la liste de détecteur que nous dressons, celui qui consomme le plus ressources de calculs. Il est basé sur des magnitudes de gradients calculées et réparties sur une fenêtre autour du point d'intérêt. Ce descripteur est l'un des plus performants en termes de répétabilité.

Le tableau 2.2 récapitule les avantages et inconvénients des descripteurs précédemment évoqués.

Dans une optique d'implémentation sur FPGA ou plus généralement dans une architecture embarquée, les descripteurs basés gradiants sont donc très difficilement implantable car trop coûteuse en termes de ressources matérielles. Du point de vue de l'implémentation logicielle et matérielle, BRIEF est le plus simple tout en restant efficace car il n'utilise que des comparaisons

	Rapidité	Répétabilité	Type de descripteur	Invariabilité au changement	Implantation HW
BRIEF	+ + +	+	Binaire	de netteté	+ + +
BRISK	+ +	+	Binaire	de netteté, de rotation & d'échelle	+ +
ORB	+ +	-	Binaire	de netteté, de rotation & d'échelle	-
SIFT	- -	+ + +	Basé gradient	de netteté, de contraste, de rotation & d'échelle	- - -
SURF	-	+ +	Basé gradient	de netteté, de contraste, de rotation & d'échelle	- -

TABLE 2.2 – Récapitulatif des avantages et inconvénients des descripteurs de points d'intérêt. Avec en bleu les descripteurs binaires et en rouge, les descripteurs basés sur les gradients.

de pixels (opération très facilement implantable sur FPGA). D'après [Calonder et al., 2012], le descripteur tolère une rotation plane de 15° ce qui est bien suffisant dans notre cas car la caméra est fixée sur un robot ou une voiture, elle ne peut donc pas faire de rotation importante autour de son axe optique. De plus, nous allons voir dans la partie suivante (section 2.1.7) que la mise en correspondance entre deux points d'intérêt décrits par BRIEF est relativement peu coûteuse en calculs.

2.1.7 Stratégie de sélection des amers à corriger et recherche active (ou Active Search)

Sélection des amers Dans les applications RT-SLAM et C-SLAM, on cherche à corriger le filtre de Kalman en comparant la prédiction des positions des amers avec leurs ré-observations. Lors de l'étape de *landmarks selection* (cf. figure 2.7), on sélectionne les amers de la cartes à observer et on calcule leur projection dans l'image courante. En sachant que l'étape de correction représente la plus grosse charge calculatoire de la boucle SLAM et que son temps de traitement est directement lié au nombre d'amers suivis, il est nécessaire de limiter le nombre de ré-observation (minimum 5). Pour cela, dans RT-SLAM et C-SLAM, on corrige en priorité les amers qui ont les incertitudes les plus grandes car ce sont ceux pour lesquels la ré-observation apporte le plus de correction au système (la ré-observation d'un amer ayant une incertitude minime n'apporte quasiment aucune correction). Dans RT-SLAM et C-SLAM la sélection des amers à corriger est donc guidée par la matrice de gain de Kalman (cf. équation 2.12) contenant l'incertitude de la position des amers dans la carte. Cette stratégie de sélection d'amers présente l'avantage de corriger de manière optimum les positions (du robot et des amers). Cependant, si le filtre de Kalman n'est plus consistant (les incertitudes des positions de tous les amers sont grandes), les amers ne seront jamais retrouvés et le SLAM aura tendance à casser et à

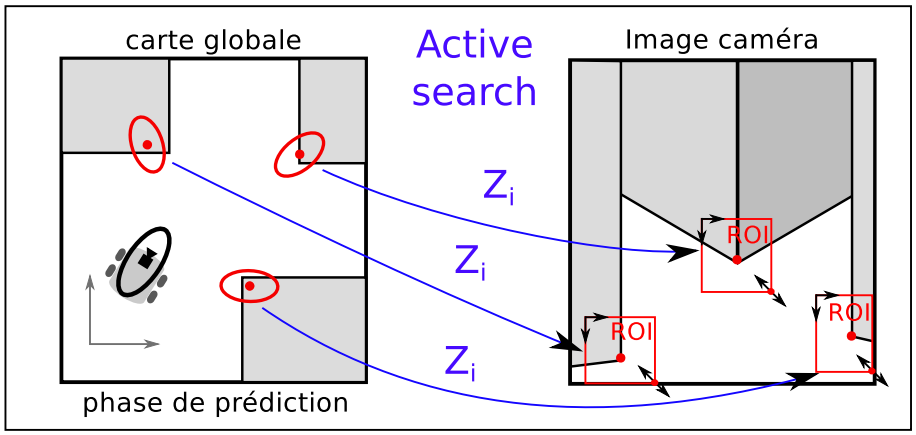

FIGURE 2.8 – La fonction active search définit la position et la dimension des ROI en fonction des données de prédiction.

dériver (l'erreur augmente à chaque correction du filtre). Dans cette situation, il est préférable de réinitialiser le SLAM au lieu de tenter de fournir des mesures à tout prix à un système qui n'est déjà plus cohérent.

Une fois les amers sélectionnés, ils sont reprojetés dans l'image (cf. figure 2.8) par la fonction de recherche active. Aussi, elle déduit de l'incertitude de la position des amers, des zones de recherche dans l'image (ou ROI [15]). La figure 2.9 présente le principe de cette recherche active. Dans cette dernière, les connaissances courantes permettent de limiter les zones de recherche des points d'intérêt. Ainsi, l'étape de prédiction fournit la position attendue de chaque amer et la matrice de covariance de l'innovation Z_i permet de calculer l'incertitude de cette position (qui prendra la forme d'ellipses dans la carte car l'incertitude sur la profondeur est plus élevée que sur les autres dimensions, cf. figure 2.8). Les ROIs obtenues sont rectangulaires pour simplifier la recherche.

Mise en correspondance Une fois les ROIs délimitées, une étape d'exploration basée corrélation permet de mesurer la position réelle du point d'intérêt dans l'image (cf. figure 2.9). C'est l'étape de mise en correspondance dans laquelle on cherche à corréler (voir section "corrélation" suivante) le descripteur associé à l'amer que l'on souhaite re-observer avec les descripteurs des pixels de l'image courante. Une fois les points d'intérêt appariés, l'étape de filtrage permet de mettre à jour l'état du système en calculant l'innovation (cf. section 2.1.2).

Cette Recherche Active permet de minimiser l'erreur de mise en correspondance entre deux points en limitant la zone de recherche. Elle permet ainsi de limiter les faux appariements et d'optimiser le temps de traitement.

Corrélation La mise en correspondance de points d'intérêt se fait grâce à la corrélation de leurs descripteurs. Il existe différentes méthodes de corrélation comme la comparaison de modèles (*template matching*). Elle consiste à calculer, à chaque position de l'image sous examination, une fonction qui mesure le degré de similitude entre un modèle (le patch décrivant le point d'intérêt que l'on souhaite rechercher) et une zone de l'image. Dans cette famille de cor-

15. Region Of Interest

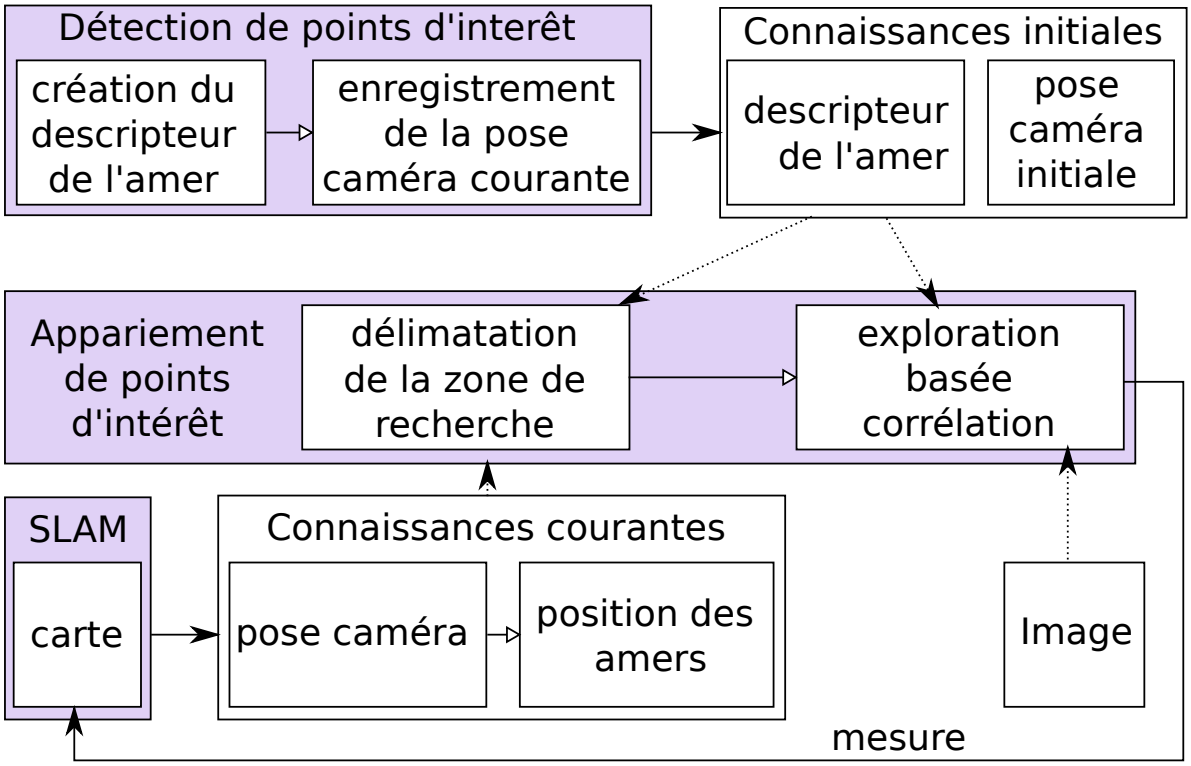

FIGURE 2.9 – Recherche Active. Les connaissances initales contenues dans la carte du SLAM et les connaissances courantes issues de la détection de points d'intérêt sont exploitées pour le processus d'appariement.

rélation par comparaison de modèles, il existe différents critères de corrélation comme ZNCC (Zero-mean Normalized Cross Correlation) utilisé dans RT-SLAM et C-SLAM. L'implantation de [Di Stefano et al., 2005] permet d'optimiser la recherche en interrompant le calcul si le score de corrélation dépasse un seuil. Ce type d'algorithme reste lourd à traiter puisqu'il exécute des calculs bas niveau sur les pixels. Afin de limiter les calculs dans le but d'une implantation sur architecture embarquée, il est préférable d'utiliser des informations de plus haut niveau. Pour cela, les descripteurs binaires comme BRIEF ou BRISK permettent une correlation simplifée. Une comparaison de ces descripteurs binaires permet de calculer simplement un score de corrélation. On peut calculer par exemple la distance de Hamming [Hamming, 1950]. Cette distance se calcule en effectuant une opération logique XOR entre les deux descripteurs binaires comparés et la distance est donnée par le comptage des bit à $'1'$ du vecteur résultant. Ces opérations sont rapidement traitées par les processeurs modernes (intégrant les instructions SIMD).

Dans [Calonder et al., 2010], les auteurs prouvent que la mise en correspondance de descripteurs BRIEF est la plus rapide de l'état de l'art actuel du domaine. Ils démontrent aussi que les résultats obtenus sont très bons dans le cas où l'invariabilité à la rotation n'est pas une condition du système. D'après [Calonder et al., 2010], la distance de Hamming est très rapidement exécutée sur des systèmes embarqués. Elle peut, de plus, être facilement implantée en matériel. En effet, l'opération XOR et celle de comptage de $'1'$ peuvent être optimisées sur FPGA comme nous le verrons au paragraphe 5.4.1.

Tesselation Afin de rendre notre système le plus robuste possible il faut favoriser l'observabilité de l'état en répartissant le plus possible les amers dans l'image. Dans le cas d'une application avec la caméra pointant sur le sol et l'horizon, les points observés au premier plan (sur le sol) donnent une bonne information de vitesse et les points lointains (à l'horizon) renseignent sur la direction du robot.

Dans RT-SLAM et C-SLAM, afin de limiter la recherche de points d'intérêt, on définit une grille (16*12 cases dans notre cas). Au moment de l'étape d'initialisation des points, une case (dans laquelle aucun amer suivi n'est présent, cf. cases vertes dans la figure 2.10) est sélectionnée au hasard afin d'y détecter un point d'intérêt (celui qui a le plus haut score calculé par l'algorithme de détection de points d'intérêt). La figure 2.10 propose un exemple de tesselation d'une image (grille 4x5). Les cases rouges sont des régions de l'image où un amer est déjà suivi. Les cases vertes sont éligibles pour y exécuter une détection de points d'intérêt avant d'injecter le meilleur dans le SLAM.

Cette méthode de répartition des points d'intérêt dans l'image permet de limiter la détection de points d'intérêt à des portions de l'image. Ceci permet de réduire les calculs sanq affecter le fonctionnement du SLAM puisque la tesselation n'intervient qu'au moment l'initialisation des amers. Par la suite, ils sont corrélés par une recherche active.

Dans cette partie 2.1, nous avons fait un état de l'art des différents SLAM (par optimisation et par filtrage). Nous avons ensuite détaillé le principe du SLAM par filtrage EKF et nous avons exposé les deux chaînes EKF-SLAM vision embarquées opérationnelles au LAAS : RT-SLAM et C-SLAM. Puis, nous avons fait l'état de l'art des détecteurs et des descripteurs de points d'intérêt. Enfin, nous avons présenté la stratégie de sélection des amers et leur recherche active. Nous allons, dans la section 2.2, présenter les différentes architectures disponibles pour accueillir notre chaîne EKF-SLAM embarquée.

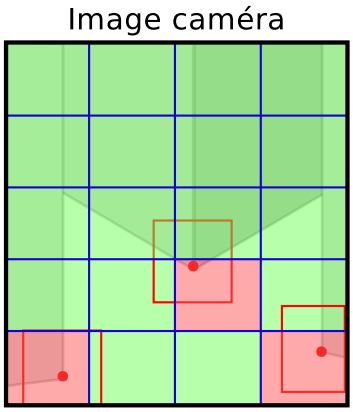

FIGURE 2.10 – Exemple de tesselation de l'image permettant la répartition des amers dans l'image. Les cases rouges sont occupées, un amer est déjà observé dans cette région et les cases vertes sont disponibles pour l'initialisation de nouveaux points.

2.2 SoC cibles et développement FPGA

Nous allons, dans cette section, présenter les SoC disponibles sur le marché dans le paragraphe 2.2.1. Nous terminerons en détaillant la technologie FPGA et les particularités du développement FPGA en section 2.2.2.

2.2.1 Les SoC cibles

Dans les systèmes numériques, il existe différents types d'unités de traitement. D'une part, les architectures nécessitant une programmation logicielle et d'autre part les solutions dédiées (figure 2.11). Dans la première famille, on retrouve les microprocesseurs (ou CPU [16]), les processeurs graphiques (ou GPU [17]) ou encore les microprocesseurs spécialisés en traitement du signal (ou DSP [18]).

Dans la catégorie des architectures développé grâce à une description matérielle, on retrouve entre autres les SoC [19] dédiés (ou ASIC [20]) ainsi que les FPGA [21].

Les CPUs Les microproceseurs ou CPUs sont des architectures **génériques** permettant tous types d'opérations. Grâce à leur **flexibilité**, ils sont utilisés dans tous les domaines d'application. En revanche, cet avantage est aussi un inconvénient puisqu'ils sont **non optimisés** pour une application précise. De par leur architecture orientée **séquentielle**, ils deviennent très vite peu efficaces par rapport à des architectures qui parallélisent massivement les opérations comme le GPU.

Les GPUs ou GPGPU Le GPU, à l'origine utilisé pour du rendu graphique, est maintenant utilisé dans de multiples domaines (GPGPU General Purpose GPU). Cette architecture

16. Central Processing Unit
17. Graphics Processing Unit
18. Digital Signal Processor
19. System-On-Chip
20. Application Specific Integrated Circuit
21. Field-Programmable Gate Array

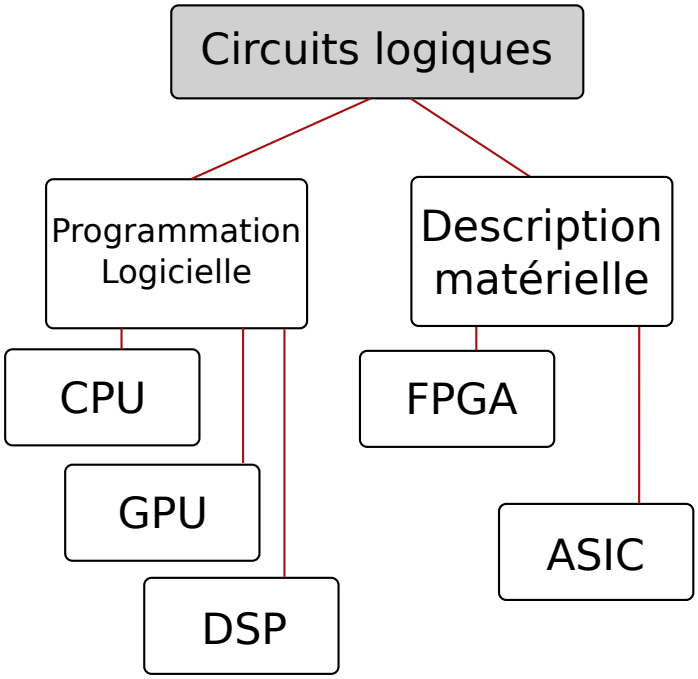

FIGURE 2.11 – Classification des SoC.

est basée sur plusieurs centaines de coeurs. De par sa configuration, elle est orientée vers le traitement de **longues séquences** de calculs en **parallèles**. Chaque coeur est alimenté par des données directement lues en mémoire DRAM. La latence du transfert mémoire est compensée par la multiplication des threads, pendant l'acheminement des données, d'autres calculs sont exécutés. C'est pourquoi, pour profiter de leur pleine puissance, les GPUs nécessitent de longues tâches de calculs parallélisables. La figure 2.12 schématise les différences entre les architectures CPU et GPU. On remarque les différences de superficies des mémoires cache. Il y aussi une différence de taille des ALUs et de leurs contrôles car les ALUs des GPU sont seulement capables d'effectuer des opérations basiques (c'est leur nombre qui compense leurs jeux d'instructions réduits).

Dans [Shi et al., 2010] les auteurs démontrent que l'utilisation de la programmation CUDA (Compute Unified Device Architecture) sur GPU dans le cadre des applications tomodensitométriques (scanner à rayons-X) pouvait accroître les performances d'un facteur 80 par rapport à une éxécution sur CPU.

Les DSPs Proches de l'architecture des CPUs, les DSPs sont des microprocesseurs dotés de **jeux d'instructions dédiés** aux applications de **traitement du signal**. L'instruction MAC (Multiply and ACcumulate) souvent implémentée dans les DSP permet par exemple la multiplication de deux opérandes et l'accumulation dans un troisième en un seul cycle d'horloge système. Les implémentations de filtres FIR ou de calcul de FFT utilisent massivement cette opération. De plus, les DPS intègrent souvent du parallélisme de données. Les instructions SIMD (Single Instruction Multiple Data) permettent par exemple de traiter plusieurs données avec une seule instruction. Certains DSP disposent d'instructions VLIW (Very Long Instruction Word) qui permettent d'adresser plusieurs ALU avec une seule instruction.

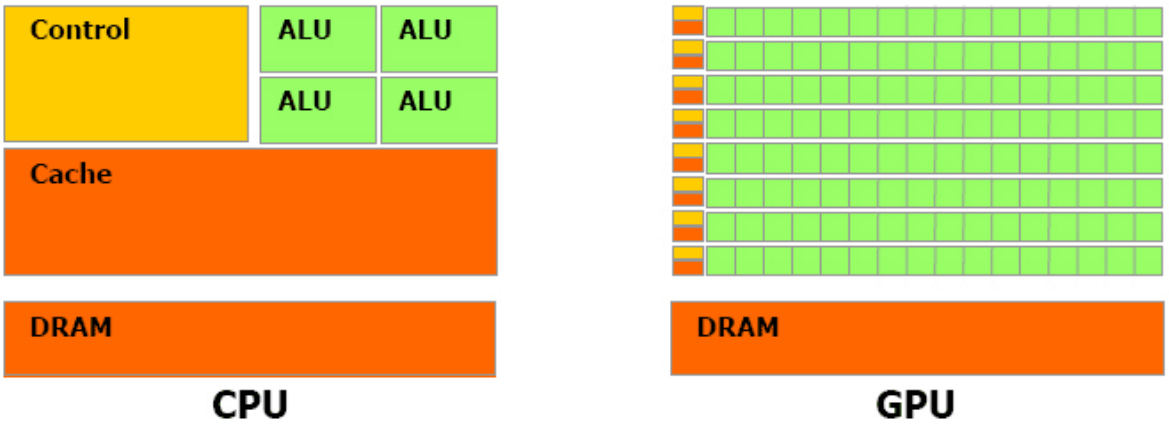

FIGURE 2.12 – Différences d'architecture entre CPU et GPU - Source : NVIDIA®

Les FPGAs Les **FPGA**s ont été, à l'origine, créés pour aider au développement des ASICs. Les principaux fabriquants sont Xilinx®, Altera®. Un FPGA est en fait un amas structuré d'opérateurs logiques élémentaires (LUT, MUX) dont le concepteur va programmer l'interconnexion pour réaliser un ensemble de fonctions complexe qui s'exécutent en parallèle. Cette technologie sera présentée plus en détails dans la section suivante (section 2.2.2).

Dans [Asano et al., 2009] les auteurs présente une comparaison de performances entre CPU, GPU et FPGA pour deux algorithmes de traitement d'images (en vision stéréo et des calculs de filtres à deux dimensions).

Vers des architectures hybrides Actuellement et, ce dans tous les domaines, la **quantité de données** à traiter **augmente** exponentiellement. Par exemple, dans la vision, les capteurs délivrent de plus en plus de pixels à des fréquences de plus en plus élevées. Dans l'automobile, le nombre de capteurs embarqués est lui aussi de plus en plus important. Dans la téléphonie mobile, la société CISCO® estime que l'augmentation des données qui transitent suit une loi exponentielle. La figure 2.13 montre cette évolution en exaoctets (10^{18} octets) de données échangées par mois.

Ces augmentations entraînent un besoin d'améliorer la performance des unités de traitement. La désormais très célèbre loi de Moore de 1905, jusque là représentative de l'évolution des performances est désormais remise en question. En effet, les finesses de gravure des transistors actuels ont atteint un plafond. Il est désormais nécessaire de multiplier le nombre de coeurs pour atteindre des performances de plus en plus élevées au lieu d'augmenter la fréquence de traitement (liée à la finesse de gravure). La figure 2.14 montre la croissance asymptotique ("frequency wall") des performances en consommation et fréquence des microprocesseurs.

Les limites de la loi de Moore sont encore plus flagrantes dans les architectures embarquées, où la consommation d'énergie et l'encombrement sont des contraintes fortes. On peut, en effet, difficilement multiplier les coeurs CPU sans consommer de l'énergie.

Afin de pallier à cette limitation technologique, en **embarqué**, de nouvelles architectures voient actuellement le jour, combinant les différents types d'unités de traitement pour créer des **architectures hybrides/hétérogènes** afin de tirer partie des spécificités de chacune des unités. Il existe deux approches, d'une part, des accélerateurs matériels dédiés couplés à un

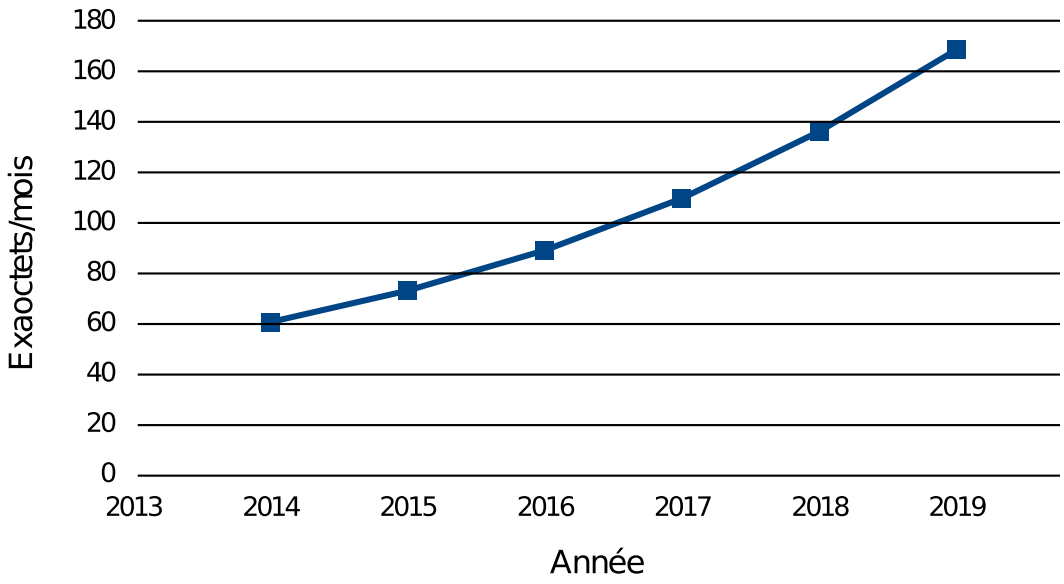

FIGURE 2.13 – Nombre de données echangées sur le réseau mobile en exabyte par mois - Source : CISCO®

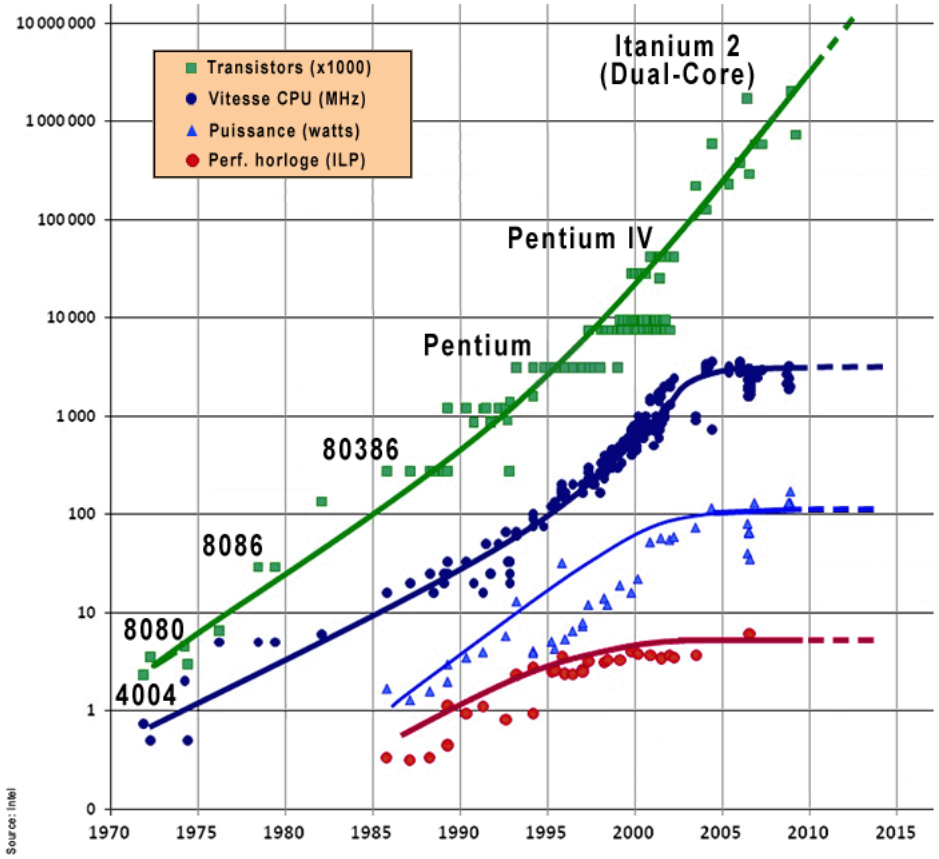

FIGURE 2.14 – Loi de moore - Source : Intel®

CPU et d'autre part, les architectures basées processeurs. Quelque soit le type d'architecture hybride, un CPU est quasiment toujours utilisé afin de réaliser les tâches principales. A ce CPU est couplé un ou plusieurs autres coeurs de traitement afin d'accélérer certains calculs.

Dans la famille des couplages entre un CPU et une architecture non-basée processeur on retrouve le CPU-ASIC. L'encodeur/décodeur mp3 en est un bon exemple et est maintenant devenu un standard pour nombre d'applications. Depuis maintenant un certain nombre d'années, l'encodage et le décodage de la musique ne sont plus assurés par le CPU mais par une fonction matérielle dédiée implantée dans les SoC CPU. On retrouve le même type d'accélération avec la vidéo dans les smartphones actuels. De la même manière, beaucoup de SoC dédiés au traitement de la vision voient le jour actuellement (Movidius® : myriad 2, Centeye® : Stonyman). A l'instar des encodeurs/décodeurs vidéos et audios ces unités de traitements sont constituées de processeurs spécialisés (CPU équipés d'instructions optimisées et accélérateurs matériels dédiés) aux tâches de traitement pour la vision (très coûteuses en temps de traitement sur une architecture CPU standard). Ces architectures dédiées sont toujours développées d'abord sur FPGA afin de les valider. Il existe aussi des SoC avec un couplage CPU-FPGA. Cette association est souvent utilisée dans des applications à petite série qui ne justifient pas un investissement dans le développement d'un ASIC. La société Xilinx® propose ce type d'association entre un FPGA et un CPU sur le même SoC. Les CPU-FPGA dispose d'une architecture de type séquentielle (le CPU) et une architecture configurable dédiée afin de décharger ce microprocesseur pour des traitements spécifiques très demandeurs en ressources de calcul. Xilinx® a dévoilé en 2011 leur premier AP-SoC (All Programmable System on Chip), le Zynq-7000. Altera®, le principal concurrent, les nomme Altera® SoC, disponibles depuis 2012.

La diversité des unités de traitements disponibles sur le marché reflète le fait qu'il n'existe, à ce jour, aucune solution universelle s'appliquant à tout type d'application et répondant à toutes les contraintes. Les solutions précédemment décrites ont toutes leurs avantages et inconvénients. En effet, les architectures de type **CPU** ont en commun une facilité de programmation et de débogage ainsi qu'une flexibilité de calcul. En revanche, cette flexibilité résulte en un rapport efficacité numérique (GFlops/Watt) faible. Les architectures **GPU**s ont une performance plus élevée dans le cadre d'une application requérant des calculs massivement parallélisables. En revanche, elles nécessitent un effort de développement plus important que les CPUs et le débogage des programmes est moins évident. Certains types de calculs ne sont de plus pas adaptés à l'implémentation GPU et la performance des calculs peut être rapidement impactée par les transferts de données (CPU->GPU->CPU). Les **FPGA** sont extrêmement performants sur des problématiques spécifiques et ont un bon rapport GFlops/Watt. En revanche, ils nécessitent un effort de développement très important car ils sont difficiles à configurer et à déboger. Dans [Yang et al., 2008], une comparaison de multiples algorithmes de traitements d'images implantés exécutés sur CPU et sur CPU+GPU montre que l'association CPU-GPU arrive à des performances de temps de traitement 100 fois supérieur aux exécutions des mêmes algorithmes sur CPU seul. Le tableau 2.3 résume et met en correspondance les avantages et inconvénients de chacune des architectures classiques CPU, GPU et FPGA.

La figure 2.15 schématise les performances de ces architectures classiques en GFlops/Watt en fonction de l'efficacité et de la flexibilité.

La figure 2.16 met en relation l'optimisation, la complexité du design en fonction du temps de développement. Pour chaque type d'architecture, il faut maximiser le rapport complexité/optimisation des traitements en fonction du temps de développement.

	consommation	performances (GFlops/Watt)	temps de développement
CPU	+	- -	+ +
GPU	-	+	+
FPGA	+ +	+ +	- - -

	Flexibilité	cadence	latence dans une chaîne de calcul
CPU	+ +	- -	- -
GPU	-	+	+ (si longues chaînes de calcul)
FPGA	- -	+ +	+ +

TABLE 2.3 – Récapitulatif des avantages et inconvénients des CPUs, GPUs, FPGAs

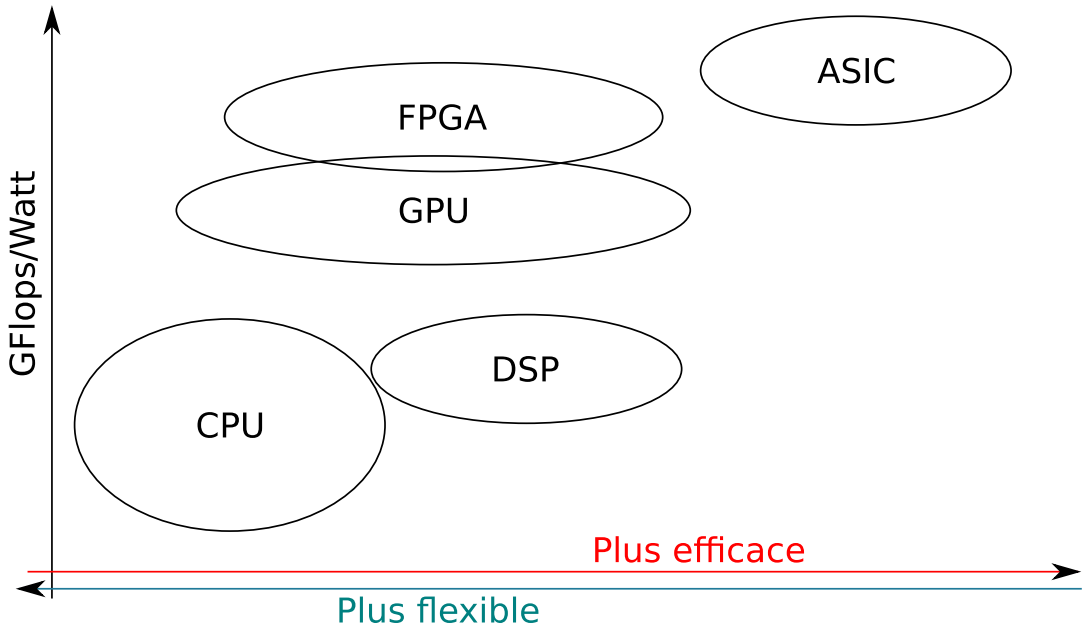

FIGURE 2.15 – Performances en fonction de la flexibilité des CPUs, FPGAs, GPUs, DSPs et ASICs

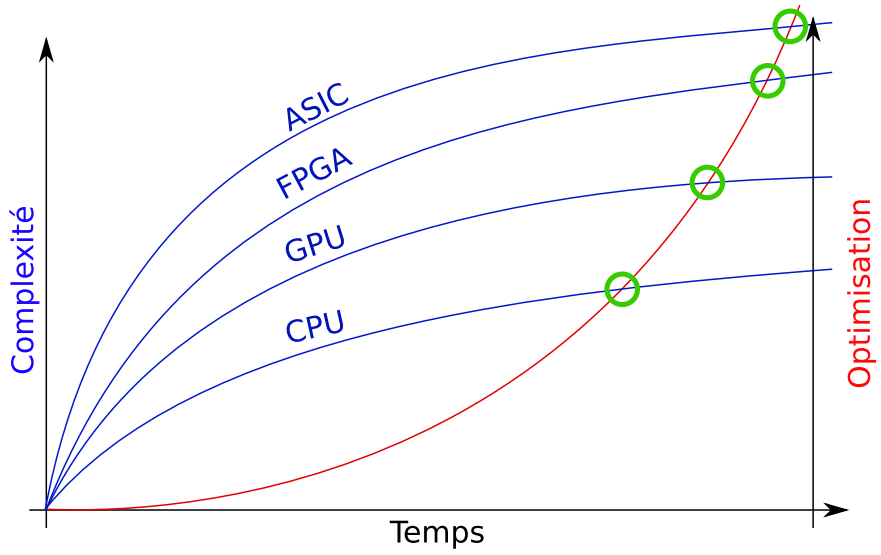

FIGURE 2.16 – Schématisation du contenu d'un DSP d'un FPGA. Source : Xilinx®

Comme précédemment expliqué, ces architectures de traitements ont été couplées au sein d'un même SoC. Ces nouvelles plateformes hétérogènes regroupent des CPU avec des GPU, des FPGA avec des CPU, plusieurs CPU ou encore des ASIC avec des CPU. Le but étant toujours d'utiliser un CPU pour les tâches générales qui ne nécessitent pas de gros calculs. A ce CPU est couplé un ou plusieurs autres coeurs de traitements afin d'accélérer certains calculs. Le tableau 2.4 montre les avantages et inconvénients de ces architectures hybrides.

On notera que, dans ce tableau 2.4, les deux architectures intéressantes pour notre problématique sont le GPGPU et le FPGA-CPU. Leur flexibilité permet de faire un travail de recherche et développement. Ces deux types de plates-formes nécessitent de maîtriser certaines informations. Les GPGPU partagent la mémoire embarquée entre le CPU et le GPU, il faut donc avoir une bonne connaissance et une maîtrise du contenu de cette mémoire et de l'utilisation de la bande passante. Les SoC CPU-FPGA nécessitent une parfaite connaissance du flux de données et la maitrise de la synchronisation entre les deux domaines.

Nous avons décidé de ne pas utiliser de GPGPU afin de pouvoir proposer une architecture la plus optimale possible au vu de nos contraintes. car ils ne valident pas nos objectifs de smartcaméra (les performances du GPU sont insuffisantes pour notre application SLAM). Nos contraintes en dimension et consommation sont trop strictes. Enfin, au vu du type d'algorithme que nous souhaitons implanter, le SLAM EKF est, par nature, très linéaire. Le GPU est donc, dans ce cas, moins efficace qu'une architecture dédiée (type FPGA) puisqu'il a besoin d'être chargé en calcul parallèle afin de rendre la latence mémoire nulle. Nous avons donc choisi, en fonction de nos contraintes, d'utiliser une plateforme CPU-FPGA. Au début du développment, en 2012, il n'existait qu'un seul SoC CPU-FPGA, le Zynq 7000 de Xilinx®. Il rassemble un processeur ARM® dual core cortex A9 et de la logique FPGA de la génération Artix® 7.

	consommation	performances (GFlops/Watt)	temps de développement
CPU-ASIC	+ + +	+ + +	- - -
FPGA-CPU	+ +	+ +	- -
GPGPU	+	+	+
multi CPU	-	- -	+ +

	flexibilité	cadence	latence dans une chaine de calcul
CPU-ASIC	- - -	+ +	+ +
FPGA-CPU	-	+ +	+ +
GPGPU	+	+	+ (si longues chaînes de calcul)
multi CPU	+ +	- -	- -

TABLE 2.4 – Récapitulatif des avantages et inconvénients des architectures hybrides GPGPU et CPU-FPGA

2.2.2 Développement FPGA

2.2.2.1 La technologie

Il existe deux technologies de FPGA, les FPGA basés sur de la mémoire SRAM et ceux basés sur de la mémoire flash.

Parmi les fabriquants de FPGA en mémoire flash, on trouve Actel (actuel leader du marché) et Quick Logic. Leur technologie est très utilisée dans le secteur spatial car les mémoires flash sont moins sensibles aux bombardements magnétiques. En revanche, les FPGA basés sur de la mémoire SRAM représentent la plus grande part de marché (90%). Xilinx® et Altera®, les deux plus gros vendeurs détiennent les 2/3 du marché actuel.

Les circuits FPGA (Field Programmable Gate Array) sont constitués d'une matrice de blocs logiques programmables (cf. figure 2.17, logic blocks), des blocs mémoires (BRAM [22]), de DSP (Digital Signal Processor), de PLL (Phase Locked Loop) entourés de blocs d'entrée/sortie programmables (I/O blocks). L'ensemble est relié par un réseau d'inter-connexions programmables (programmable interconnect).

Les blocs logiques sont principalement composés de

— LUT (Look-up table -> table de vérité) qui permettent de synthétiser des fonctions combinatoires élémentaires
— Multiplexeurs qui permettent la sélection
— DFF (D Flip-Flop -> bascule D) qui permettent la mémorisation de bits et la composition de fonctions séquentielles

A ces modules configurables s'ajoutent des blocs de mémoires :

— Les Block RAM (ou BRAM) : mémoires atomiques de taille fixe (ou configurable)
— Les mémoires distribuées : Mémoires de un bit pouvant étre associées pour former une mémoire de taille plus importante

Ces éléments diffèrent (en taille et nombre d'entrée/sortie) selon les constructeurs et les types de FPGA.

22. Bloc Random Acces Memory

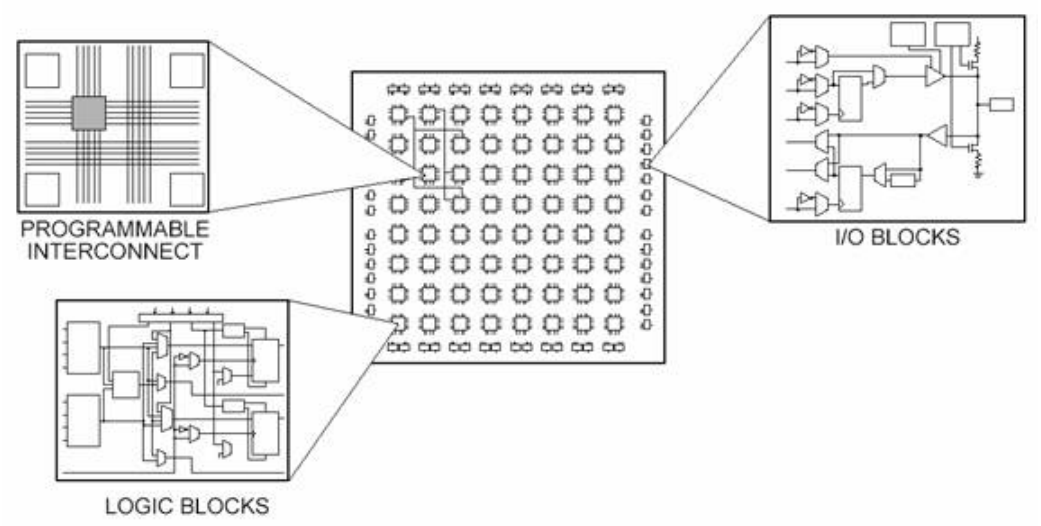

FIGURE 2.17 – Structure d'un FPGA.

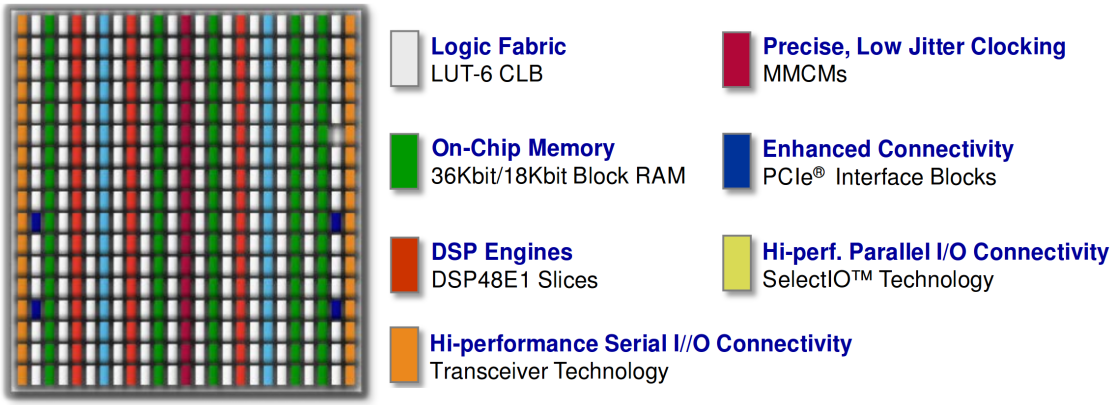

FIGURE 2.18 – Architecture en colonne d'un FPGA Virtex® série 7 Xilinx®. Source : Xilinx®

La figure 2.18, illustre l'architecture en colonne des FPGAs avec notamment les blocs mémoires (*BRAM*, les multiplieurs dédiés (*Dedicated multiplier*).

Les blocs logiques Les blocs logiques configurables (CLB, en anglais) constituent l'unité logique élémentaire d'un FPGA (cf. figure 2.19). Les blocs logiques se composent de deux parties : des bascules (flip-flops) et des tables de correspondance (LUT, pour Look-Up Table en anglais). La manière dont les bascules flip-flops et les tables de correspondance LUT sont assemblées diffère selon les types de FPGA. Dans le cas du Virtex® 7 de Xilinx®, il y a 2 flip-flops par LUT, 4 LUTs par slice et deux slices par CLB (cf. figure 2.19). Un FPGA de la série 7 de Xilinx® contient jusqu'à 2000 CLBs.

Les blocs mémoires Les blocs mémoires, ou BRAM, des FPGAs sont des mémoires configurables en mémoire simple dual-port ou vrai dual-port. Elles permettent des accès en lecture et écriture simultanées à deux adresses mémoire différentes. Un FPGA de la série 7 de Xilinx® contient des blocs RAM de 36Kb/18Kb et embarque jusqu'à 68Mb de mémoire de ce type.

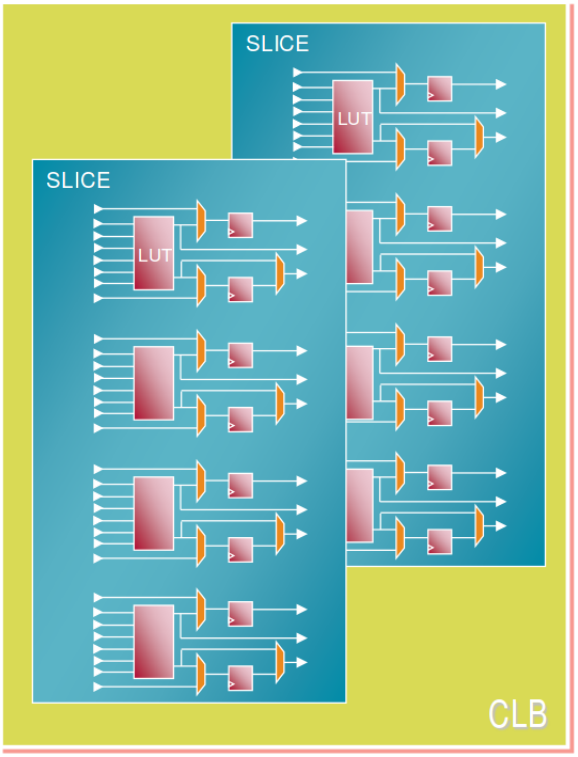

FIGURE 2.19 – Représentation simplifiée du contenu d'une LUT de FPGA. Source : Xilinx®

Les DSPs Les DSPs sont des unités de traitement optimisées pour le traitement du signal. Ils intègrent entre autres, un bloc de multiplications dédié, un bloc MACC (Multiplication et Accumulation), une ALU (Aritmetic Logic Unit). La figure 2.20, schématise un DSP d'un FPGA de la série 7 de Xilinx®. Cette série de FPGA peut contenir jusqu'à 1200 DSP.

FPGA vs CPU Le CPU est conçu pour exécuter des calculs de manière séquentielle, on dit qu'il implémente une distribution temporelle du calcul(cf. figure 2.21). Le FPGA, lui, permet une distribution spatiale, massivement parallélisable. La figure 2.21 met en parallèle le calcul de la même d'équation 2.25 sur CPU et sur FPGA.

$$y = Ax^2 + Bx + C \tag{2.25}$$

La fréquence maximale que peut atteindre un FPGA est plus basse que la fréquence maximale d'un CPU du fait de la topologie d'interconnexion qui doit être flexible et des finesses de gravure. Or, si une application doit exécuter un maximum de multiplications à virgule fixe en parallèle, le FPGA sera plus performant. Par exemple, pour un Virtex 7 il y a 3600 DSP slices (contenant chacune un multiplicateur à virgule fixe) pouvant atteindre la vitesse théorique maximale de 741MHz. L'application sera théoriquement équivalente à un processeur mono-coeur à 2,66THz (741x3600 MHz).

2.2.2.2 Conception FPGA

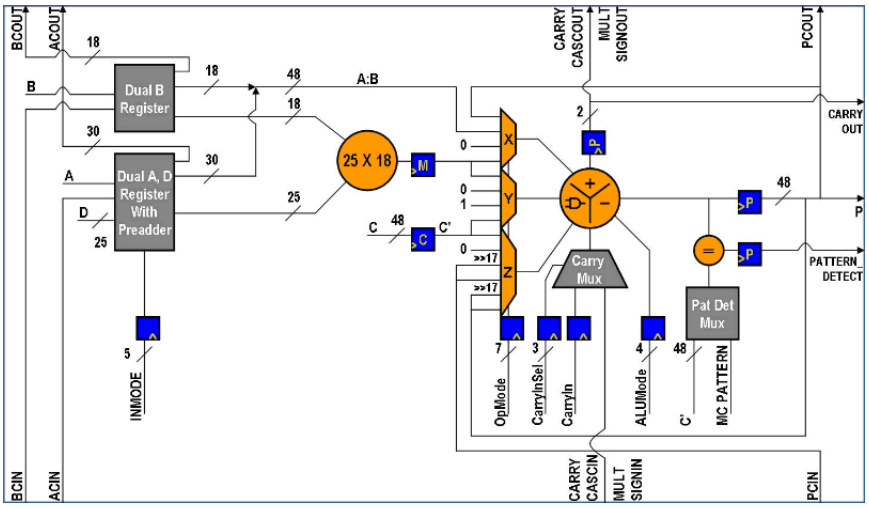

FIGURE 2.20 – Schématisation du contenu d'un DSP d'un FPGA. Source : Xilinx®

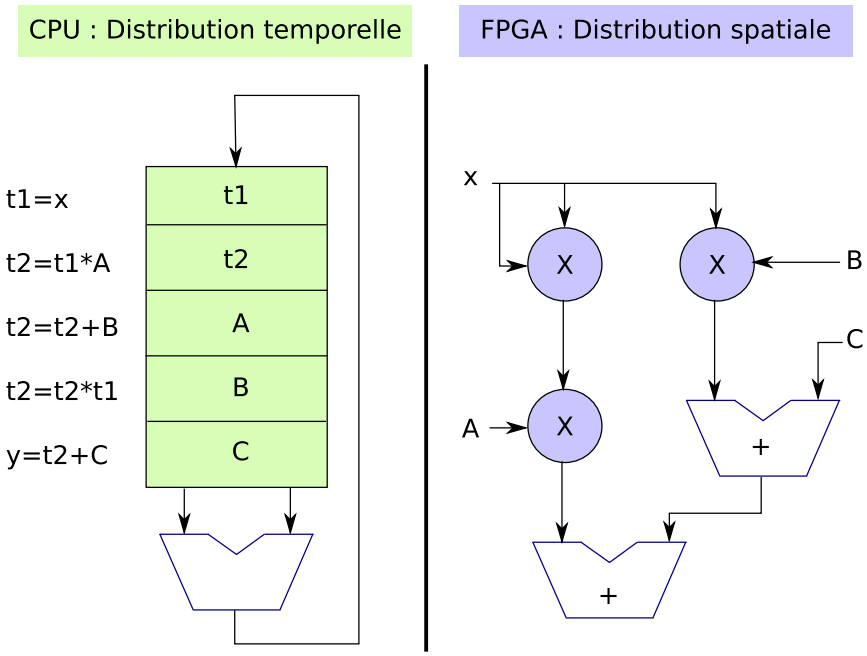

FIGURE 2.21 – Comparaison du traitement d'une fonction basique sur architecture CPU et FPGA.

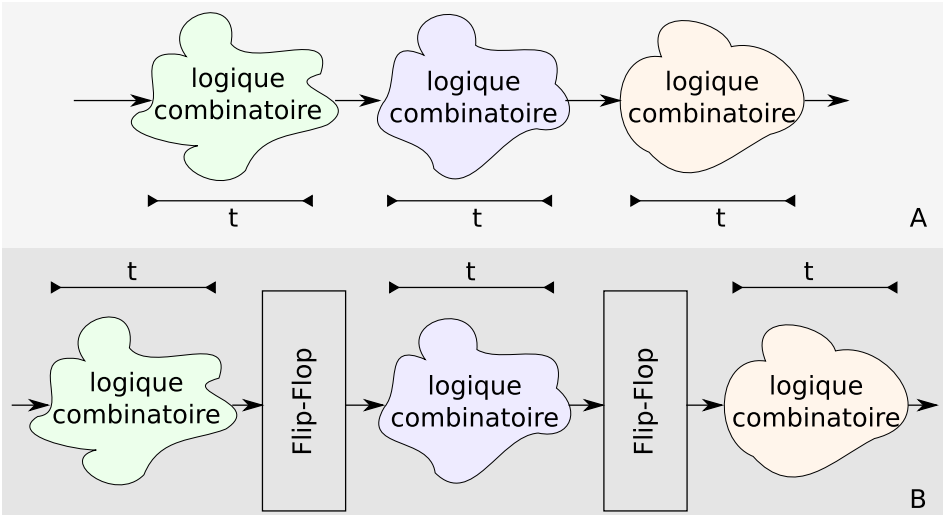

FIGURE 2.22 – Exemple de pipeline. A. Un exemple de calcul non pipeliné. B. Le même exemple avec un pipeline.

Pipelining Lors du développement FPGA, le concepteur doit tirer parti de l'architecture du FPGA pour utiliser efficacement les ressources. Le pipeline permet d'exprimer un ensemble séquentiel d'opérations sur un flux d'entrée, comme l'exécution en paralléle de ces différentes opérations sur des données indépendantes du flux. Dans le cas des FPGA, le pipeline permet par exemple de découper une longue séquence d'opérations combinatoires en plusieurs opérations combinatoires de petites tailles s'éxecutant en parallèle. La fréquence maximum d'une chaine d'opérations étant contrainte par le chemin combinatoire le plus long, le pipeline permet de réduire la longueur de ce chemin et ainsi augmenter la fréquence maximum du circuit.

Dans la figure 2.22(A), on peut voir un exemple simple d'une séquence de calcul combinatoire divisible en 3 parties. Le temps de latence de ce circuit est de $tps_{latence} = t + t + t$. Les données mettront $3t$ de temps à parcourir le circuit logique. Nous avons représenté la même séquence de calcul dans la figure 2.22(B) avec un pipeline. Le temps de latence sera égal à $tps_{latence} = t$. Le système pourra atteindre une fréquence maximale plus élevée et délivrer un résultat de calcul toutes les t secondes.

Chaîne de développement FPGA En développement FPGA, un système doit être décrit grâce à un langage de description par exemple VHDL, Verilog ou encore SystemC avant d'être intégré dans le FPGA. La figure 2.23 illustre cette chaîne. Une fois le design décrit et validé, un logiciel de synthèse le transforme en une Netlist décrivant le système à l'aide de portes logiques basiques. Puis chacune de ces portes sont placées et inter-connectées dans le FPGA, ce positionnement est placé dans un fichier appelé Placelist. Le résultat de ces deux phases différe en fonction du FPGA utilisé puisque chaque fabriquant et famille a une architecture spécifique. Enfin, une opération de bitgen génère un fichier bitstream binaire qui permet la configuration du FPGA.

Environnement de développement HDL Afin de réaliser les fonctions de synthèse et de bitgen (cf. figure 2.23) deux familles d'outils sont disponibles. Les premiers proposent une

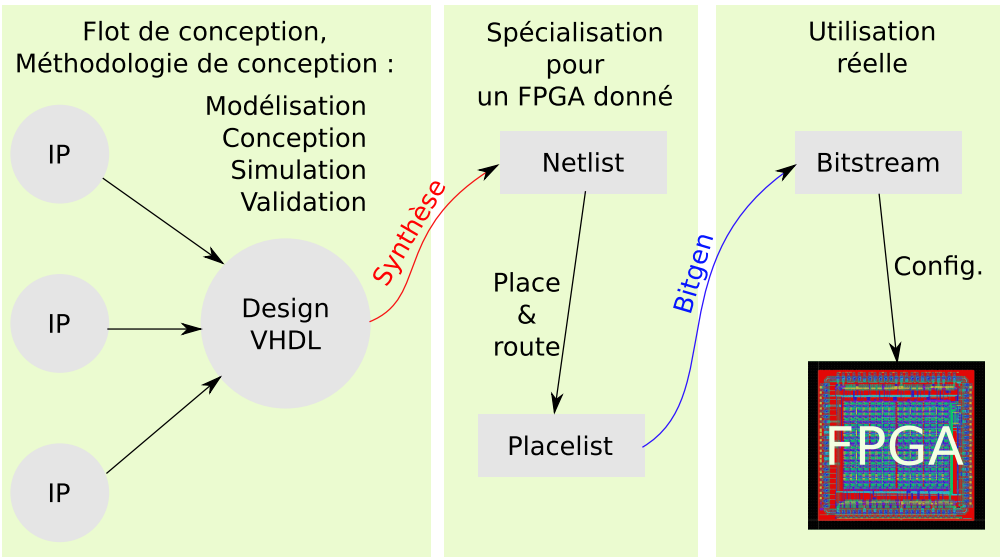

FIGURE 2.23 – Chaîne de développement FPGA.

chaîne de développement complète (jusq'à la génération du bitstream). Ces environnements de développement sont fournies par les fabriquants de FPGA. Xilinx® [23] fournit un logiciel pour une chaîne complète appelé Vivado® et supporte encore leur ancienne suite ISE®. Altera® [24] développe Quartus® II qui propose aussi la chaîne complète pour leur FPGA. Dans la deuxième catégorie on retrouve les logiciels de synthèse RTL [25] (ou de pré-synthèse), on peut citer par exemple Synplify® [26] (synthèse HDL) et Mentor Graphics HDL Designer [27]. Les résultats de pré-synthèse sont souvent plus optimisés que les logiciels fournis par les fabriquants de FPGA. Dans le cadre de notre application, le gain apporté est discutable comparé au prix de leur licence.

Langage de développement matériel - HDL Au cours des 20 premières années de développement sur FPGA, les langages de description du matériel (HDL pour High Description Langage) tels que le VHDL et Verilog sont devenus les langages de prédilection pour la conception d'algorithmes exécutables sur le circuit FPGA. Ces langages de bas niveau intègrent certains avantages généralement offerts par d'autres langages textuels et prennent en considération le fait que, sur un FPGA, c'est l'utilisateur qui architecture le circuit. La syntaxe hybride qui en résulte exige que les signaux soient mappés ou connectés à partir de ports d'E/S externes sur des signaux internes qui, à leur tour, sont câblés aux fonctions qui contiennent les algorithmes. Ces fonctions s'exécutent en séquences et peuvent faire référence à d'autres fonctions au sein du FPGA. La nature intrinsèquement parallèle de l'exécution des tâches sur un FPGA est toutefois difficile à visualiser dans un flux séquentiel ligne par ligne. Les langages HDL reflètent certains attributs des autres langages textuels comme le C mais ils en diffèrent considérablement car ils sont basés sur un modèle de flux de données dans lequel les E/S sont

23. http ://www.xilinx.com/content/xilinx/en/products/silicon-devices/fpga/
24. https ://www.altera.com/
25. Register-Transfer Level.
26. https ://www.synopsys.com/tools/implementation/fpgaimplementation/pages/synplify-pro.aspx
27. https ://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/

connectées à une série de blocs de fonctions via des signaux.

Les langage HDL permettent de décrire une architecture électronique qui est ensuite traduite en modèle RTL ((Register Transfert Level). Le HDL facilite la conception de circuits puisque d'après [Wakabayashi, 2004], pour une architecture de 1M de portes logiques il faut 300k lignes de code RTL.

Cependant, il permet de faire de la description bas niveau qui nécessite de développer l'application en prenant en compte tout le flux de données. Il faut, par exemple, gérer la lecture écriture dans les mémoires. Ces dernières années, les acteurs du domaine cherchent à mettre en place des outils de conception haut niveau pour la description matérielle afin de faciliter le développement FPGA.

Outils de développement haut niveau Contrairement aux langages HDL ou le concepteur modèlise l'application au niveau structurel ou RTL, les outils HLS permettent la saisie d'un modèle séquentiel classique dans une syntaxe C/C++ (ou autre) dont l'outil extraira une représentation structurelle ou RTL pour la synthèse matérielle. La description ainsi extraite doit pouvoir permettre l'optimisation de code séquentiel existant avec une connaissance mineure des architectures matérielles et ainsi favoriser la pénétration des FPGA sur le marché.

L'émergence d'outils de conception graphique HLS (High Level Synthesis), tels que LabVIEW FPGA® [28], Catapult® [29] ou VivadoHLS® [30], a fait disparaître certains obstacles majeurs du processus de conception traditionnel en HDL. L'environnement de programmation graphique LabVIEW est adapté à la programmation sur FPGA parce qu'il représente clairement la parallélisation et les flux de données. Cela permet aux utilisateurs, expérimentés ou novices en conception sur FPGA, de pouvoir décrire une architecture et l'implanter sur FPGA. Les outils de compilation LabVIEW FPGA permettent notamment, en cas erreur de timing, de mettre en évidence ces chemins critiques afin de savoir quelles fonctions doivent être accélérées. Pour simuler et vérifier le comportement de la logique FPGA, LabVIEW offre certaines fonctionnalités, directement dans l'environnement de développement. Il est possible de créer des bancs de test afin d'exécuter l'architecture développée.

VivadoHLS® de la société Xilinx® a choisi de supporter des langages haut niveau comme le Cpragma ou le SystemC (présenté dans le paragraphe 2.2.2.2).

De la même manière, Altera A++ Compiler® de Altera® supporte le language C/C++ et openCL et Catapult® [31] utilise le langage C/C++ pour faire du design HLS.

Ces outils présentent l'avantage de simplifier les développements en offrant une synthèse de haut niveau. Cependant, ils ne sont pas encore suffisamment matures pour être aussi efficace que la synthèse niveau RTL. D'après [Hill et al., 2015], à architecture équivalente, une synthèse HLS (OpenCL) utiliserait entre 60% et 70% de ressource logique de plus qu'une synthèse au niveau RTL (VHDL). En revanche, les performances en fréquence sont identiques (255MHz $<$ fmax $<$ 325MHz). Nous avons choisis de décrire notre architecture en langage RTL (VHDL) et d'utiliser la suite développement de Xilinx.

28. http ://www.ni.com/fpga/f/
29. https ://www.mentor.com/hls-lp/
30. http ://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
31. https ://www.mentor.com/hls-lp/

SytemC Après avoir constaté que C et C++ sont actuellement parmi les langages les plus utilisés, la société Synopsys a développé un nouvel environnement basé sur C++. L'un des objectifs de System-C est de fournir un outil permettant la description et la simulation de systèmes incluant des parties logicielles et matérielles. Une fois les tests achevés, le compilateur CoCentric System-C génère une netlist exploitable par les outils de placement-routage. Soutenu par des compagnie telles Xilinx, Synopsys, Innoveda ou Cadence, System-C concurrencera certainement VHDL.

SystemC est un langage de description matériel basé sur le langage C/C++. A la différence des langages de plus bas niveau tels que VHDL/Verilog, il permet de modéliser non seulement des systèmes matériels, mais aussi des systèmes logiciels mixtes ou non-partitionnés (où on ne sait pas encore ce qui sera fait en logiciel, et ce qui sera fait en matériel).

SystemC fait donc le pont entre les langages de description matérielle comme le VHDL et les langages de programmation logicielle comme le C. Il permet d'utiliser un langage commun pour du co-design et ainsi conduire le flot de conception avec une représentation unique.

La figure 2.24, illustre le diagramme du flot de conception en co-design en langage HLS (Hardware : VHDL/sytemC et software : C/C++). Nous avons :

— Flèches et boîtes rouges : flot de conception matériel. On modélise et on valide l'architecture au niveau RTL en SystemC (*RTL modeling*), on la traduit en VHDL pour pouvoir suivre le flot de conception HDL classique avec la synthèse (*synthesis*) jusqu'à la validation Hardware-In-the-Loop (cf. setion 3.1.4) de l'application (l'étape *SOC*).
— Flèches et boîtes violettes : flot de conception logiciel. Cette partie est évolutive (en phase avec l'étape *RTL modeling*) puisque l'utilisation du SystemC permet de modifier facilement le partitionnement matériel/logiciel au cours du développement.
— Flèche et boîtes vertes : flot de test. Un banc de test est élaboré en SystemC, il permet de tester l'application à tous les stades de la conception (simulation SystemC puis HDL, simulation post-synthèse et simulation/validation Hardware-In-the-Loop)

Ce diagramme met en avant l'omniprésence du langage systemC dans le flot de conception. En effet, contrairement aux langages HDL, il permet de combiner le haut niveau (Algorithm modeling) ainsi que les modèles RTL, software et test bench.

SystemC (cf. annexe A) est une extension du langage C++, plus précisément une bibliothèque de classes C++ contenant des modèles de matériel (modules, process), ainsi que toutes les briques de base pour modéliser un système matériel (registres, signaux). Elle comprend aussi un moteur de simulation événementiel rapide. Ce simulateur permet de faire des simulations plus rapides que les simulateurs HDL. Cette différence de vitesse vient du fait que le simulateur HDL interprète l'architecture alors que le systemC compile et produit un exécutable de l'architecture. La bibliothèque contient de plus, les éléments structurels qui représentent la structure des systèmes matériels (modules, ports de communication) et les éléments de communication entre processus et modules (événements, canaux).

Certains de ces éléments de communication servent à représenter du matériel (modules, ports, signaux, fifo), d'autres à représenter du logiciel (sémaphores, mutex).

SystemC est un langage relativement récent et de haut niveau. Les synthétiseurs haut niveau comme VivadoHLS® et Altera A++ Compiler® ne permet de transcrire du code haut niveau mais que dans certains contexte. En revanche, le SystemC peut s'avérer utile pour designer un premier prototype avant d'optimiser le code au niveau VHDL.

Le gros avantage de SystemC par rapport aux autres HDL vient du fait que ce n'est qu'un

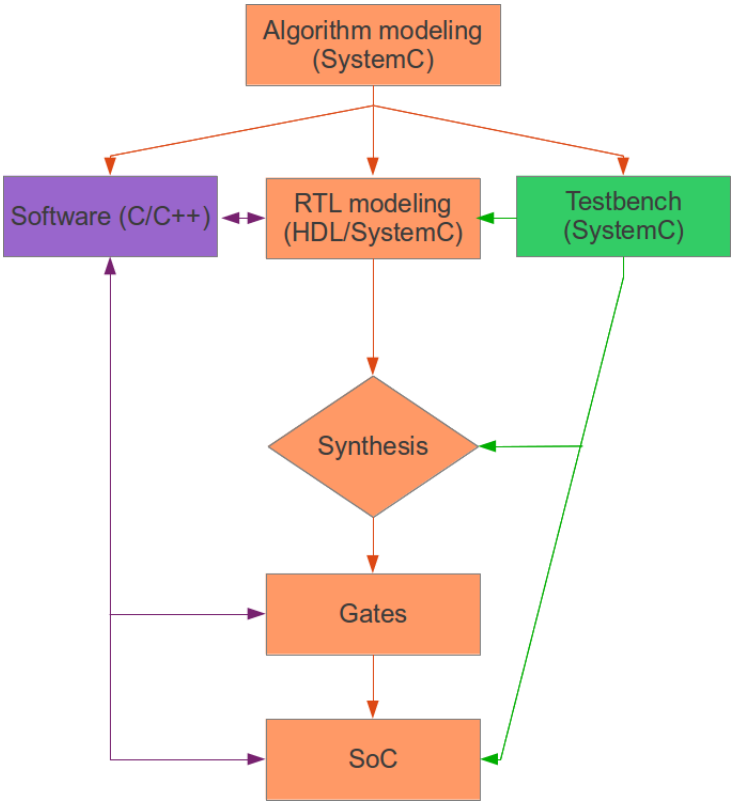

FIGURE 2.24 – Diagramme du flot de conception co-design.

ensemble de bibliothèques C++. Le seul outil nécessaire est donc un compilateur C++ (comme gcc/g++ : sous licence GNU). De plus, couvrant tous les types de modélisations comme RTL, TLM (modélisation niveau transactionnelle), il permet d'utiliser un seul et même langage durant tout le flot de conception d'un circuit. Étant donné que l'architecture est compilée, les simulations sont beaucoup plus rapides. Elles sont, de plus, plus efficaces pour scruter certains signaux pouvant être inclus dans des sous modules du module **top_level**. Enfin, SystemC offre tout l'environnement C/C++ et donc la bibliothèque openCV (open source) dédiée à la vision. Cette dernière permet d'élaborer une première architecture logicielle C/C++ de référence. On peut ensuite transcrire des fonctions (par exemple de la bibliothèque opencv) en blocs SystemC au fur et à mesure et ainsi tester et valider chacun de ces blocs.

Nos travaux s'appuient sur une chaîne SLAM logicielle existante (C-SLAM). Nous souhaitons l'implanter et la modifier pour qu'elle puisse valider nos contraintes temps réel (30 Hz). Nous avons utilisé SystemC et appliqué le flot de conception de la figure 2.24 au moment du développement d'accélérateurs matériels (cf. section 5.1.1.6). Nous allons voir dans la section suivante 2.3 que ce flot de conception s'intègre dans une méthodologie de co-design de système embarqué.

2.3 Méthodologies de co-design, modélisations et ordonnancement multi-opérateurs

Nous allons maintenant présenter des méthodologies de co-design qui incluent le flot de conception précédemment présenté (cf. section 2.3.1) et ajoutent des outils d'optimisation (cf. section 2.3.2) et d'ordonnancement multi-opérateurs (cf. section 2.3.3).

2.3.1 Méthodologies de co-design

Dans la littérature, il existe une multitude de travaux relatifs au développement de méthodologies appropriées au prototypage de systèmes embarqués. Une partie d'entre eux se réfèrent aux architectures hétérogènes et ont été classées dans [Shaout et al., 2009] en trois catégories :

- (A) Les approches top-down partant du but à atteindre, les spécifications comportementales du système final. Dans ce type de méthodologie, l'architecture se construit et évolue progressivement en ajoutant des fonctionnalités au design pour ainsi converger vers le cahier des charges fixé ;
- (B) Des approches basées design où l'architecture du système est prédéfinie et est utilisée pour en déduire le comportement du système ;
- (C) Et enfin, l'approche Bottom-up où un designer assemble l'architecture matérielle finale en interfaçant et interconnectant des composants validés auparavant.

Pour référencer efficacement les fonctionnalités d'un système dans un processus de co-design Software/Hardware, dans [Sciuto et al., 2002] les auteurs proposent une méthodologie pour définir des critères d'analyse en co-design (*co-design analysis metrics*) dans le but de classer chaque tâche en fonction du/des type(s) d'unité de traitement à préférer pour leur exécution (microprocesseur, FPGA, DSP, GPU etc). Cependant, comme le font remarquer les auteurs, il ne peut y avoir une seule méthode efficace couvrant tous les domaines d'applications. Dans [Vincke et al., 2012] une approche basée plateforme (le type de plateforme oriente les

choix de conception) est utilisée pour développer une application EKF-SLAM visuel bas-coût sur une architecture multi-cœurs avec un DSP (Digital Signal Processor). Les auteurs concluent en présentant l'étape suivante du développement, l'implantation de leurs travaux sur une plate forme hétérogène dédiée sur un ASIC. C'est l'ultime étape commune à toutes ces méthodes de développement co-design.

Nous nous sommes basé sur une méthodologie top-down plus générale (sans à proiri sur le choix de l'architecture cible) pour co-designer un EKF-SLAM visuel sur une architecture hétérogène. La méthodologie de base est présentée dans [Shaout et al., 2009] et se divise en 4 étapes :

1. Trouver et adapter l'application à l'analyse globale des critères de co-design ;
2. En considérant les mesures de performance du système, identifier les goulets d'étranglement ;
3. Re-partitionner l'architecture, élaborer des modules Hardware et les intégrer dans le design ;
4. Si les contraintes ne sont pas satisfaites, réitération depuis l'étape 2, sinon stop ;

La figure 2.25 illustre la méthodologie de co-design que nous avons adaptée à nos besoins (nous avons intégré le flot de conception évoqué dans la présentation de SystemC dans la section 2.2.2.2) et la figure 2.26 résume les différentes étapes et illustre le processus itératif de la conception.

Nous allons maintenant résumer les étapes de cette méthodologie (figure 2.26). Elles seront détaillées au moment de leur mise en application pour optimiser notre chaîne EKF SLAM, en section 3.

Afin d'explorer efficacement l'espace de conception (l'étendue des degrés de liberté) durant la phase de *Modelling* (cf. figure 2.26), une modélisation flux de données (ou Dataflow modelling, présentée plus en détail dans 2.3.2) permet de représenter notre application sous la forme d'un flux de données. Cette modélisation permet de travailler sur le niveau de parallélisme que l'application peut exposer dans le but d'optimiser son exécution. C'est une optimisation dirigée par les modèles. Le degré de parallélisme exposé fixe les métriques d'analyse du co-design. En parallèle, on spécifie les contraintes de notre design comme la consommation, l'empreinte mémoire, l'encombrement, la vitesse, la précision ou encore le délai de commercialisation. Elles influenceront le partitionnement matériel/logiciel et valideront le prototype final.

Dans l'étape *HW specifications* (ou spécifications matérielles), les mesures du coût du système sont choisies en adéquation avec la carte de développement mixte qui pourrait convenir au degré de parallélisme choisi. Puis, les tâches sont classées en fonction de leur degré de parallélisme et les spécifications sont mises à jour en fonction de cette répartition. Dans la phase de *Co-synthesis*, le design est partitionné de manière incrémentale ([Balboni et al., 1996]) en blocs logiciels et matériels. L'effort de répartition des modules Sw/Hw est porté sur la déportation de tâches sur matériel jusqu'à satisfaire les performances globales désirées. Pendant cette phase, un processus de *Hardware in the loop* est utilisé afin de s'assurer que les IPs (Intellectual Properties) soient compatibles avec le reste du système. Enfin, dans la phase de *Prototyping*, plusieurs prototypes fonctionnels sont créés (un à chaque itération de la méthodologie) jusqu'à obtenir un prototype conforme aux exigences du cahier des charges.

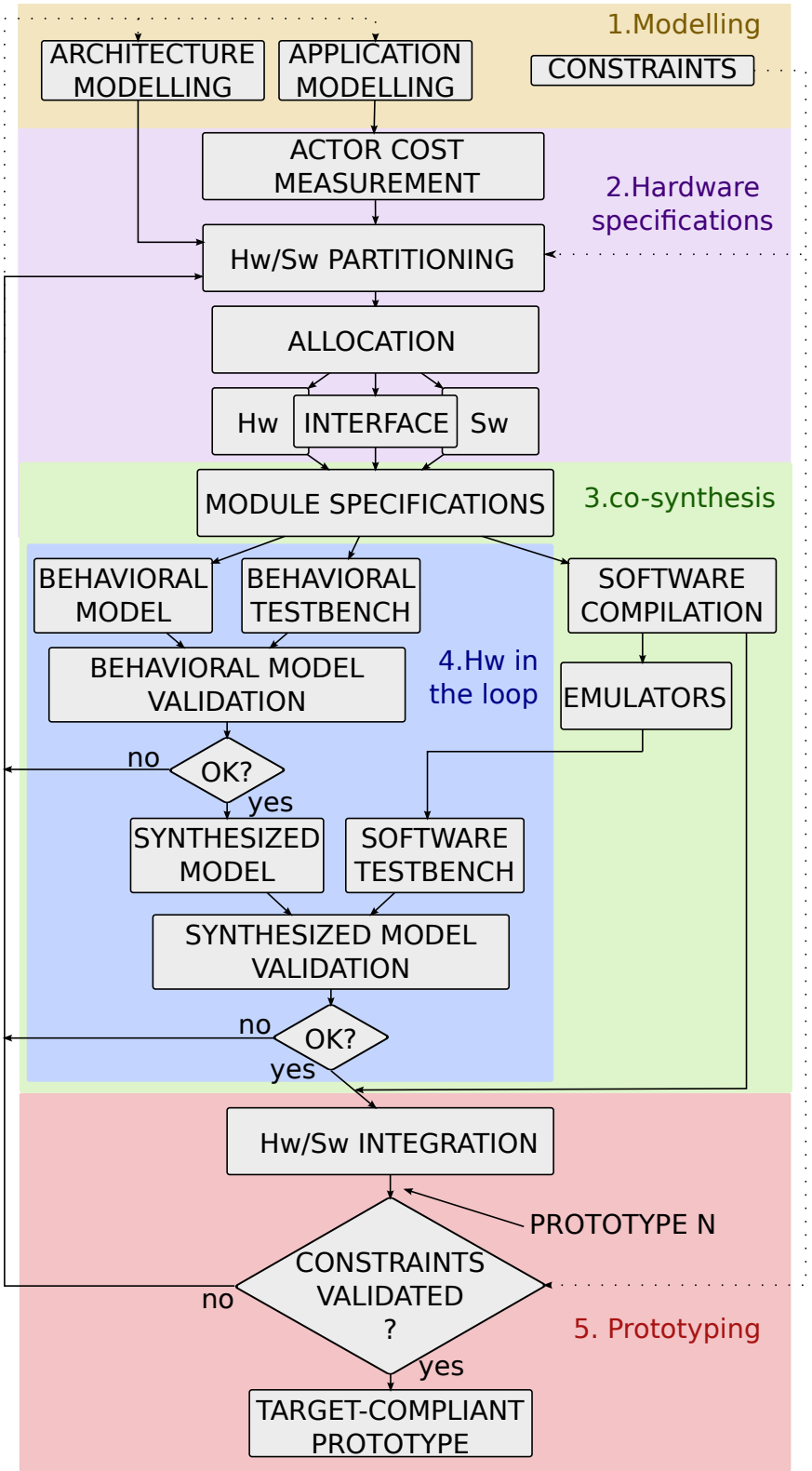

FIGURE 2.25 – Méthodologie de co-design HW/SW appliquée au prototypage d'un EKF-SLAM sur architecture hétérogène.

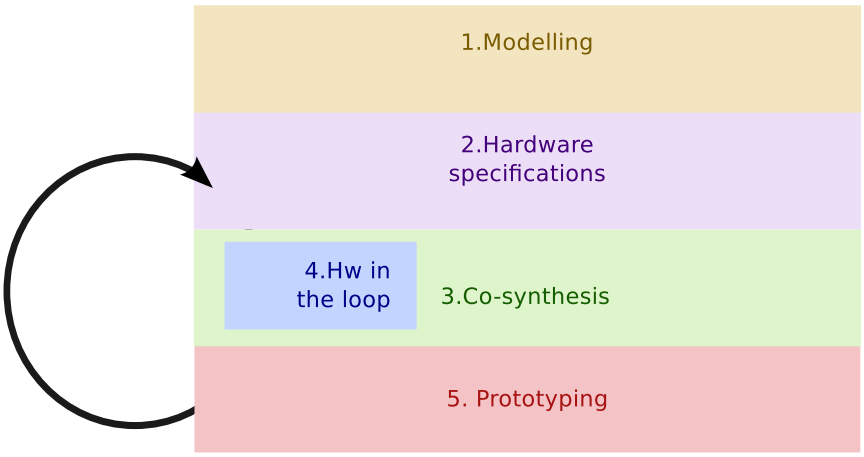

FIGURE 2.26 – Résumé de la méthodologie de co-design : *1.modélisation*, *2.Specifications hardware*, *3.Co-synthèse*, *4.HIL* et *5.Prototypage*.

2.3.2 Modélisations flux de donnée

Les applications telles que l'encodage/décodage vidéo ou les communications numériques avec des fonctionnalités avancées sont de plus en plus complexes et le besoin de puissance de calcul augmente en conséquence. Afin de satisfaire les exigences logiciel, l'utilisation de l'architecture parallèle s'est généralisé. Pour réduire l'effort de développement logiciel induit par ce type d'architecture, des outils facilitent la résolution du paritionnement et l'ordonnancement du système. Ces derniers assurent une modélisation de l'application afin de permettre une optimisation anticipée et assure que le système se comporte correctement lors de l'exécution. Nous allons faire une présentation des modélisations flux de données, en particulier, le modèle flux de données synchrone (de l'anglais Synchronous Data-Flow, SDF).

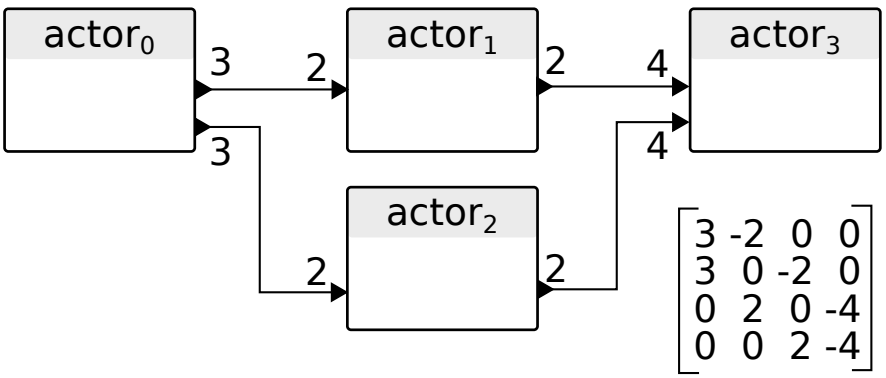

FIGURE 2.27 – Exemple d'une modélisation Data-Flow Synchrone ou SDF avec ses acteurs, arcs, jetons et sa matrice topologique en bas à droite (cf. section 3.2.2).

Le modèle de flux de données vise à représenter une application en prenant en compte les dépendances de données qui existent entre les différentes parties de l'application. L'application est donc représentée comme un réseau de plusieurs actions. Ce type de représentation permet d'obtenir du parallélisme intrinsèque et identifier les besoins mémoire de l'application.

Les concepts du modèle sont :

— l'activité : un ensemble homogène de traitements qui transforme ou manipule des données ;
— l'acteur : représente une unité active intervenant dans le fonctionnement d'un système opérant. Il peut être stimulé par des flux, transformer des flux et/ou renvoyer des flux ;
— le flux : représente un échange entre deux acteurs ;

Les modèles de calcul flux de données s'attachent à décrire le comportement d'une application au travers des dépendances de données qui existent entre les tâches qui la composent. La description flux de données repose sur trois éléments illustrés dans un exemple en figure 2.27 :

— Acteur : un acteur représente une entité de calcul de l'application ;
— Jeton de données : un jeton de données est l'élément d'une donnée atomique que s'échangent les acteurs ;
— Arc : un arc connecte deux acteurs. Un acteur produit des données sur l'arc que l'acteur/consommateur peut ensuite consommer.

Le paradigme de flux de données a été introduit par Kahn dans [Kahn, 1974]. Le *Kahn Process Network* est un réseau d'acteurs reliés par des FIFOs unidirectionnelles non limitées qui transportent des jetons de données. Un jeton de données est un bloc de données atomique (il ne peut pas être divisé). Kahn a également établi une représentation formelle pour le Kahn Process Network. Un acteur peut être décrit comme un processus fonctionnel qui mappe un ensemble de séquences de jetons dans un ensemble de séquences de jetons. Les réseaux de processus flux de données décrits dans [Lee and Parks, 1995] héritent du *Kahn Process Network* mais attribuent à chaque acteur un ensemble de règles de déclenchement (firing rules) qui définissent le nombre minimum de jetons nécessaires pour qu'un acteur s'exécute.

2.3.2.1 Modélisation Synchronous Data Flow (SDF)

Pour permettre une analyse anticipée plus approfondie, le modèle flux de données synchrone (ou *Synchronous Data-Flow*, SDF) [Lee and al, 1987] reprend les réseaux de processus flux de données et les limite en spécifiant le taux de production/consommation de données pour chaque acteur. Ces informations permettent de s'assurer qu'il n'y a pas de blocage de l'application. De plus, elles permettent d'en déduire un ordonnancement statique et d'identifier les besoins mémoire de l'application. Afin de permettre l'expression d'un modèle SDF, plusieurs modèles ont été proposés. Le modèle *flux de données booléen* (BDF) [Tobin Buck, 1993] propose d'inclure des acteurs spécifiques dans le modèle dont les taux production/consommation peuvent être contrôlés par une entrée conditionnelle prenant des valeurs booléennes. Le modèle flux de données Cyclo-statique (FCDE pour *Cyclo-static Data Flow*) [Bilsen et al., 1995] permet de décrire les taux de production/consommation comme une séquence d'entiers, ce qui permet à l'acteur de se comporter d'une manière différente en fonction d'une séquence d'activations. Le flux de données synchrone paramétré (ou *Parametrized Synchronous DataFlow*, PSDF) [Bhattacharya and Bhattacharyya, 2001] introduit des paramètres qui affectent le taux production/consommation et le comportement structurel d'un acteur. Ces modèles offrent une meilleure expression du SDF mais diminuent la possibilité d'analyse du réseau.

Le Synchronous Data Flow (SDF, [Lee and al, 1987]) permet de décrire des systèmes afin d'obtenir un ordonnancement statique périodique efficace dans le cas de composants synchrones

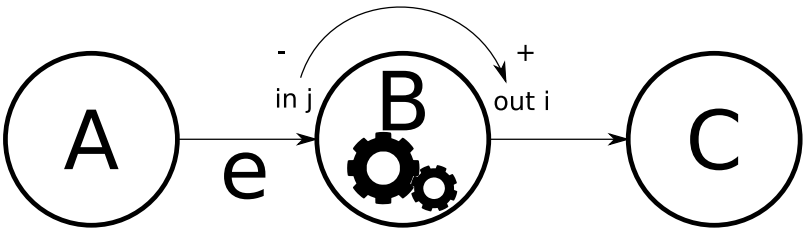

FIGURE 2.28 – Principe et des productions/consommations de données dans un graphe SDF.

fonctionnant à la même fréquence d'horloge mais disposant d'échantillonnages différents [32]. Formellement, un graphe SDF est un graphe orienté fini $G(V, E)$ où V est l'ensemble des acteurs, E l'ensemble des arcs. La notation $e = a \rightarrow b$ signifie que l'arc e vient de l'acteur a et entre dans l'acteur b (cf. figure 2.28). Nous affectons un nombre $M(e)$ de jetons (non négatif) à chaque arc e. Cette fonction est appelée marquage. À chaque acteur on associe un ensemble d'entrées et sorties qui sont toutes annotées par un nombre strictement positif (ou poids). Un acteur est dit exécutable si le nombre de jetons sur chacun des arc entrants est égal ou supérieur au poids associé à l'entrée correspondante. Une exécution d'un acteur ajoute *out* i jetons à chaque sortie et retire *in* j jetons à chaque entrée où *out* i et *in* j sont les poids associés à une sortie i de l'acteur et respectivement à une entrée j de l'acteur (cf. figure 2.28).

Le problème principal dans ce modèle est d'identifier l'inconsistance entre les taux de production/consommation qui demandent des ressources infinies de stockage temporaire. Un graphe SDF peut être caractérisé par une matrice où nous affectons une colonne à chaque acteur et une ligne à chaque arc. La (i, j)ième entrée dans la matrice est la quantité de données produites et consommées par un acteur j sur l'arc i, la valeur consommée est négative et celle similairement produite est positive. Nous appellerons une telle matrice "matrice topologique" (voir l'exemple figure 2.27). Cette entrée donne la différence entre la quantité de données produites et consommées à chaque fois par cet acteur.

Lee a montré dans [Lee and al, 1987] et [Lee and Messerschmitt, 1987] qu'une condition nécessaire pour l'existence d'un tel ordonnancement est que le rang de la matrice topologique r est $rang(r) = s-1$ où s est le nombre d'acteurs dans la matrice. Même si les taux de production/consommation sont consistants, il peut être impossible de construire un ordonnancement périodique admissible.

2.3.3 Ordonnancement multi-opérateurs et partitionnement

Grâce à un modèle des caractéristiques de l'architecture multi-opérateurs et un modèle flux de données de l'application, il est possible de déterminer la meilleure implantation de l'application sur l'architecture. Pour y parvenir, il est tout d'abord nécessaire de répartir les acteurs qui composent la description flux de données sur les différents opérateurs de l'architecture. Cette opération est appelée "allocation". Il faut ensuite, pour chaque processeur, décrire l'ordonnancement des acteurs et organiser les communications entre opérateurs. Cette implantation cherche à optimiser un ou plusieurs critères d'exécution tel que le temps d'exécution

32. De même, si deux acteurs ont des largeurs de bus différentes en entrée/sortie, leur fréquence de consommation de donnée et celle de production sera différente.

(cadence ou latence), l'empreinte mémoire et/ ou la consommation électrique.

Ordonnancement multi-opérateurs Sur une architecture multi-opérateurs, l'ordonnancement vise à attribuer un ensemble de tâches aux opérateurs / processeurs disponibles de l'architecture et détermine ensuite le temps de démarrage des tâches sur chacun des processeurs. Trouver cet ordonnancement est un problème NP-complet (Nondeterministic Polynomial time).

L'ordonnancement d'un modèle flux de données statique repose sur l'analyse du modèle DAG (Directed Acyclic Graph) équivalent. Deux nœuds d'un DAG ne peuvent être reliés que par un arc associé à un poids qui représente le coût du transfert de données. La plupart des algorithmes d'ordonnancement de DAG sont basés sur un ordonnancement par liste [Adam et al., 1974]. Dans la liste ordonnancement, on attribue aux nœuds une priorité et on les classe dans une liste par ordre décroissant des priorités. Ensuite, les nœuds de la liste sont analysés et ordonnancés en suivant l'ordre de cette liste. Différentes méthodes proposent de guider cette attribution des priorités.

Deux attributions sont fréquemment utilisées, le *t-level* et le *b-level*. Le *t-level* d'un nœud n_i est la longueur du chemin le plus long à partir d'un nœud d'entrée où, la longueur d'un chemin est la somme de tous les nœuds et tous les poids le long de ce chemin. Le *t-level* d'un nœud est la longueur d'un chemin le plus long entre le nœud n_i et un nœud de sortie. La complexité en temps de ces deux attributions est $O(e+v)$. Le *t-level* est contraint par la taille du chemin critique tandis que le *b-level* est tenu par le chemin le plus long du DAG.

Les algorithmes d'ordonnancement par liste attribuent des priorités aux nœuds et définissent des critères de choix de processeur. En premier lieu, l'algorithme construit la liste ordonnée de nœuds en se basant sur les priorités assignées. La pertinence de l'ordre de nœud pour la longueur de l'ordonnancement a été démontrée dans [Sinnen and Sousa, 2005].

La complexité des algorithmes d'ordonnancement par liste dépend fortement du nombre d'acteurs. Un simple algorithme d'ordonnancement par liste comme celui présenté dans [Sinnen, 2007] a une complexité de $O(P(V+E))$. [Kwok and Ahmad, 1998] présente un autre algorithme d'ordonnancement par liste appelé Highest Level First with Estimated Time et [Wu and Gajski, 1990] expose un second appelé Modified Critical Path. Tout deux ont une complexité en $O(pv^2)$.

Partitionnement Le modèle SDF permet de définir un réseau qui révèle le parallélisme d'une application. Le comportement du réseau est déduit des dépendances de données entre les acteurs et extraire un partitionnement valide est alors relativement simple. Dans le cadre d'un prototypage rapide, la spécification de l'application est optimisée avant la programmation. Lorsque l'on essaie d'optimiser ce type réseau pour une architecture parallèle, il faut considérer le parallélisme existant entre plusieurs invocations d'acteurs. Le modèle SDF qui peut être extrait de la représentation SDF expose tout le parallélisme de l'application mais conduit à une explosion du nombre d'acteurs à ordonnancer. En l'absence d'architecture à degré infini de parallélisme, on doit choisir d'extraire une quantité de parallélisme limitée et adaptée à l'architecture cible.

Lorsque l'on utilise une spécification hiérarchique de l'application, tout le parallélisme enfoui au niveau hiérarchique reste indisponible au partitionnement. Pour extraire tout le parallélisme, la meilleure approche serait de mettre à plat tous les niveaux de la hiérarchie

pour exposer le parallélisme à grain fin. Encore une fois, tout ce parallélisme disponible ne pourrait pas être adapté au niveau de parallélisme disponible sur l'architecture cible.

2.4 Synthèse

Le but de nos travaux est d'implanter la **première chaîne 3D EKF SLAM monoculaire embarquée complète sur une architecture mixte** (CPU-FPGA) répondant au contraintes temps réel des systèmes ADAS que nous nous sommes fixé (un temps de traitement supérieur à 30 img/s, une latence inférieur à une image et une consommation inférieur à 5W). Pour cela nous nous basons sur une implantation logicielle existante au LAAS. Cette application (C-SLAM) est codée en langage C et fonctionne en temps réel sur un processeur Intel® i5.

Configuration de C-SLAM Dans le but d'implanter C-SLAM sur une architecture embarquée, nous l'avons configuré et modifié de la manière suivante :

- Utilisation des capteurs : caméra, GPS et IMU ;
- Résolution caméra de 640x480 ;
- Paramétrisation des amers de type AHP ;
- Suivi de 20 amers simultanément ;
- Utilisation du détecteur de points d'intérêt de Harris (par la suite remplacé par FAST, moins coûteux en calcul et plus facilement implantable sur architecture matérielle) - cf. paragraphe 2.1.5 ;
- Tessellation de l'image afin de limiter le coût de traitement. La détection des points n'est pas calculée sur l'image entière mais seulement dans les zones où aucun amer n'est observé. La tesselation est détaillée en section 2.1.7 ;
- Sélection d'amers à corriger réalisée par Recherche Active, présentée en 2.1.7.
- Élection des points d'intérêt à initialiser dans la carte par tirage au sort.

Choix d'implantation Nous avons choisi d'utiliser FAST en nous basant sur les résultats de l'implantation FPGA de [Kraft et al., 2008] que nous avons optimisé ainsi que ceux de l'implantation logicielle de [Rosten and Drummond, 2005] (cf. paragraphe 2.1.5). Ce choix nous permet d'accélérer le traitement de la détection de points d'intérêt en logiciel en remplaçant le détecteur de Harris par FAST dans C-SLAM puis, une fois le comportement du SLAM validé, d'implanter FAST sur l'architecture matérielle pour accélérer le traitement tout en gardant les mêmes performances en précision et robustesse que C-SLAM.

Afin de décrire les points d'intérêt du détectés par FAST, nous avons choisi d'utiliser le descripteur BRIEF (cf. paragraphe 2.1.6). Il présente l'avantage d'être léger (en terme de traitement) rapide et robuste. De plus, il présente des caractéristiques intéressantes pour une implantation matérielle.

Dans le but de mettre en correspondance des points d'intérêt dans une image, nous avons développé un IP matériel original de corrélation de descripteurs (BRIEF) basé sur le calcul de la distance de Hamming (cf. paragraphe 2.1.7).

Enfin, l'implantation de l'algorithme EKF-SLAM sur architecture mixte CPU-FPGA a été guidée par une méthodologie de co-design itérative que nous avons adapté de [Shaout et al., 2009] (cf. paragraphe 2.3.1) que nous appliquerons à notre système en chapitre 3. Nous

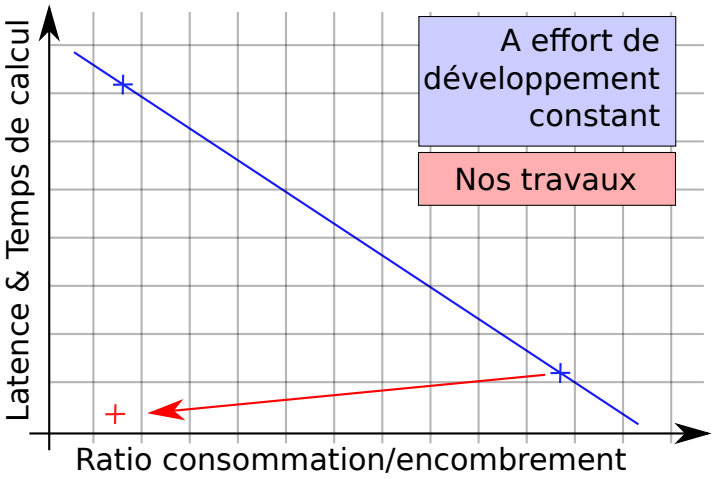

FIGURE 2.29 – A effort de développement constant, le ratio consommation/encombrement est inversement propotionnel aux performances en vitesse du sytème

avons réalisé 4 itérations de cette méthodologie avant d'obtenir un prototype satisfaisant les contraintes que nous nous sommes fixées.

Contraintes Voici un récapitulatif des contraintes de notre système :

- Temps de traitement : nous souhaitons que notre application soit capable de traiter un EKF-SLAM en temps réel. Nous avons défini ce temps réel à 30Hz (cf. paragraphe 1.3) ;
- Latence : nos travaux visent à localiser un robot ou un véhicule autonome. Il est donc intéressant de minimiser la latence du système pour fournir une mesure la plus actualisée possible. Nous fixons une latence de maximum 1 image ;
- Consommation : nos travaux s'inscrivent dans le contexte des systèmes embarqués ADAS. Nous souhaitons obtenir un prototype consommant moins de 5W ;
- Encombrement : le système final doit pouvoir être implanté dans le boitier d'une smart-caméra. Nous avons donc fixé la contrainte d'encombrement physique du produit fini (sans la caméra) à 5cm×5cm.

Dans le cadre d'un système embarqué, le challenge est de maintenir des contraintes de temps réel et de latence faible tout en maîtrisant la consommation et l'encombrement. Comme schématisé sur la figure 2.29, à effort de développement constant, l'augmentation de la fréquence de traitement du système est souvent synonyme de l'augmentation du ratio consommation/encombrement du système.

Afin d'obtenir un bon rapport : tempsDeTraitement/latence VS consommation VS encombrement, nos travaux ont pour but d'**optimiser** les traitements afin de maintenir un ratio consommation/encombrement acceptable pour des performances très élevées. Pour cela, nous avons donc joué sur le paramètre : effort de développement.

Dans la partie suivante, partie II, nous allons présenter les travaux de recherche et décrire les itérations de la méthodologie de co-design réalisées pour intégrer une chaine 3D EKF-SLAM dans une architecture mixte embarquée.

Deuxième partie

Travail de recherche - Intégration d'une chaine SLAM embarquée

La partie I était consacrée à la présentation du contexte des travaux de cette thèse en introduisant les systèmes ADAS, le projet DICTA, les objectifs des nos travaux et en synthétisant les contributions liées à cette thèse. Après avoir posé le cadre de nos recherches, nous avons ensuite dresser l'état de l'art des techniques de SLAM existantes (par filtre et par optimisation), nous avons expliqué la méthode EKF-SLAM et nous avons fait un état de l'art des chaînes SLAM vision embarquées. Nous avons ensuite introduit les chaînes SLAM par vision embarquées opérationnelles au LAAS (RT-SLAM et C-SLAM). Nous avons ensuite effectué un état de l'art des détecteurs et descripteurs de points d'intérêt existants et nous expliqué le principe de la recherche active dans C-SLAM. Nous avons ensuite comparé les différentes architectures mixtes existantes puis nous nous sommes concentré sur la technologie FPGA et le flot de conception associé. Enfin nous avons terminé cette partie I en détaillant la méthodologie de co-design que nous avons appliquée à nos développements et les concepts d'optimisation par modélisation flux de données.

Nous allons, dans la partie II, exposer nos travaux de thèse relatifs à l'intégration d'une chaîne 3D EKF SLAM opérationnelle embarquée. Nous présenterons les différents IP matériels originaux que nous avons conçu (détecteur et descripteur de points d'intérêt, une achitecture de suppression de non maxima et un corrélateur de descripteurs de points d'intérêt). De plus, nous appliquerons la méthodologie de co-design à notre application EKF SLAM et détaillerons les optimisations effectuées (au niveau structurel, logiciel et matériel).

Nous allons décrire toutes les itérations de la méthodologie de co-design que nous avons dû effectuer avant d'obtenir un prototype satisfaisant les contraintes fixées (cf. section 2.4). Cette partie II est construite en quatre chapitres. Afin de faciliter la lecture et bien décorréler les travaux effectués dans le cadre de cette thèse de ceux effectués par Daniel Törtei (doctorant au LAAS qui a contribué à cette intégration), nous avons choisi de ne pas présenter les itérations effectuées dans l'ordre chronologique.

Ainsi, dans le chapitre 3, nous allons détailler les étapes de la méthodologie de co-design et nous modéliserons le système afin de l'optimiser (cf. colonne grise dans la figure 2.30).

Dans le chapitre 4, nous présenterons le travail d'optimisation et d'intégration logicielle effectué. Ces développements ont été effectués lors de la **première** itération de la méthodologie de co-design. Puis nous présenterons les travaux de Daniel Törtei qui a effectué la **troisième** itération de la méthodologie (cf. colonne grise dans la figure 2.30).

Dans le chapitre 5, nous exposerons le reste des travaux réalisés dans le cadre de cette thèse. Nous détaillerons la **deuxième** itération dans laquelle on intègre un accélérateur vision pour la tâche de détection de points d'intérêt. Puis, nous décrirons la **quatrième** et dernière itération de la méthodologie : l'intégration d'un accélérateur matériel pour la tâche de corrélation (cf. colonne grise dans la figure 2.30).

Nous terminerons cette partie II par la conclusion générale de ce document et ouvrirons sur les perspectives envisagées.

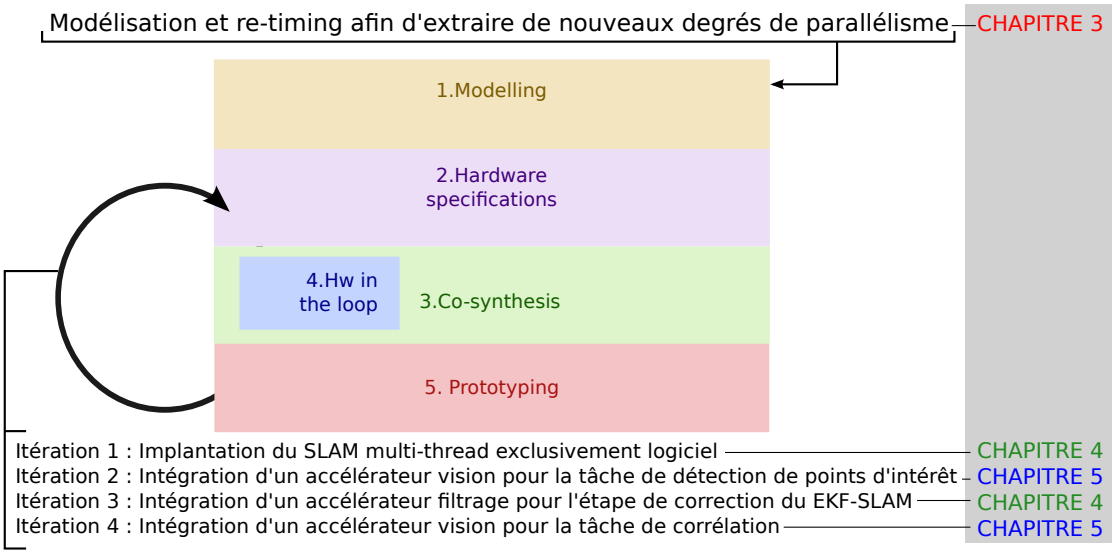

FIGURE 2.30 – Organisation de la partie II du mémoire et étapes du co-design itératif d'une chaîne 3D EKF SLAM complète embarquée.

Chapitre 3

Application de la méthodologie de co-design

Dans la première partie de ce chapitre nous allons présenter les différentes étapes de la méthodologie de co-design que nous avons utilisée (introduite en section 2.25). La seconde partie de ce chapitre est dédiée à l'optimisation niveau de structurel de notre application.

Dans un premier temps, nous allons décrire toutes les étapes de la méthodologie de co-design itérative que nous avons adapté de [Shaout et al., 2009] (cf. figure 3.1) : *Modelling* (cf. paragraphe 3.1.2) ; *Hardware specifications* (cf. paragraphe 3.1.3) ; *Co-synthesis* (cf. paragraphe 3.1.4) ; *Hw in the loop* (cf. paragraphe 3.1.4) ; *Prototyping* (cf. paragraphe 3.1.5).

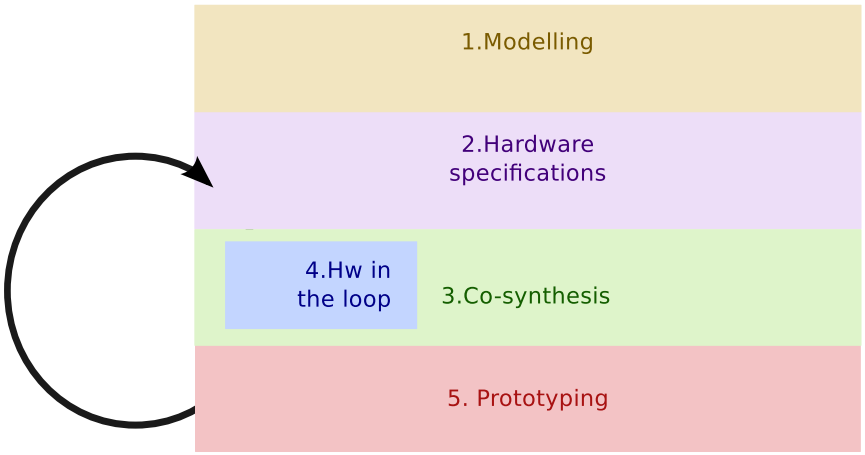

Figure 3.1 – Résumé de la méthodologie de co-design : *1.Modélisation*, *2.Specifications hardware*, *3.Co-synthèse*, *4.HIL* et *5.Prototypage*.

Dans un deuxième temps, nous appliquerons les premières étapes de *Modelling* et de *Hardware specifications* (en section 3.2) de cette méthodologie à notre chaîne SLAM complète. Dans l'étape de *Modelling*, nous modéliserons notre application afin d'extraire de nouveaux degrés de parallélisme grâce à des techniques de re-timing. Puis, dans la deuxième étape (*Hardware spécifications*), nous allons statuer sur une architecture de traitement hétérogène comprenant un CPU et de la logique FPGA : le ZynQ®-7020. Enfin, nous terminerons ce chapitre en proposant un partitionnement théorique SW/HW.

3.1 Flot de conception global

Dans cette section, nous allons décrire étape par étape la méthodologie de co-design. Nous présenterons, dans un premier temps, le processus itératif que cette méthodologie propose (en

section 3.1.1). Puis, nous détaillerons, dans les sections 3.1.2 à 3.1.5, les étapes de ce flot de conception co-design en nous référant à la figure 3.2.

3.1.1 Prototypage itératif

La méthodologie de co-design que nous utilisons permet de faire un prototypage de type itératif. Ceci implique que l'architecture évolue au fur et à mesure du développement tant que le système ne satisfait pas les contraintes fixées (temps d'exécution, consommation, latence, encombrement, cf. section 2.4). Cette méthodologie permet à chaque itération de remettre en cause les choix opérés au cours des itérations précédentes. On peut, par exemple, reconsidérer le choix de la plateforme de traitement ou le partitionnement logiciel/matériel tout au long du processus de prototypage. Cependant, nous avons essayé de statuer sur une architecture cible dès le début de la conception afin de limiter le temps de développement. En partant d'une base EKF-SLAM purement software développée précédemment au LAAS (C-SLAM - [Gonzalez et al., 2011] - cf. section 2.1.4), ce processus de développement co-design itératif permet de proposer, à chaque nouvelle itération, de nouveaux partitionnements de l'application et d'accélérer des fonctions spécifiques préalablement identifiées comme goulot d'étranglement.

3.1.2 Spécifications des contraintes & Modélisation

Dans l'étape de modélisation (*Modelling*), la première phase est de définir les **contraintes** de notre système (*Constraints*, cf fig 3.2). Elles ont été définies dans la section 2.4. La satisfaction de ces contraintes valide le prototype. Dans la figure 3.2, cette validation est représentée par le test : ***constraints validated*** dans lequel on évalue les performances du système.

En parallèle de cette définition de contraintes, deux étapes de modélisation (***Architecture modelling*** et ***Application modelling*** cf. figure 3.2) permettent de représenter notre application sous la forme de modèles. Ces phases permettent notamment d'optimiser le système et d'orienter nos choix futurs de partitionnement et d'implantation. Ces deux étapes sont présentées plus en détail dans la suite de ce chapitre, en section 3.2.2.

3.1.3 Spécifications hardware

Dans la phase de *Hardware specifications* de la méthodologie (cf fig 3.2) nous caractérisons le matériel. Cette étape est composée d'une étape de ***Actor cost measurement*** où nous analysons l'exécution de l'application correspondante à la modélisation effectuée dans l'étape précédente de *Modelling*. La phase suivante de **partitionnement logiciel/matériel** (*Hw/Sw partitionning*) définit et sépare les tâches s'éxécutant sur le matériel de celles réalisées sur la partie logicielle. A ce niveau, un rebouclage de la méthodologie permet de redéfinir un nouveau partitionnement dans le cas où le prototype précédent ne satisfait pas les contraintes. Ainsi, on peut déporter de nouvelles fonctions logicielles vers le matériel afin d'accélérer les traitements. Le choix de partitionnement est guidé par la modélisation de l'architecture (effectuée dans l'étape de *Modelling*), des contraintes et des spécifications hardware. Suite à ce partitionnement les phases d'***allocation*** mémoire et de définition des **interfaces SW/HW** finalisent cette étape de spécifications matérielles. L'une permet de gérer les ressources partagées entre les architectures (la mémoire) et l'autre permet de caractériser les frontières entre le bloc matériel et le logiciel.

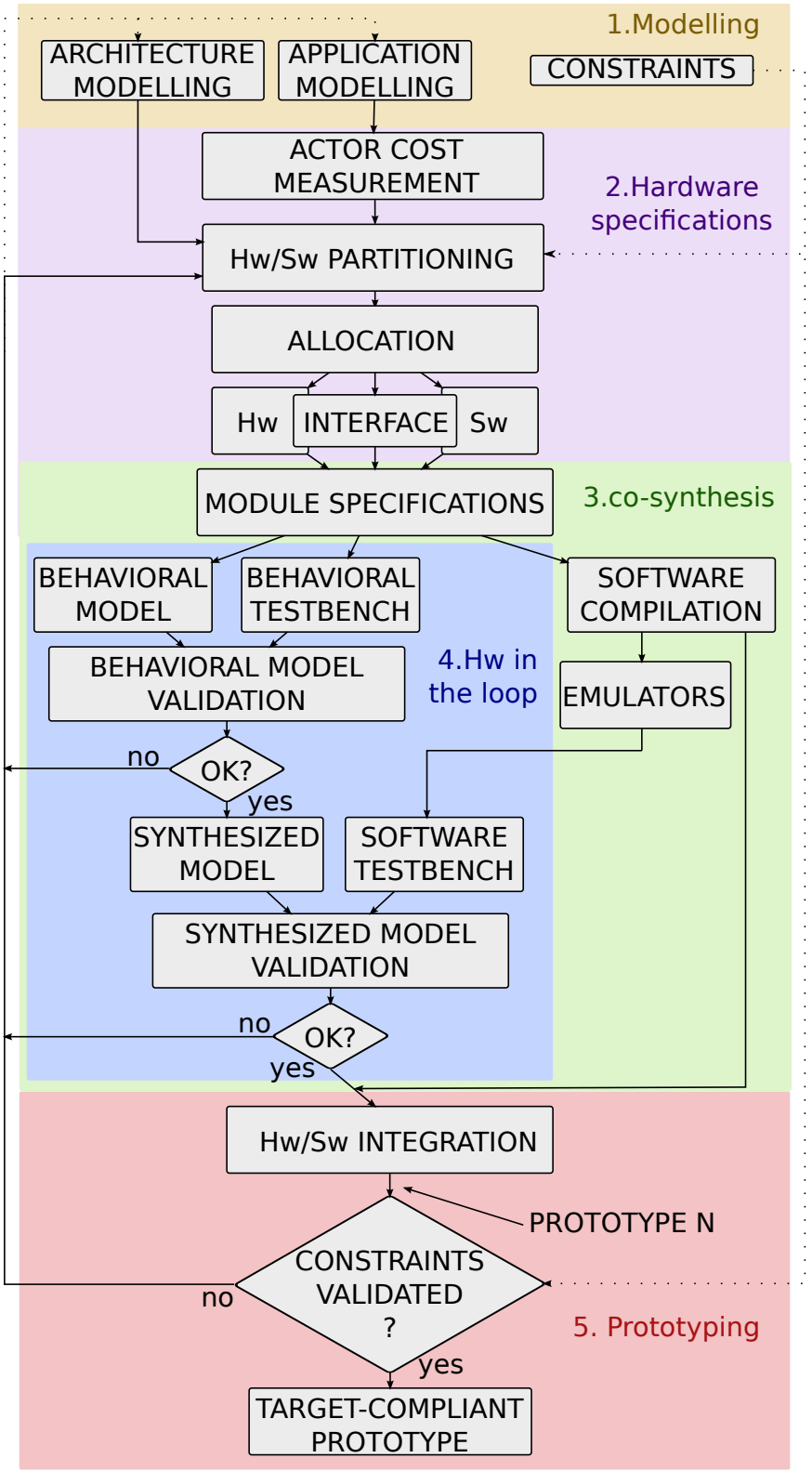

FIGURE 3.2 – Méthodologie de co-design HW/SW appliquée pour le prototypage d'un EKF-SLAM sur architecture hétérogène.

3.1.4 Co-synthèse et validation hardware in the loop

Les différentes fonctions à accélérer sont implantées en matériel pendant la phase de *Co-synthesis* de la méthodologie (cf. figure 3.2). Cette étape est divisée en deux parties, l'une concerne le développement matériel comprenant une phase de conception *Hw in the loop*. L'autre concerne l'élaboration du logiciel.

Avant la conception matérielle, une analyse théorique méticuleuse est effectuée avant la phase de programmation HDL [1]. Cette analyse permet d'évaluer si le transfert de la fonction logicielle vers un accélérateur matériel ne change pas le comportement du bloc fonctionnel correspondant (pour les mêmes entrées, les sorties des fonctions logicielles et matérielles sont identiques). Elle permet, de plus, de vérifier les performances du bloc matériel : une fréquence de traitement élevée et un temps d'exécution réduit (latence faible). Cette phase préliminaire est facilitée par l'utilisation de langages haut niveau comme SystemC permettant de faire une vérification rapide du fonctionnement de l'architecture (cf. section 2.2.2.2).

Dans le diagramme de la méthodologie de co-design (cf. figure 3.2), les blocs :

- ***Behavioral model*** et ***Behavioral testbench*** sont des architectures HDL conformes aux spécifications établies dans ***Module specifications***. ***Behavioral model*** est l'architecture HDL qui remplit la fonction que l'on souhaite accélérer et ***Behavioral testbench*** est un module HDL permettant de la tester. ***Behavioral model validation*** est la phase qui valide le ***Behavioral model***, grâce aux stimuli de ***Behavioral testbench*** ;
- ***Synthesized model*** est un modèle répondant aux contraintes de temps d'éxécution, de latence et de consommation de ressources FPGA. Dans l'étape de ***Synthesized model validation***, nous migrons le ***Software testbench*** vers une application bare-metal standalone [2] avec des fonctions de wrapper [3] du *Synthesized model*. Si le modèle passe avec succès les validations par couverture de code [4], nous obtenons une propriété intellectuelle (IP) pleinement opérationnelle.

La validation Hardware In the Loop (HIL) est utilisée pour valider les accélérateurs de la manière la plus fidèle et la plus proche de leur implantation finale dans l'application. Dans le cadre de développement sur FPGA, cette phase permet de valider la fonction matérielle seule sans avoir besoin de gérer les intéractions/synchronisations avec le reste de l'architecture. Les entrées du bloc testé sont activées par des stimuli (générés par un bloc matériel ou logiciel). Enfin, les résultats sont évalués afin de valider la conformité du comportement de l'architecture.

Du point de vue développement logiciel et pendant la phase de compilation logicielle (***Software compilation***), la fonction devant être accélérée en matériel est ré-écrite afin d'émuler son comportement en matériel (***emulators***). Elle est ensuite traduite en bloc matériel afin de la tester dans la phase de *hardware in the loop*.

C'est aussi durant cette phase que nous développons l'interface entre le bloc matériel et le reste du système (synchronisations, chargement et stockage des données). Dans notre prototypage, nous nous servons des interfaces de communication Xillybus® (distribuée par Xillinux [5]) afin de transmettre les données entre la logique et le microprocesseur du ZynQ®. Pour pouvoir les utiliser, il faut instancier les IPs suivants sur FPGA :

1. Hardware Description Language
2. La fonction est exécutée sur le processeur sans Operating System.
3. Encapsulation des modules et gestion de l'interface entre le ***Software testbench*** et le ***Synthesized model***.
4. La couverture de code est une mesure permettant de mesurer la qualité des tests effectués.
5. http ://xillybus.com/xillinux/

— *Xilly_vga*, fonction permettant l'affichage de Xillinux ;
— *Xillybus_lite*, permettant de réservé de la mémoire BRAM du FPGA (mapping mémoire) pour y avoir accès depuis le logiciel en écriture et en lecture ;
— *Xillybus*, IP autorisant l'interfaçage streaming entre la distribution Xillinux et la logique FPGA.

Au fur et à mesure de notre développement itératif de prototype, nous avons intégré un par un des accélérateurs matériels. A chaque nouveau partitionnement de l'application, nous avons modifié les frontières entre le logiciel et le matériel. Deux interfaces de communication sont disponibles sur *Xillybus* pour transmettre les données entre la partie matérielle et logicielle : les FIFOs et les registres mémoires. Nous avons choisi de transmettre les données avec des :
— FIFOs matérielles si le nombre de données à transmettre est variable (par exemple, les points d'intérêt détectés par le bloc matériel *features detection*) ou si les données sont envoyées en streaming (l'envoi d'une image pixel par pixel à la logique FPGA) ;
— Registres mémoires (stockés en BRAM) si le nombre de données à transmettre est connu (par exemple les points d'intérêt à corréler dans une image ou la matrice P).

Dans les deux cas, nous nous assurons d'une bonne synchronisation entre les tâches :
— les FIFOs sont bloquantes [6] et nous contrôlons leurs taux de remplissage ;
— et nous créons des registres stockés directement en mémoire permettant de connaître l'état du bloc matériel (en attente, en cours de traitement, fin de traitement), dans le cas d'une interface par mémoire(s) partagée(s).

A la fin de ce co-développement, nous obtenons un IP standalone fonctionnel. Il reste à l'implanter dans l'application en gérant les itérations et synchronisations entre les frontières SW/HW.

3.1.5 Intégration et validation

La dernière étape de Prototyping de cette méthodologie de co-design est la phase d'**intégration HW/SW**. Dans cette dernière, le design partitionné est intégré puis validé dans dans la chaîne de traitement EKF-SLAM.

Une fois le bloc matériel intégré, le comportement de l'application est à nouveau évalué et validé. A ce stade, une **mesure de performances** est effectuée afin de voir si les objectifs sont atteints. Si toutes les contraintes sont satisfaites, le prototype est validé.

Dans le cas contraire, le processus de la méthodologie est réitéré et on effectue une nouvelle optimisation basée sur les modèles (*Modelling*, cf figure 3.2) ou un nouveau partitionnement matériel/logiciel (*HW/SW partitioning*, cf figure 3.2). Pour déterminer les nouvelles fonctions à accélérer, un nouveau profilage [7] de la chaîne est effectué afin d'identifier les fonctions qui engendrent des goulots d'étranglement pour ensuite essayer de les accélérer.

Chaque itération de la méthodologie de co-design aboutit à une nouvelle fonction accélérée, une nouvelle IP standalone réutilisable.

6. Les fonctions matérielles ou logicielles attendant des données via la FIFO sont mises en pause tant que ces données ne sont pas entièrement disponibles en FIFO.
7. Le profilage d'un logiciel permet d'analyser son exécution. On contrôle la liste des fonctions appelées, leur temps d'exécution et l'utilisation processeur et l'utilisation mémoire.

3.2 Modélisation & spécifications hardware

Dans cette section, nous allons présenter les deux premières étapes de la méthodologie de co-design, la modélisation (*Modelling*) et la spécification matérielle (*Hardware specifications*, cf. figure 3.3).

Dans un premier temps, nous allons détailler la phase de *Modelling* dans laquelle nous définirons les contraintes de notre SLAM puis nous optimiserons le système en faissant une modélisation au niveau structurel. Pour faire cette optimisation, nous utiliserons une description flux de données (ou SDF [8], cf. 2.3.2.1) qui décrit l'application comme un réseau d'acteurs qui s'échangent des jetons de données au travers des arcs qui les connectent (cf. section 2.3.2). Puis nous optimiserons ce modèle en extrayant du parallélisme entre les acteurs [9]. Ce travail nous permettra par la suite d'éxécuter plus de tâches en parallèle pour réduire le temps de traitement. Ces phases de *Modelling* ont été initiées par les travaux de [Piat et al., 2013] où les auteurs présentent l'optimisation du EKF SLAM par les modèles SDF.

Puis, dans la phase de *Hardware specifications*, nous mesurerons le temps d'exécution des acteurs (phase *actor cost measurement*) de notre application en fonction des résultats de cette optimisation par flux de données.

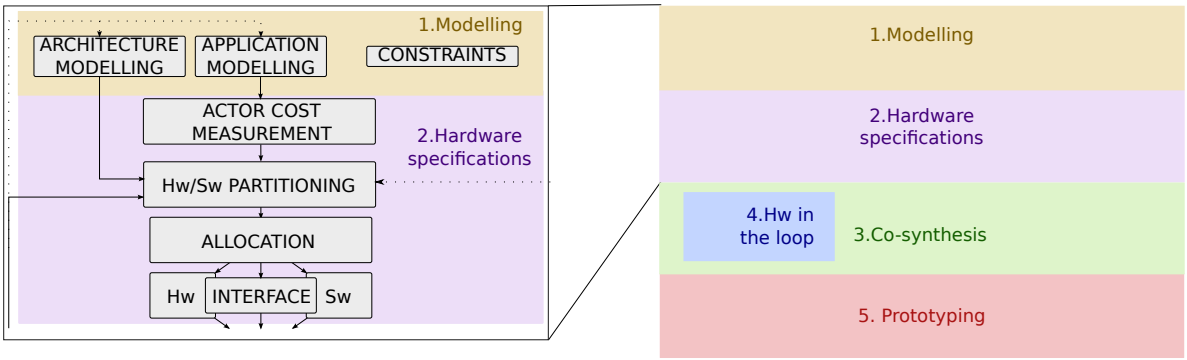

FIGURE 3.3 – Méthodologie de co-design mise en oeuvre : étape 1 de modélisation et étape 2 de spécifications hardware.

3.2.1 Spécifications des contraintes

La première phase dans l'étape de modélisation (*Modelling*) est de définir les **contraintes** de notre système (*Constraints*, cf fig 3.2). Les contraintes que nous souhaitons atteindre au terme de notre développement sont :

— Temps de traitement : nous souhaitons que notre application soit capable de traiter un EKF-SLAM en temps réel. Nous avons défini ce temps réel à 30Hz (cf. paragraphe 1.3) ;
— Latence : nos travaux visent à localiser un robot ou un véhicule autonome. Il est donc intéressant de minimiser la latence du système pour fournir une mesure la plus actualisée possible. Nous fixons une latence de maximum 1 image ;
— Consommation : nos travaux s'inscrivent dans le contexte des systèmes embarqués ADAS. Nous souhaitons obtenir un prototype consommant moins de 5W ;

8. Synchronous DataFlow.
9. Un acteur représente une entité de calcul de l'application (cf. section 2.3.2).

— Encombrement : le système final doit pouvoir être implanté dans le boitier d'une smart-caméra. Nous avons donc fixé la contrainte d'encombrement physique du produit fini (sans la caméra) à 5cm×5cm.

Ces contraintes permettront de valider ou non le prototype obtenu à chaque itération de la méthodologie de co-design.

3.2.2 Modélisation SDF du SLAM

Une fois les contraintes identifiées, nous pouvons passer à la phase de **modélisation** (étape *Modelling*).

La modélisation flux de données, présentée dans [Lee and al, 1987] et [Lee and Messerschmitt, 1987], se base sur la description fonctionnelle de l'application (cf. figure 3.4). Cette description du système permet de faire une étude anticipée, avant d'avoir fait des choix dans le développement. Elle intervient avant d'avoir sélectionné la plateforme qui accueillera l'architecture et avant d'avoir commencé les développements (choix du partitionnement, accélération de fonctions sur architecture dédiée, etc). Elle se déroule en deux phases (cf. section 2.3.2.1) : modélisation/étude du système pour un unique coeur de traitement, puis, optimisation de ce modèle pour une architecture multi-processeur (sans toutefois choisir le type d'unité de traitement cible). Notre application EKF-SLAM peut se décrire de façon séquentiel à l'aide du pseudo-code suivant :

```
1:  for n = 0; ; n++ do
2:      image = acquire_frame()
3:      predict(slam, robot_model);
4:      i = 0;
5:      j = 0;
6:      lmk_list = select_lmks();
7:      correl_list = correl_lmks(lmk_list, frame);
8:      for i = 0; i < nb_correl and j < nb_correct; i++ do
9:          if score(correl_list[i]) > correl_threshold then
10:             correct_slam(correl_list[i]);
11:             j++;
12:         end if
13:     end for
14:     feature_list = detect_features(frame);
15:     j = 0;
16:     for i = 0; i < grid_size and j < nb_init; i++ do
17:         if init_lmk(feature_list[i]) then
18:             j++;
19:         end if
20:     end for
21: end for
```

Avec les variables :
— *lmk_list* : une liste d'amers ;
— *correl_list* : une liste de d'observations dans l'image pour les amers de la carte ;
— *feature_list* : une liste de points d'intérêt détectés dans l'image.

Et les fonctions :
— *acquire_frame*() : une fonction permettant l'acquisition des images.

— *predict*() : une fonction effectuant l'étape de prédiction du filtre EKF ;
— *select_lmks*() : une fonction sélectionnant/retournant une liste d'amers devant être observés dans l'image ;
— *correl_lmks*() : une fonction mettant en correspondance une liste d'amers provenant de l'image $n-1$, dans l'image n ;
— *correct_slam*() : une fonction effectuant l'étape de correction à l'aide du filtre EKF en se basant sur l'observation des amers effectuée par la fonction *correl_lmks*() ;
— *detect_features*() : une fonction renvoyant une liste de points d'intérêt détectés dans l'image ;
— *init_lmk*() : une fonction initialisant les amers dans la carte du EKF-SLAM à partir des points d'intérêt détectés dans l'image n. De nouveaux amers sont intégrés dans la carte au moment de l'initialisation du SLAM et pour remplacer des amers dont les niveaux de confiance des observations sont devenus trop faibles (en dessous d'un seuil de confiance fixé, les amers sont enlevés de la carte).

La figure 3.4 présente la description fonctionnelle de notre application. Voici le tableau de correspondance des noms entre les blocs de la description structurelle et les fonctions du pseudo-code :

acquire_frame()	*camera*
predict()	*prediction*
select_lmks()	*landmarks selection*
correl_lmks()	*landmarks correlation*
correct_slam()	*correction*
detect_features()	*features detection*
init_lmk()	*landmarks initialization*

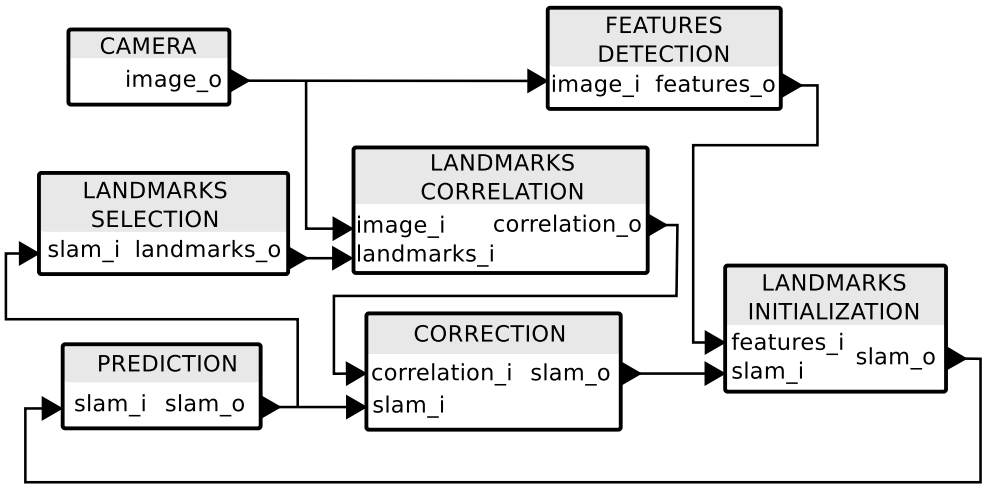

FIGURE 3.4 – Graphe fonctionnel de l'application vision EKF SLAM.

La modélisation SDF est applicable à tout niveau de granularité d'un système (aussi bien dans une fonction bien précise que sur la représentation générale de l'application). Cependant, afin d'optimiser au mieux une application, il est nécessaire de la modéliser d'abord au niveau

fonctionnel. En étudiant les données échangées dans le système (pseudo-code et description structurelle), nous pouvons modéliser notre chaîne 3D EKF-SLAM en un graphe SDF, présenté dans la figure 3.5. On associe un jeton à une donnée consommée/produite par un acteur quelque soit son type (listes, images, matrices, etc).

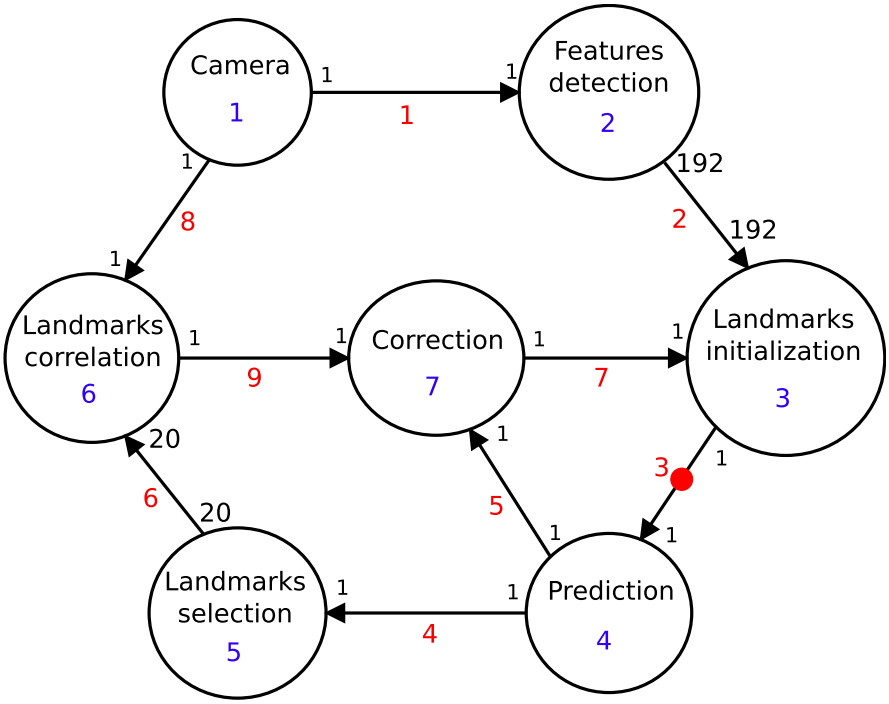

FIGURE 3.5 – Modèle SDF de l'application vision EKF SLAM.

La figure 3.6 montre le modèle SDF de notre application SLAM avant optimisation. Le jeton de données initial (symbolisé en rouge) permet le séquencement initial du réseau en forçant l'acteur *prediction* à s'exécuter en premier. Les boucles (1 et 2 dans la figure 3.6) autour des fonctions *init_lmk()* et *correct_slam()* de la description algorithmique (cf. pseudo-code) sont décrites dans le comportement des acteurs *correction* et *landmarks initialization* et non au niveau du réseau d'acteurs afin de simplifier la description.

La matrice topologique associée à ce graphe SDF est présentée en figure 3.7 (où les numéros de noeuds [10] et arcs [11] se réfèrent à la modélisation SDF, figure 3.5). On retrouve les données de cette matrice topologique dans la figure 3.5. Cette dernière est la représentation de notre système SLAM sous la forme d'arcs et de noeuds avec le nombre de données échangées sur les arcs.

10. Un noeud est un acteur. Un acteur représente une entité de calcul de l'application (cf. section 2.3.2).
11. Un arc connecte deux acteurs. Un acteur produit des données sur l'arc que l'acteur/le noeud peut ensuite consommer (cf. section 2.3.2).

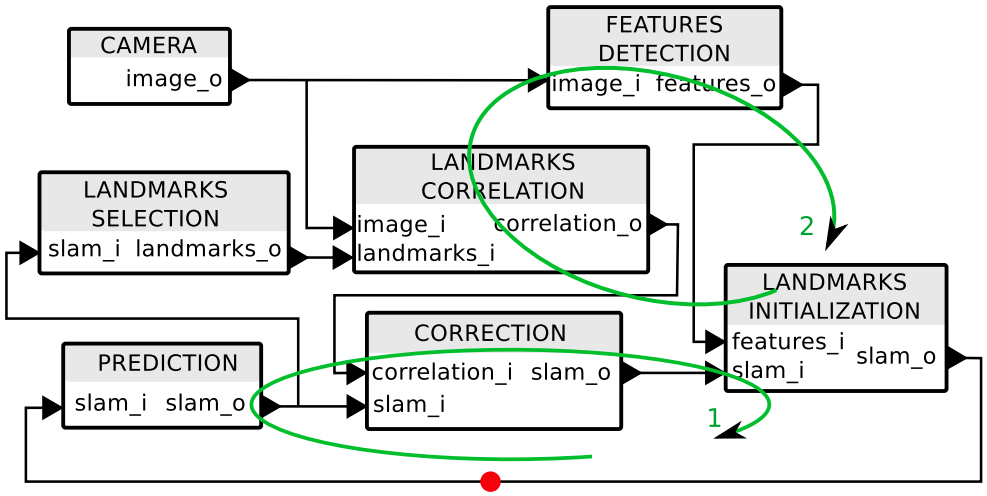

FIGURE 3.6 – Graphe HSDF de l'application vision EKF SLAM.

$$r = \begin{array}{c} arcs \setminus noeuds \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 192 & -192 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 20 & -20 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \end{array}$$

FIGURE 3.7 – Matrice topologique de la modélisation SDF de l'application 3D EKF SLAM (cf. figure 3.5).

Pour un processeur mono-coeur (symbolisé par : Ψ), les tâches sont exécutées comme ceci (cf. figure 3.5) :

$$\Psi_1 = \{4_{prediction}, 5_{landmarks\ selection}, 1_{camera}, 2_{features\ detection}, 6_{landmarks\ correlation}, 7_{correction}, 3_{landmarks\ initialization}\}$$

L'ordonancement du réseau obtenu grâce aux techniques de scheduling SDF montre que l'application ne présente qu'un degré de parallélisme. L'acteur *features_detection* peut s'exécuter indépendamment du reste du réseau (le point de rencontre étant l'acteur *landmarks initialization*). Ceci est dû à l'absence de dépendance de données entre l'acteur *features_detection*

et le reste du réseau [12]. On peut distribuer les tâches sur deux processeurs comme dans le système suivant :

$$\begin{cases} \Psi_1 = \{1_{camera}, 2_{features\ detection}\} \\ \Psi_2 = \{4_{prediction}, 5_{landmarks\ selection}, 6_{landmarks\ correlation}, 7_{correction}, 3_{landmarks\ initialization}\} \end{cases} \tag{3.1}$$

Dans ce système, nous appelons Ψ un processeur de traitement. La notation "1_{camera}", renseigne le numéro et le nom de l'acteur (cf. figure 3.5)

Afin de concevoir une application pouvant satisfaire nos contraintes, il est nécessaire d'explorer le parallélisme potentiel de l'application pour envisager le portage vers une architecture multi opérateurs.

Le profilage unitaire des différents acteurs de l'application C-SLAM montre que les tâches *detect_features()*, *correl_lmks()* et *correct_slam()* représentent à ce stade $94,16\%$ du temps d'exécution (cf. tableau 3.11) d'une itération du réseau. Il faut donc explorer au niveau fonctionnel le parallélisme potentiel afin d'exécuter ces acteurs de manière concurrente qui monopolisent, en l'état, trop de temps processeur.

3.2.3 Réordonnancement des tâches de C-SLAM

Dans les applications modélisées en SDF, de nouveaux degrés de parallélisme peuvent apparaître au niveau structurel en appliquant des techniques de re-timing. Ces dernières consistent à ajouter des jetons initiaux sur les arcs du modèle SDF afin de modifier le niveau de pipeline de l'application. Cette modification n'affecte pas l'aspect structurel du modèle (la façon dont les noeuds sont connectés). Il en va de même pour la latence de l'ordonnancement. Cette opération de réordonnancement n'autorise que la modification du pipeline dans le but de gagner des degrés de parallélisme par rapport à celui initialement possible. De cette manière, nous optimisons la cadence de l'application sans modifier sa latence.

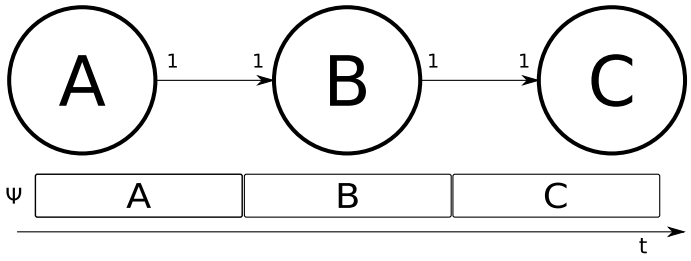

FIGURE 3.8 – Partie supérieure de la figure : Exemple d'un graphe flux de données (SDF) modélisant trois acteurs (A,B,C). Partie inférieure : Planification du traitement de ces tâches sur une unité de traitement (Ψ).

La figure 3.8, illustre un exemple simple de réordonnancement avec 3 noeuds/acteurs (A,B,C). Elle représente l'ordonnancement initial d'un réseau SDF. Le flux de données peut être modifié en ajoutant un jeton initial (représenté par un point rouge sur un arc de la figure 3.9) afin de modifier l'ordre d'ordonnancement et ainsi générer un degré de parallèlisme,

12. Les boucles d'acteurs travaillent sur des données indépendantes.

comme le montre la figure 3.9 (partie inférieure). Ajouter un simple jeton de délai sur l'arc entre les acteurs A et B revient à autoriser l'acteur B à pouvoir s'exécuter avant que A n'ait lui même produit de jetons. Une itération du réseau permet donc de revenir à l'état initial après l'exécution de A.

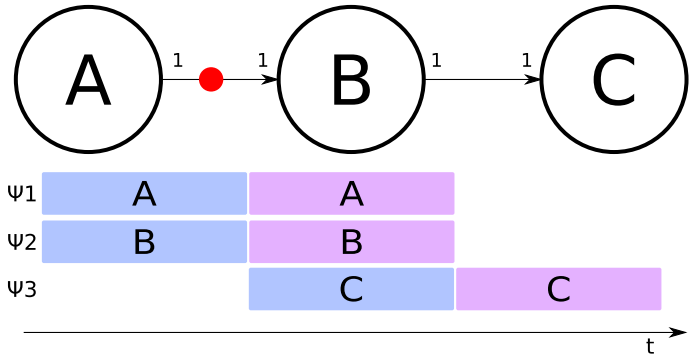

FIGURE 3.9 – Partie supérieure de la figure : Exemple d'un graphe flux de données (SDF) avec réordonnancement des tâches (jeton de delai symbolisé par un point rouge). Partie inférieure de la figure : planification du traitement de ces tâches sur multi-processeur (Ψ_n).

Ce réordonnancement temporel expose un niveau de pipeline qui pourra être exploité lors du partitionnement sur une architecture multi-processeur (Ψ_n). Ce ré-ordonnancement, bien que n'affectant pas le niveau structurel, doit être pris en compte dans le comportement de l'application.

Dans notre application 3D EKF-SLAM, le re-timing consiste en l'ajout de jetons de délai sur les arcs en sortie des acteurs (cf. figure 3.10) : *features detection* (1), *landmarks correlation* (2) et *landmarks initialization* (3). Le placement de ces jetons est choisi de façon à dissocier les 2 boucles de traitement (boucles 1 [13] et 2 [14] sur la figure 3.6) et ainsi pouvoir accélérer le traitement des tâches *detect_features()*, *correl_lmks()* et *correct_slam()* (représentant à elles trois 94, 16% du temps total de traitement). Cette modification résulte en un ordonnancement disposant de nouveaux degrés de parallèlisme.

Après réordonnancement, l'application expose 3 degrés de parallélisme car la corrélation d'amers (*landmarks correlation*) et la détection de points d'intérêt (*feature detection*) peuvent être maintenant exécutées en parallèle de l'étape de la partie filtrage (*prediction*, *correction*, *landmarks initialization* et *landmarks selection*).

On peut désormais optimiser les calculs en répartissant les tâches sur trois processeurs, comme dans le séquencement suivant (où les numéros de noeuds et arcs se réfèrent à la figure 3.5) :

$$\begin{cases} \Psi_1 = \{4_{prediction}, 7_{correction}, 3_{landmarks\ initialization}, 5_{landmarks\ selection}\} \\ \Psi_2 = \{1_{camera}, 2_{features\ detection}\} \\ \Psi_3 = \{6_{landmarks\ correlation}\} \end{cases} \tag{3.2}$$

Ce parallélisme permet de créer un partitionnement bien distinct entre la tâche de filtrage

13. *features detection*, *landmarks initialization*, *correction* et *landmarks correlation*.
14. *prediction*, *correction* et *landmarks initialization*.

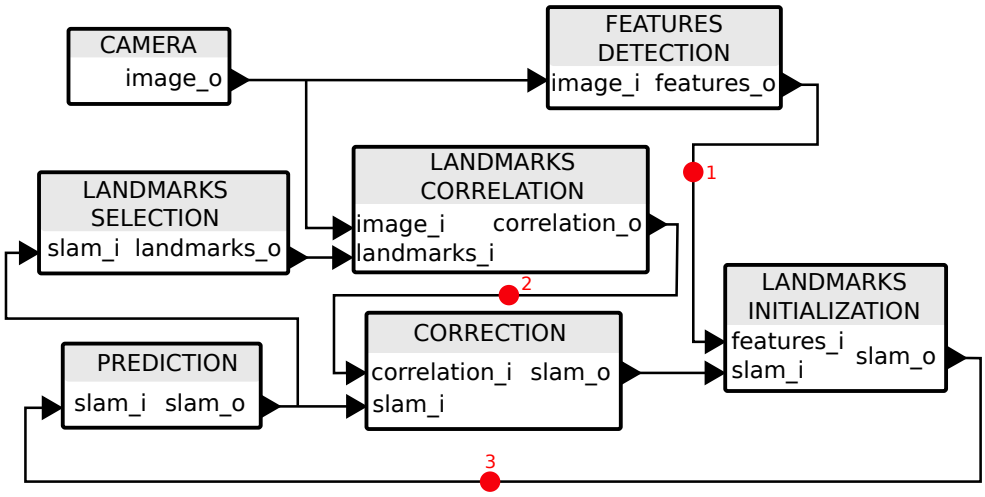

FIGURE 3.10 – Modèle SDF réordonnancée de l'application EKF-SLAM

(*prediction*, *correction*, *landmarks initialization* et *landmarks selection*) et la tâche de vision (*features detection* et *landmarks correlation*).

Ce parallèlisme permet l'exécution en parallèle des opérations de :
— **vision** (partie front-end de l'application) traitant des données bas niveau (les pixels dans l'image) ;
— et de **filtrage** (partie back-end) qui exécute des traitements de haut niveau (points d'intérêt, mise à jour de la matrice d'état du robot, de la carte, etc).

Cependant, ce réordonnancement pourrait affecter les performances du SLAM puisque l'étape de *correlation* pour l'itération N du réseau d'acteurs travaille sur les données d'entrée générées à l'instant $N-1$ (cf. figure 3.11(B)). Ce retard dans l'observation impacte la capacité de l'algorithme à calculer une position cohérente lorsque un nombre important d'amers ne sont plus observés. En effet pour un un amer échouant à être observé à l'itération N, la composante filtrage de l'application n'en sera notifiée qu'à l'itération $N+1$. Pour chaque amer disparaissant, le filtre initialise un nouvel amer, dans notre cas cette initialisation ne se fera qu'à l'itération $N+1$ pour une première observation de ce nouvel amer à l'itération $N+2$. De ce fait, la phase d'observation du filtre de Kalman bénéficiera de moins de données d'observation pendant deux itérations. Cette modification peut surtout avoir un impact sur les phases de mouvement rapide où la perte d'observations est importante. Cependant il est possible de limité cet effet grâce une cadence de traitement d'images élevée et constante pour limiter les changements de pose entre deux images.

La figure 3.11 illustre une planification de notre application sur trois unités de traitement (notées PE pour Processing Element). Dans 3.11(A) on trouve la représentation séquentielle de C-SLAM correspondant à l'ordonnancement de départ (figure 3.6). Dans 3.11(B) on retrouve la même application avec le réordonnancement temporel après l'optimisation au niveau

structurel [15]. En effectuant cette séparation distincte entre la partie vision et la partie filtrage, même sans prendre en compte le temps d'exécution de chaque tâche (ce travail sera fait dans la section suivante 3.2.4) nous pouvons déjà entrevoir le gain en performance et donc le potentiel temps réel de notre application sur architecture embarquée avec des ressources de calcul limitées.

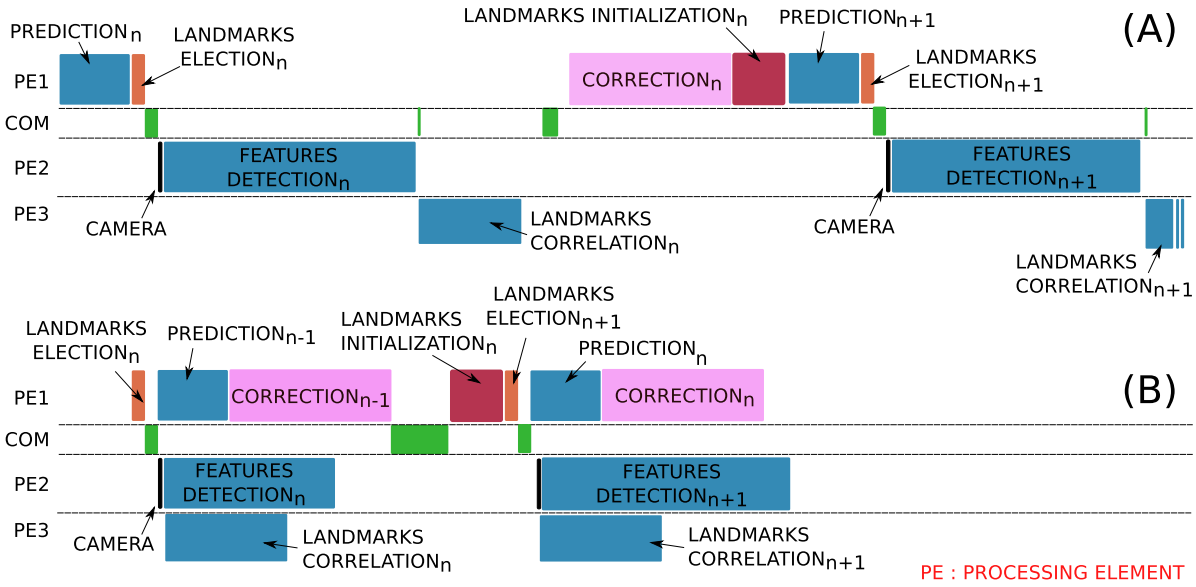

FIGURE 3.11 – Répartition des tâches de l'application SLAM avant et après réordonnancement.

Après avoir ainsi optimisé notre application au niveau structurel (et obtenu de nouveaux degrés de parallélisme), nous allons, dans la section 3.2.4, passer à l'étape *2.hardware_specifications* de la méthodologie de co-design.

3.2.4 Mise à jour des spécifications

En analysant les différentes tâches que nous pouvons exécuter en parallèle, nous allons, dans cette section, définir l'architecture qui accueillera notre application EKF-SLAM.

La méthodologie proposée (cf. 2.3) permet de remettre en cause la plateforme d'intégration à tout moment du développement. Nous avons vu précédemment (cf. section 2.2.1) que l'analyse du code C-SLAM nous a permis de statuer sur un un type d'architecture mixte (FPGA-CPU) dès le début de l'application de la méthodologie de co-design. Nous avons identifié deux types d'opérations :

— Le **front-end** correspond aux traitements de la vision (détection de points d'intérêt et corrélation des amers) ;
— Le **back-end** exécute la tâche de filtrage de Kalman (sélection et correction des amers, prédiction et initialisation des amers).

La partie **front-end** est composée d'opérations de base (comparaisons, additions, etc) avec un débit de données pixel élevé (9,2 MB/s pour 30 images/s et une résolution 640x480 pixels). La partie **back-end** doit, elle, traiter des données de haut niveau à plus faible débit

15. La taille des barres est à titre indicatif

et doit exécuter des opérations principalement composées d'algèbre linéaire. Une architecture hétérogène est donc tout à fait adaptée à notre application. En effet, un opérateur de type CPU est adapté aux calculs constitués d'opérations complexes (calculs dans des espaces vectoriels et transformations linéaires) et bas débit du filtrage. Il peut être secondé par un autre type d'architecture de traitement qui prendrait en charge les traitements d'images. Si le choix d'une architecture CPU est évident, il est en revanche intéressant de comparer les possibilités d'architectures adaptées aux traitements de la partie front-end. Comme présenté dans 2.2.1, les GPU et FPGA sont deux architectures adaptées à ce type de calcul haut débit. Mais, après avoir étudié les tâches à accélérer, nous avons orienté notre choix vers le FPGA afin d'optimiser au maximum les fonctions de vision pour garantir les contraintes de débit de notre application (30 images/s) et de plus de minimiser la consommation.

Le FPGA nous permettra :

— d'exécuter des fonctions de traitement d'images avec une latence minimale et un temps d'exécution maximisé ;
— à la fin du développement, de prendre en charge toutes les fonctions de vision (front-end) pour décharger le CPU des opérations simples mais exécutées à très haut débit ;
— de connecter en fin de développement la caméra directement sur la logique afin de supprimer complètement le transfert des images du CPU vers la logique et ainsi de supprimer la tâche *camera* (cf. tableau 3.1).

Après avoir retenu l'architecture CPU-FPGA, nous avons exploré les différentes solutions sur étagère disponibles sur le marché. Nous avons orienté cette recherche en fonction des ressources disponibles sur le FPGA, de la fréquence et du type du CPU ou encore des I/O et des mémoires disponibles sur la plateforme de développement.

Nous avons fait notre choix en étudiant le profiling du SLAM, en analysant les fonctions qui peuvent être accélérées et en modélisant le flux de données entre les opérateurs pour évaluer le volume de données qui transite entre les deux unités de traitement. Notre choix de FPGA s'est porté sur une architecture hétérogène proposant une partie FPGA et une partie CPU, le ZynQ® de la société Xilinx®. Cette nouvelle architecture a révolutionné le monde du FPGA en offrant des taux de transferts entre la partie logique et le processeur très élevés (4 BUS AXI disponibles). Cette bande passante est possible grâce à la proximité des deux architectures réunies sur une seule et même puce électronique. Nous avons ensuite cherché les différentes plates-formes équipées avec ce SoC. La plateforme zedboard® [16] de la société Avnet® [17] est un bon compromis pour prototyper une application car elle intègre :

— un processeur multicoeur ARM®v7-Cortex A9 (667MHz) économe en énergie (<5W) ;
— un FPGA ZynQ®-7020 permettant un fort degré de parallélisme ;
— 512MB de RAM DDR3.

Nous l'avons choisie pour les raisons suivantes :

— La carte était disponible au début de nos travaux (fin 2012) ;
— La quantité de ressources de la partie logique était approriée aux accélérations que nous avions envisagées (85K de cellules logiques ; 53K de LUT ; 106K de flip-flop et 140 BRAM de 36Kb) ;
— Le CPU-ARM9 NEON de la plateforme supporte les instructions SIMD ;

16. http ://zedboard.org/
17. http ://www.avnet.com/en-us/Pages/default.aspx

— Le CPU-ARM9 est équipé de 2 coeurs permettant d'exécuter 2 threads en parallèle ;
— La communauté d'utilisateurs est active et conséquente ;
— Le coût d'achat de cette carte est réduit (environ 500€ au moment du début des travaux) ;
— La distribution Xillinux® (distribuée par Xillybus® [18]) supporte le FPGA ZynQ®. Cette dernière propose des interfaces entre le FPGA et le CPU de type memory-mapped (des BRAMs sont accessibles en lecture et écriture du côté CPU et du côté FPGA) et streaming (via des FIFOs). Cette distribution nous décharge d'un effort d'ingénierie en proposant les drivers et une mise en oeuvre simplifiée (Xillybus propose un assistant de haut niveau implantant la communication FPGA-CPU). Elle nous permet de prototyper rapidement.

La figure 3.12 montre l'architecture ZynQ® avec les fonctions du EKF SLAM (E) exécutées sur Xillinux (D) par le processeur ARM cortex A9 (B). On peut voir les blocs matériels Xillybus (A) décris en section 3.1.4 connectés aux processeur les BUS AXIs (C) disponibles dans le ZynQ®.

Le tableau 3.1, présente les résultats d'une exécution monothread du C-SLAM sur ZedBoard. Ces résultats ont été obtenus grâce à gprof [Spivey, 2003], un logiciel de profiling intrusif (il modifie légèrement l'exécution du programme) :

Taches C-SLAM	Temps d'occupation CPU [%]
landmarks selection	0.28
prediction	0.89
correction	11.74
landmarks initialization	0.1
camera	4.62
landmarks correlation	4.55
features detection	77.89

Table 3.1 – Taux d'occupation CPU (mono-coeur) des tâches de C-SLAM.

Le tableau 3.1 est à mettre en relation avec la répartition des tâches proposées en figure 3.11(B). Les fonctions de *correction*, *landmarks_correlation* et *features_detection* sont les plus coûteuses. On peut donc, dès cette étape, prévoir de devoir les accélérer grâce au FPGA afin de satisfaire nos contraintes de temps réel.

Nous allons, dans la suite de ce mémoire, continuer d'appliquer la méthodologie de co-design 3.2. Le chapitre suivant explique le prototypage itératif et hardware-in-the-loop (HIL), le processus que nous avons suivi afin d'intégrer un EKF-SLAM visuel sur Zedboard® affichant des performances proches du réordonnancement optimum théorique (figure 3.11(B)).

Dans notre processus de développement, nous avons effectué quatre itérations de la méthodologie avant d'obtenir un prototype final répondant au cahier des charges. Elles sont listées en figure 3.13 et seront présentées dans les deux prochains chapitres de ce mémoire.

18. http ://xillybus.com/xillinux/

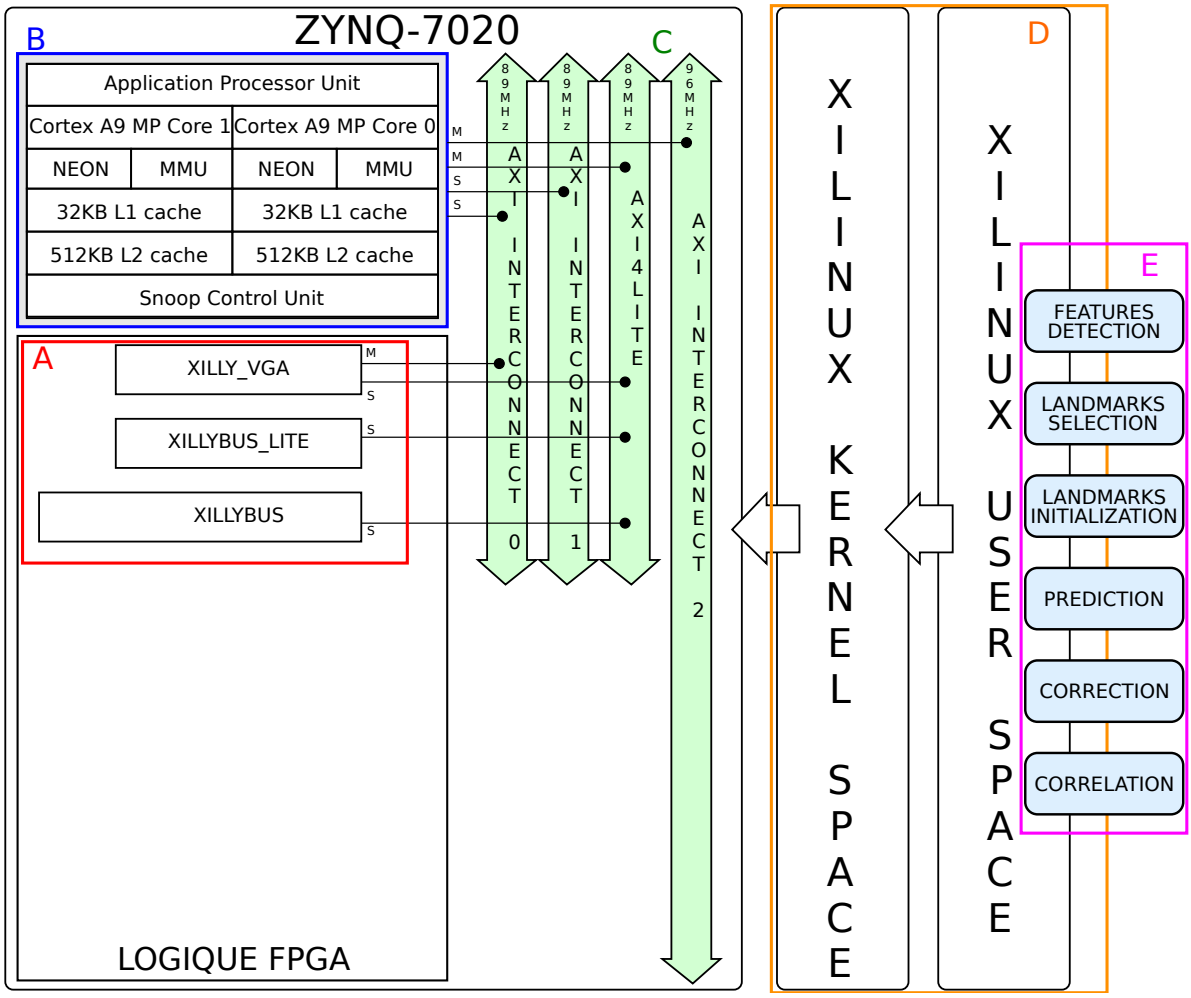

FIGURE 3.12 – Architecture de C-SLAM embarqué sur ZynQ®.

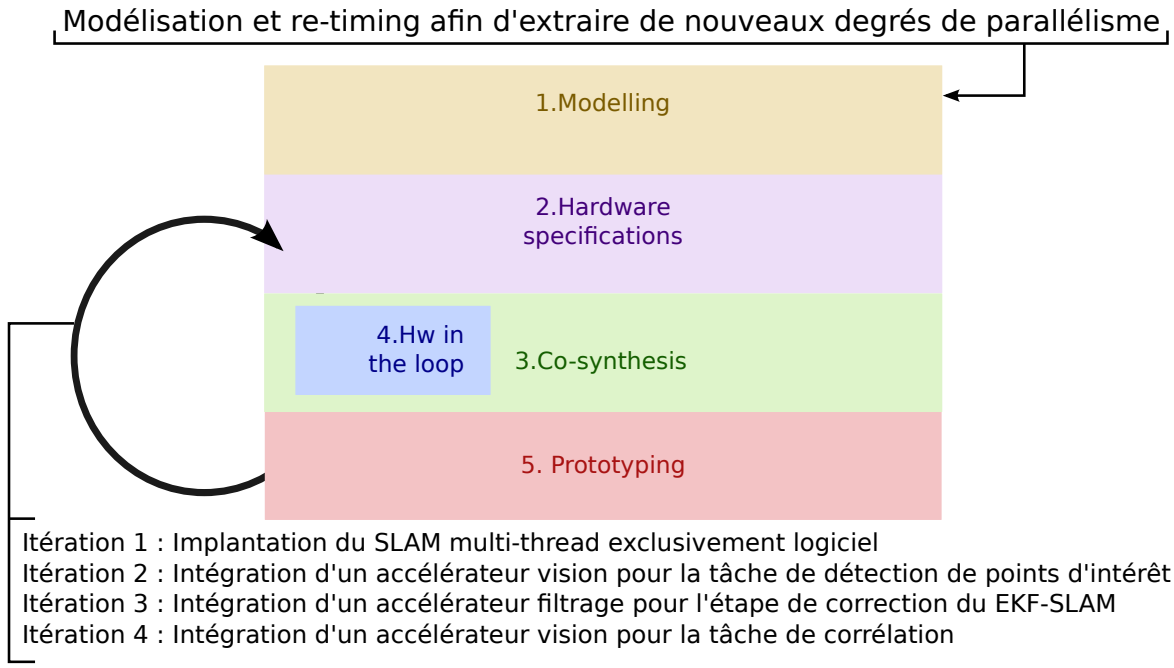

FIGURE 3.13 – Les différentes itérations de méthodologie de co-design effectuées avant d'obtenir un système qui satisfait les contraintes imposées.

Chapitre 4

Optimisations logicielles et Accélérateur back-end

Dans le chapitre 3, nous avons appliqué la méthodologie de co-design à notre application. Nous allons maintenant continuer d'appliquer cette méthodologie de co-design rappelée dans la figure 4.1. Dans un souci de cohérence des chapitres, nous avons choisi de regrouper dans ce chapitre 4 la **première** et la **troisième itération** de la méthodologie de co-design afin de réunir au sein d'un seul et même chapitre (chapitre 5) les travaux d'accélération de la partie front-end sur FPGA. Nous présenterons, dans un premier temps, le travail effectué lors de la **première** itération de la méthodologie de co-design (cf. figure 3.13), l'implantation et l'optimisation d'un SLAM purement logiciel grâce au multi-threading. Puis nous décrirons les travaux effectués pendant la **troisième** itération par Tertei, un autre doctorant du LAAS travaillant sur le projet, présentés dans [Tertei et al., 2014]. Nous exposerons la démarche qu'il a suivie lors de la troisième itération de la méthodologie de co-design. Dans cette dernière, Tertei a appliqué le flot de conception co-design (présenté dans le chapitre 3) à l'accélération de la tâche de correction (partie back-end) pour implanter un calculateur de produits de matrices sur FPGA. Nous présenterons sa démarche en suivant la méthodologie de co-design jusqu'au prototype qu'il a obtenu.

4.1 Optimisations au niveau logiciel

Cette première section est consacrée à la présentation des trois axes du travail d'optimisation du programme C-SLAM, à savoir : l'optimisation des calculs grâce aux instructions SMID - section 4.1.1, l'utilisation des deux coeurs de traitements de l'ARM par le multi-threading - section 4.1.2 et enfin, l'instanciation de FIFOs bloquantes permettentant la synchronisation des données qui naviguent entre les threads - section 4.1.3.

4.1.1 Optimisations NEON

Durant la première itération, nous avons travaillé sur l'implantation et l'optimisation logicielle. Nous sommes partis de l'algorithme de SLAM codé en langage C (C-SLAM - [Botero et al., 2012]) prévu pour fonctionner sur une station de travail standard (type processeur Intel® i5) que nous avons porté sur le CPU ARM du ZynQ®. Ce processeur étant moins performant que l'Intel® i5 (double coeur cortex A9 à 667MHz), nous avons constaté une baisse drastique des performances, passant de 25 images/s à 3 images/s. Le but de cette première itération de la méthodologie est de faire un travail au niveau du logiciel afin de l'optimiser au maximum pour ce type de CPU. Utilisant un ARM Cortex A9, nous bénéficions de la technologie NEON permettant une optimisation des multiplications de matrices et du traitement d'images. Cette

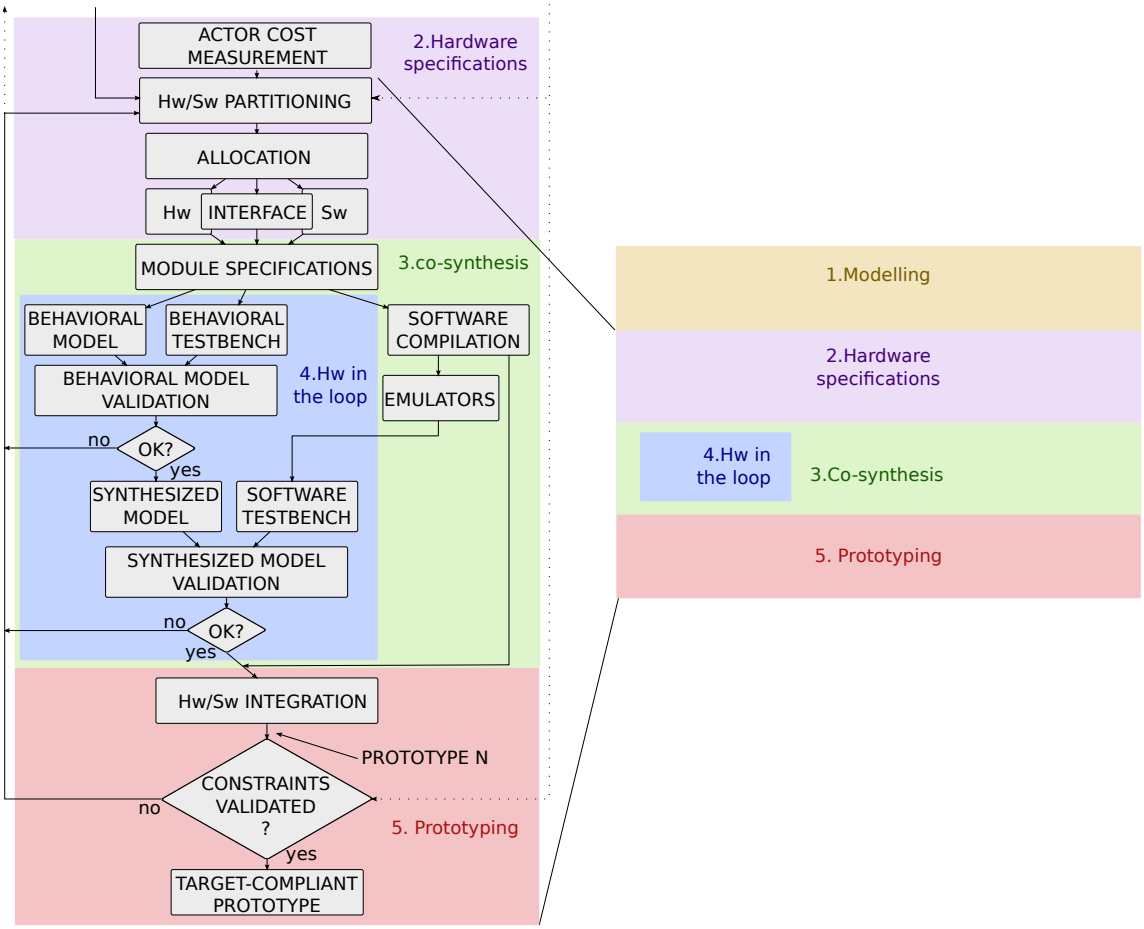

FIGURE 4.1 – Rappel de la méthodologie de co-design : Suite de l'étape 2 de spécifications hardware, 3. co-synthèse, 4 HIL et 5.Prototypage.

technologie basée sur les instruction SIMD [1] permet d'appliquer une seule et même instruction à plusieurs données en parallèle. Ce mode de fonctionnement est très efficace pour les données dont la structure est régulière [2] comme les calculs matriciels (très utilisés dans la partie filtrage du SLAM) ou encore le traitement d'images (les images sont des matrices de pixel). Grâce à cette technologie et en fonction des implantations, il est possible d'exécuter 4 opérations de 32 bits, 8 opérations 16 bits ou 16 opérations 8 bits simultanément.

4.1.2 Multi-threading

Afin de tirer parti des 2 coeurs disponibles sur l'ARM cortex A9, nous avons réparti les tâches de notre EKF-SLAM sur plusieurs threads. Ce partitionnement a été effectué en adéquation avec la modélisation précédemment obtenue, en section 3.2.4. Il est composé de trois threads (cf. figure 4.2) :

— Un thread **vision** exécutant toutes les tâches de la partie front-end du SLAM. Ce thread manipule des données bas niveau comme par exemple les pixels de l'image. Les tâches

1. Single Instruction, Multiple Data.
2. La quantité données traitées est invariante.

prises en charge sont : la récupération des données caméra (*camera*), la détection de points d'intérêt (*features detection*) et la *correlation* ;

— Un thread de **filtrage** réalisant les fonctions de la partie back-end du SLAM. Il manipule des données haut niveau comme, par exemple, des points d'intérêt ou encore le vecteur d'état du robot. Les tâches effectuées sont : la *correction*, l'initialisation des amers (*landmarks initialization*), la sélection des amers (*landmarks selection*) et la *prediction* ;

— Un thread **capteurs** prenant en charge les données capteurs. Ce thread n'est employé qu'en cas de configuration de SLAM en mode multi-capteur. Ces capteurs peuvent être : l'odométrie, une centrale inertielle (IMU) ou encore un GPS. Il permet de récupérer les données et de les synchroniser (à l'aide d'un horodatage [3] des données). Les données capteurs sont utilisées pour affiner la *prédiction*.

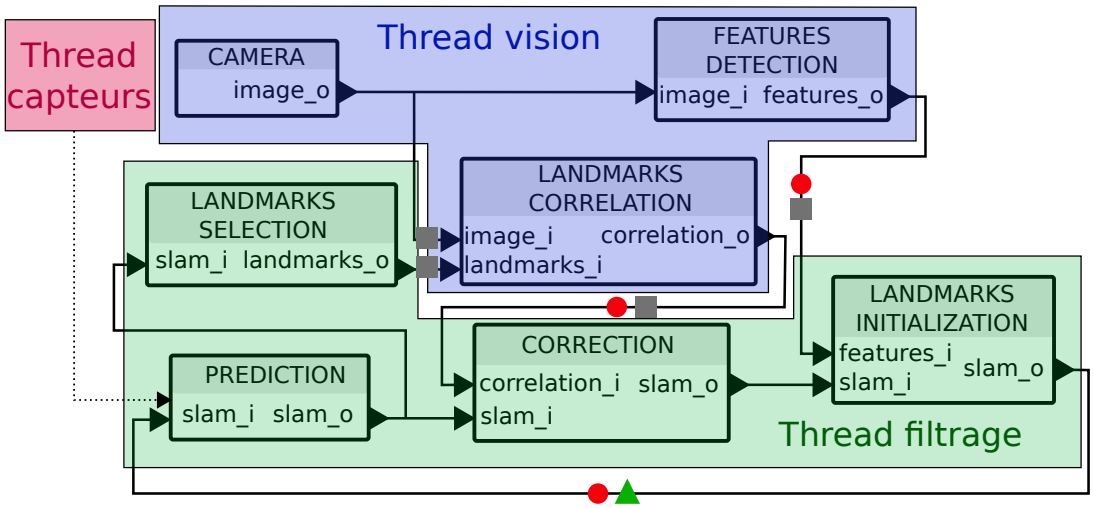

FIGURE 4.2 – Modèle SDF du SLAM avec jetons de délai pour permettre une optimisation grâce au re-timing et les différents threads logiciel. Les carrés gris et le triangle vert représentent des FIFOs logicielles (cf. section 4.1.3 pour plus de détails).

Les deux threads principaux de vision et de filtrage permettent une exécution des tâches de front-end et back-end sur les deux coeurs du processeur ARM. La figure 4.3 présente l'exécution de ces tâches en fonction du temps. Nous avons représenté les threads de filtrage (s'exécutant sur le thread T2) et de vision (sur le thread T1). Une ligne de communication (COM) illustre les transferts de données entre le thread T1 et T2.

4.1.3 Gestion des interfaces de communication

Dans le but de prévoir l'accélération matérielle des tâches de *features detection*, de *correction* et de *correlation*, nous avons mis en place une communication par FIFOs (logicielles [4]) entre les threads de vision et de filtrage. Elles sont utilisées en mode bloquant pour permettre d'assurer la synchronisation entre threads. Les threads sont mis en attente tant qu'ils n'ont pas **toutes** les données nécessaires pour exécuter un calcul. Ces interfaces par FIFOs sont symbolisées par des carrées gris sur la figure 4.2. De plus, afin de pouvoir exécuter notre SLAM comme

3. On associe une date et une heure aux données des capteurs : GPS, IMU et aux images.
4. Un espace mémoire RAM est alloué à ces FIFOs.

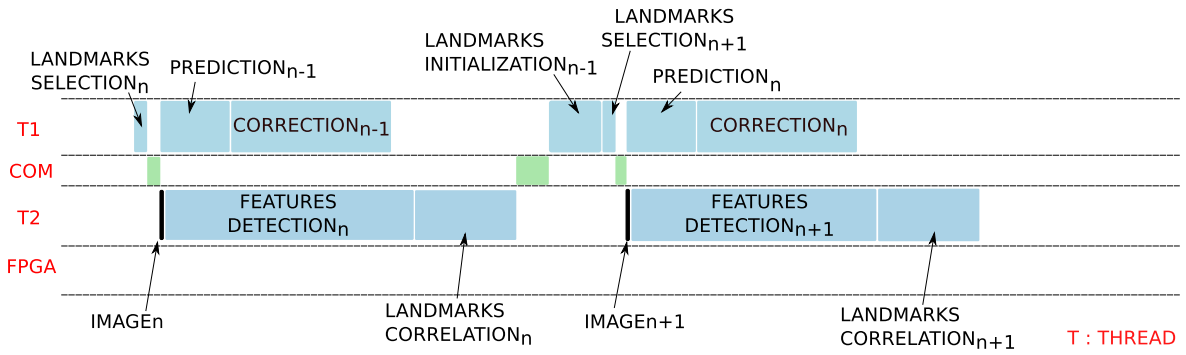

FIGURE 4.3 – Exécution des tâches du SLAM sur les deux coeurs du CPU ARM en fonction du temps.

prévu par l'optimisation par la modélisation SDF (cf. section 3.2.2), nous avons ajouté une FIFO entre les fonctions de *landmarks initialization* et *prediction*. Cette FIFO est symbolisée par le triangle vert dans la figure 4.2. Avec cette mise en place, nous avons trois FIFOs qui permettent de créer des files d'attentes pour les jetons de données (points rouges dans la figure 4.2) et permettent d'appliquer le re-timing de notre EKF SLAM (cf. section 3.2.3).

Suite à ces trois modifications du logiciel (utilisation des instructions SMID, multi-threading et instanciation de FIFOs bloquantes), nous avons obtenu un prototype qui ne répond pas encore aux contraintes fixées puisqu'il est capable d'atteindre 5 fps. Cependant, ces modifications prévoient les accélérations futures et facilitent l'instanciation des IPs VHDL dans le SLAM.

4.2 Accélération back-end

Cette section présente les travaux menés en parallèle à cette thèse par Daniel Tertei, doctorant au LAAS/CNRS. Ils sont le fruit de la **troisème itération** (cf. figure 3.13) de la méthodologie de co-design (exposée dans la section 2.3). Nous présenterons tout d'abord l'accélérateur de calculs matriciels EKF qu'il a développé en section 4.2.1. Puis nous décrirons son intégration dans la chaîne EKF SLAM en section 4.2.2. Cette modification a permis de diminuer les temps de traitement de la tâche de *correction*. Nous allons succinctement décrire son travail [Tertei et al., 2014].

4.2.1 Accélérateur de produits matriciels EKF

Dans [Thrun et al., 2005], les auteurs démontrent que la charge de calcul dans un algorithme EKF dépend du nombre d'amers N conservé dans la carte avec la relation : $L_{EKF} = O(N^2)$. Dans ce type de SLAM, les équations les plus coûteuses en ressources sont localisées dans la boucle de *correction*, au moment de la mise à jour de la matrice de cross covariance :

$$P^{(t+1)} = P^{(t)} - K^{(t)} Z^{(t)} K^{(t)T} \tag{4.1}$$

Cette tâche représente 85% de toutes les opérations à virgule flottante quand $N = 20$. D'autre part, elle représente 68% du temps nécessaire pour le calcul de toutes les équations du

bloc EKF. Nous obtenons une analyse identique en utilisant l'outil de profiling Gprof - [Spivey, 2003] - sur notre application EKF-SLAM. Nous pouvons en déduire qu'une accélération significative est possible en déportant les opérations à virgule flottante de l'opération matricielle KZK^T et les soustractions sur une fonction matérielle dédiée.

Une autre particularité du SLAM visuel basé EKF est que la matrice de covariance d'innovation est toujours de dimensions (2,2). Dans la littérature, il existe des solutions exécutant des opérations de multiplication matricielles avec une précision à virgule flottante sur architectures reconfigurables. Ces dernières sont orientées Processing Element - [Zhuo and Prasanna, 2007], [Jovanovic and Milutinovic, 2012]. En identifiant les opérations de base (multiplication a virgule flottante et addition) on peut concevoir une unité de calcul spécifique au problème appelé Processing Element (PE). Cette cellule est ensuite dupliquée afin de créer une architecture systolique. Le montage est commandé par des séquenceurs afin de traiter les flux de données entrantes dans un pipeline.

En se basant sur le concept de processing element, Tertei a réécrit l'équation 4.1 en un pseudo-code basé processing element (PE) :

Phase I : $K \times Z$

1: **for** $i = 0; i < 7N + 19; i++$ **do**
2: $PE_{in}(i) = \{K_{i0}, K_{i1}\}$
3: **for** $j = 0; j < 2; j++$ **do**
4: $PE_{in}(i) = \{Z_{0j}, Z_{1j}\}$
5: $PE_{out}(i) = K_{i0} \times Z_{0j} + K_{i1} \times Z_{1j}$
6: **end for**
7: **end for**

Phase II : $P - KZ \times K^t$

1: **for** $i = 0; i < 7N + 19; i++$ **do**
2: $PE_{in}(i) = \{KZ_{i0}, KZ_{i1}\}$
3: **for** $j = 0; j < 7N + 19; j++$ **do**
4: $PE_{in}(i) = \{{K^t}_{0j}, {K^t}_{1j}, P_{ij}\}$
5: $PE_{out}(i) = P_{ij} - (KZ_{i0} \times {K^t}_{0j} + KZ_{i1} \times {K^t}_{1j})$
6: **end for**
7: **end for**

avec la notation PE_{in} indicant l'entrée d'un Processing Element et PE_{out} sa sortie. Z_{ij} indique la donnée située à la ligne i et la colonne j de la matrice Z.

Les phases I et II sont exécutées séquenciellement sur les mêmes processing elements. Chaque PE consiste en deux multiplicateurs à virgule flottante, un additionneur à virgule flottante (qui exécute KZK^t, ligne 5 de la Phase I du pseudo code ci-dessus) et un soustracteur à virgule flottante (qui exécute $P - KZK^t$, ligne 5 de la Phase II du pseudo code ci-dessus). Grâce à la dimension fixe de la matrice Z , Tertei a pu instancier une multitude de PEs qui n'ont pas besoin de communiquer entre eux des résultats intermédiaires de multiplication (sur la ligne 5 dans les deux pseudocodes la sortie des PEs ne dépend pas des résultats intermédiaires de multiplication). Cette conception permet aux soustractions, dans l'équation 4.1, d'être exécutées directement après la multiplication à condition que chaque élément correspondant dans la matrice P ait été déjà récupéré dans la mémoire. Ainsi, non

seulement la multiplication tri-matricielle est exécutée dans un pipeline mais on gagne aussi en performances sur la soustraction car sa latence n'est désormais que d'un seul soustracteur de virgule flottante. Les résultats sont produits colonne par colonne afin d'éviter les problèmes d'adressage dans le logiciel - figure 4.4.

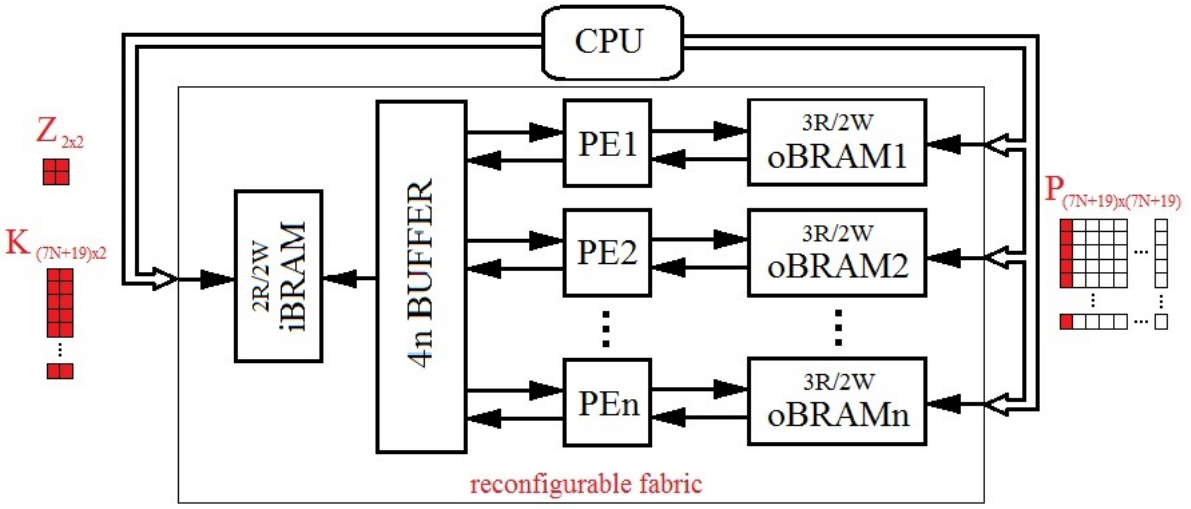

FIGURE 4.4 – Tuile générique pour l'équation 2.12

Les BRAM Tri-port (oBRAM) de la figure 4.4 sont en fait des BRAM double port avec un port multiplexé pour l'opération de lecture supplémentaire. Il est nécessaire d'avoir des BRAM avec une telle caractéristique afin d'assurer la récupération synchrone des P éléments de matrice correspondants pendant la multiplication KZK^t : on effectue une lecture et une opération d'écriture à partir d'adresses différentes (en utilisant deux ports) en même temps, tout en laissant un accès R/W [5] permanent au processeur. Comme l'ensemble de notre algorithme EKF SLAM est prévu pour fonctionner comme un SoC sur un dispositif Zynq, Tertei a conçu et mis en œuvre un accélérateur générique qui prend en compte à la fois la latence et les problèmes d'intégration mentionnés dans le chapitre 3. Quatre PEs ont été instanciés. En termes de latence, la performance est proche de la vitesse théorique - [Tertei et al., 2014]. Tertei a délibérément instancié peu de PE afin de limiter les ressources FPGA mobilisées (Le nombre de LUT pour la logique et celui des multiplicateurs embarqués - DSP84E - pour les calculs en virgule flottante). Cette limitation permet de réduire le pourcentage de surface FPGA utilisée permettant de raccourcir les chemins de connexion et augmenter la fréquence de fonctionnement maximale de l'accélérateur de produit de matrice.

La matrice carrée P étant la plus grande structure en termes d'utilisation de la mémoire de notre algorithme SLAM, il est impératif d'assurer une communication efficace entre l'accélérateur et le reste du système.

Dans le tableau 4.1, on peut voir l'influence de N^2 (où N est le nombre d'amers suivis) dans l'équation 4.1 liée au nombre d'accès mémoire, élément par élément, plus ou moins important en changeant la valeur de N. Un accélérateur qui retransférerait tous les résultats en mémoire externe (où les données de programme sont stockées) finirait par perdre son rôle d'accélérateur avec l'augmentation de N.

5. Read/Write : Lecture/Ecriture

Transferts (en%) pour	N=6	N=13	N=20
Equation 2.5	1.21	0.2	0.06
Equation 2.6	2.68	0.95	0.48
Equation 2.11	3.95	1.4	0.7
Equation 2.12	17.2	11	7.96
Equation 2.14	74.96	86.45	90.8

TABLE 4.1 – Analyse des transferts concernant la matrice P dans les équations EKF

De plus, parmi les opérations intervenant sur cette matrice P, seule l'équation 4.1 met à jour **toute** la matrice. La stocker en mémoire du FGPA au lieu de la stocker dans la mémoire externe permet de réduire les transferts de données entre le FPGA et le CPU. Le reste des opérations de la tâche de *correction* (les équations 2.9, 2.10, 2.11, 2.12 et 2.13 dans la section 2.1.2) s'exécutant sur CPU, une modification logicielle permet d'aller lire et écrire les données en mémoire FPGA à chaque exécution. Ainsi, à chaque itération EKF :

1. Les matrices P_1, P_2, P_3 et P_4 sont récupérées et mises à jour depuis la mémoire on-chip via un bus AXI dédié ;
2. Les matrices Z et K sont copiées de la mémoire externe vers la mémoire on-chip (iBRAM) ;
3. Les données de contrôle sont communiquées directement de l'espace utilisateur Xillinux vers le co-processeur ;
4. La logique va chercher par elle-même les matrices Z, K et P et stocke le résultat dans P (oBRAM) via le bus dédié XIL dédié XIL_BRAM ;
5. Et, enfin, le processeur est informé de la fin de l'exécution par l'intermédiaire d'un signal d'interruption.

La mise en œuvre d'une communication efficace et à faible latence avec le co-processeur garantit une exécution plus rapide de la tâche réordonnancée de *correction* (section 3.2.3) de notre SLAM visuel. De cette façon, Tertei a pu maintenir le nombre de FLOPs dans l'équation 4.1 proche d'une valeur constante en le rendant indépendant de N^2 en termes de temps d'exécution processeur. Étant donné que la plateforme hétérogène utilisée pour la co-conception de notre prototype de SLAM embarqué est la ZedBoard, cet accélérateur est un périphérique AXI mappé en mémoire d'un processeur ARM Cortex A9 (Figure 4.5(A)).

Afin de supporter cette communication Sw/Hw dans Xillinux, Tertei a d'abord déclaré des variables globales pour la matrice de cross-covariance P - *obram*, le co-processeur *triMatrix* mappé en mémoire et la mémoire d'entrée *ibram* :

```
//config.h
#define N        20
#define MAP_SIZE 7*N+19
#define PE       4
#define EKF      0x80000000
#define oBRAM    8192
#define TRIMAT   1024
#define iBRAM    1024
```

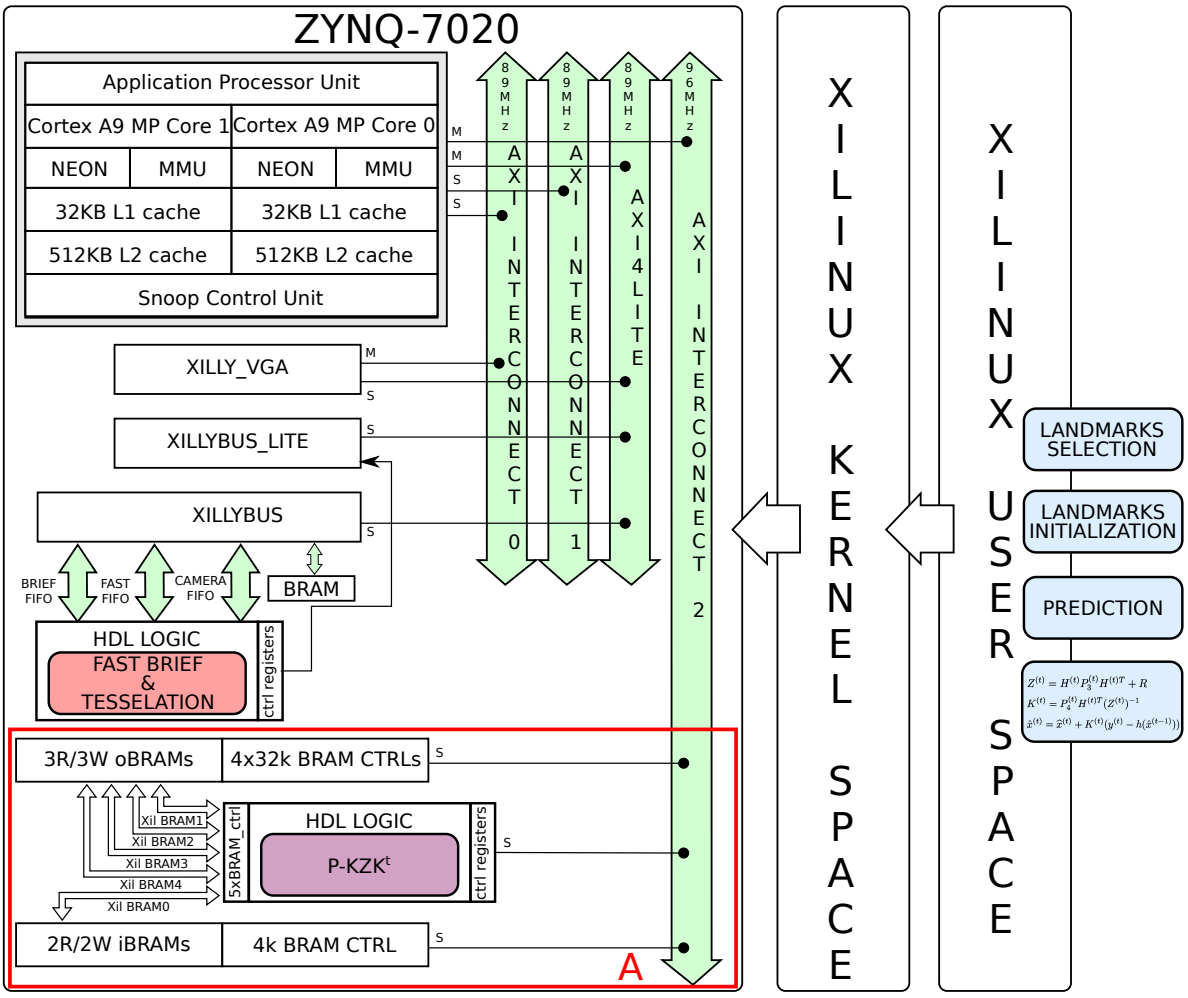

FIGURE 4.5 – Intégration de l'architecture matérielle dédiée dans la chaîne EKF-SLAM.

```
#define BYTES    (PE*oBRAM+TRIMAT+iBRAM)*4

extern volatile uint* ibram;
extern volatile uint* trimatrix;
extern float** obram;
```

Dans le programme *main*(), Tertei a aussi instancié des variables globales :

```
//main.c
...
#include "config.h"
int fd;
volatile uint* ibram;
volatile uint* trimatrix;
float** obram;
...
void init_ekf(){

fd = open("/dev/mem", O_RDWR);
uint map = mmap(0, BYTES, PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, EKF);
obram=(float**)malloc(sizeof(float*)*MAP_SIZE);
int i,j,temp;
// OBRAM INIT
temp = ceil((double)MAP_SIZE/PE);
for(i = 0; i < temp; i++){
  for(j = 0; j < PE; j++){
    obram[i*PE+j] = map+j*oBRAM+MAP_SIZE*i;
  }
}
// TRIMATRIX INIT
trimatrix = map+PE*oBRAM;
// IBRAM INIT
ibram = map+PE*oBRAM+TRIMAT;
}// end init_ekf{}
...
int main(char* args[]){
...
// sw/hw integration
init_ekf();
float** P = obram;
...
//in correct_lmk()
  // copy matrices Z and K
  memcpy(ibram,&Z,4*sizeof(float));
  memcpy(ibram+4,&K,2*MAP_SIZE*sizeof(float));
```

```
  // reset accelerator
  trimatrix[0] = 0x0000ffff;
  // send go
  trimatrix[0] = 0xffff0000;
  trimatrix[0] = 0x0000000;
...
}// end main(char* args[])
```

Enfin, ce code a été écrit de manière générique afin de supporter des changements algorithmiques et/ou architecturaux.

4.2.2 Intégration dans la chaîne SLAM

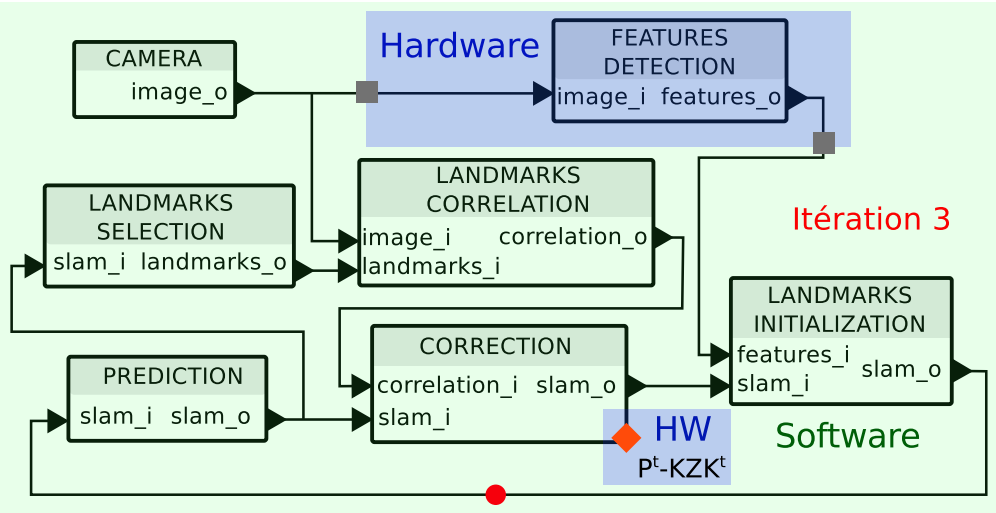

FIGURE 4.6 – Architecture fonctionnelle du SLAM avec le partitionnement réalisé à l'itération 3 de la méthodologie de co-design. Les carrés gris représentent des FIFOs matérielles et les losanges oranges représentent une mémoire (BRAM) partagée.

Pour intégrer l'accélérateur de back-end dans le SLAM, Daniel Tertei a utilisé une mémoire partagée utilisant les BRAM du FPGA. La figure 4.6 présente l'architecture fonctionnelle du SLAM avec le partitionnement réalisé à la troisième itération de la méthodologie de co-design. On peut voir un losange orange représentant la mémoire partagée disponible en lecture/écriture par les parties matérielle et logicielle. Cette mémoire partagée contient la matrice P.

Après l'intégration de cet accélérateur back-end dans la chaine EKF SLAM, le changement de synchronisation des tâches du 3D EKF SLAM est illustré sur la figure 4.7.

Comme cet IP accélérateur est générique, cette technique de co-conception peut être migrée vers des systèmes reconfigurables plus gros en cas de besoin de puissance de traitement plus forte (Pour permettre une augmentation du nombre de PEs instanciés) . L'accélérateur a également été testé avec succès en tant que périphérique PLB pour les systèmes hétérogènes FPGA avec le processeur IBM PowerPC440 intégré.

Le tableau 4.2 détaille les taux d'occupation processeur (ARM cortex A9) en fonction des tâches. Dans ce dernier, on remarque que le taux d'images par seconde passe de $14,9$img/s à

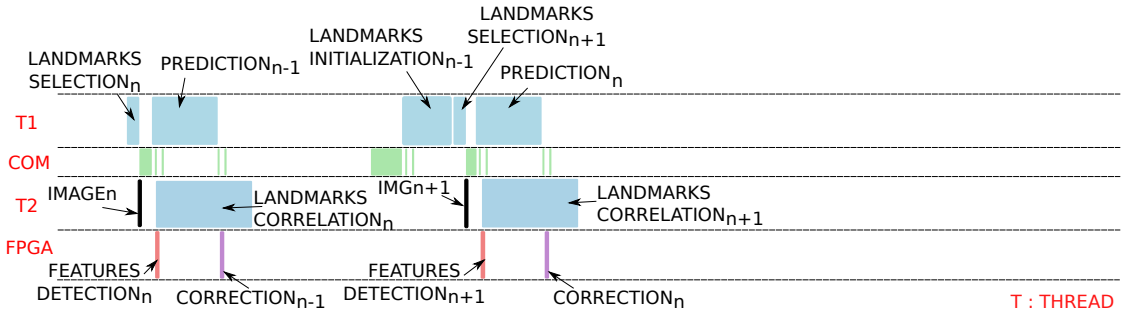

FIGURE 4.7 – Synchronisation des tâches du 3D EKF SLAM après accélération du produit de matrice

20, 7img/s. Cette amélioration est due aux performances de l'accélérateur matériel de produit matriciel qui n'occupe plus que 10% du temps le CPU.

Itération n^o	I	II	III ($P^t - KZK^t$ en HW)
Taches C-SLAM	Taux d'occupation CPU [%]		
Communication	5.97	17.83	24.84
Sélection des amers	0.4	1.19	1.66
Prédiction	1.04	3.12	4.35
Correction	11.94	35.66	10.35
Initialisation	0.15	0.45	0.62
Caméra	6.47	19.32	26.92
Corrélation	4.38	13.08	18.22
Détection de coins	69.65	9.36	13.04
Taux d'img/s [Hz]	5	14.9	20.7

TABLE 4.2 – Taux d'occupation CPU des tâches de C-SLAM.

La figure 4.8 illustre l'amélioration obtenue en comparant les exécutions de C-SLAM des prototypes I [6] et II [7] avec le prototype obtenu à l'itération II de la méthodologie de co-design (prototype II + accélération de produits matriciels).

6. EKF SLAM entièrement en logiciel.
7. EKF SLAM logiciel avec accélérateur matériel de détection de points d'intérêt FAST (la fonction *features detection*.

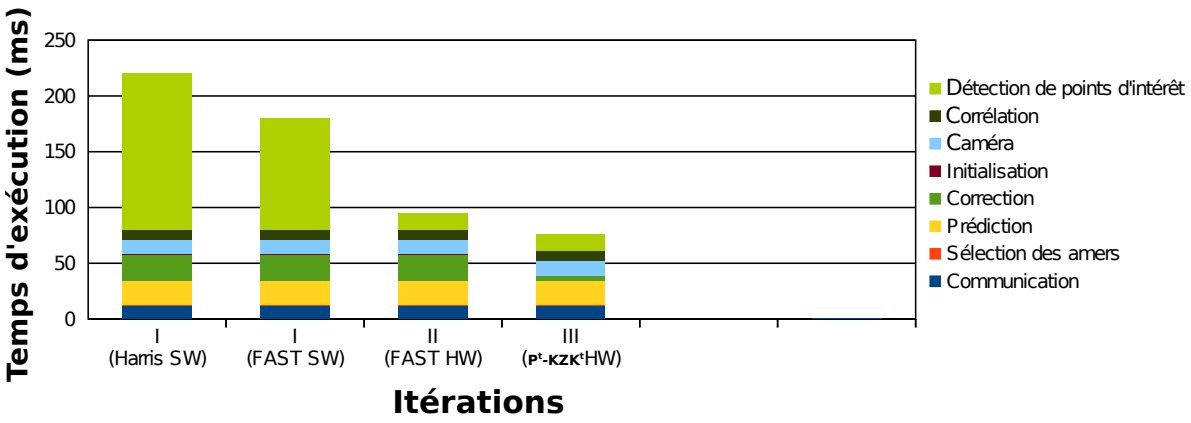

FIGURE 4.8 – Comparaison du temps d'exécution des prototypes I (C-SLAM en logiciel avec le détecteur de points d'intérêt Harris), I (C-SLAM en logiciel avec le détecteur de points d'intérêt FAST), II (C-SLAM logiciel avec accélérateur matériel de détection de points d'intérêt FAST) et le prototype III (C-SLAM avec l'accélérateur matériel de produit matriciel).

Chapitre 5

Accélérateur front-end

Ce chapitre est consacré aux itérations de la méthodologie permettant l'intégration itérative d'accélérateurs matériels (sur FPGA) originaux dans une approche HIL [1] pour le traitement d'images mis en jeu dans notre chaîne SLAM. Nous avons effectué deux itérations afin d'implanter la partie vision (ou front-end) dans la logique du FPGA. Dans un premier temps, nous présenterons la conception et l'intégration d'un détecteur de points d'intérêt - section 5.1 - couplé à un système de répartition de ces caractéristiques dans l'image - section 5.2. Puis nous présenterons le développement d'un descripteur de coins en section 5.3 et d'un module de corrélation en section 5.4. Tous ces modules permettront de traiter toutes les données de bas niveau (les pixels de l'image) sur la partie FPGA.

5.1 Conception d'un accélérateur pour l'extraction de points

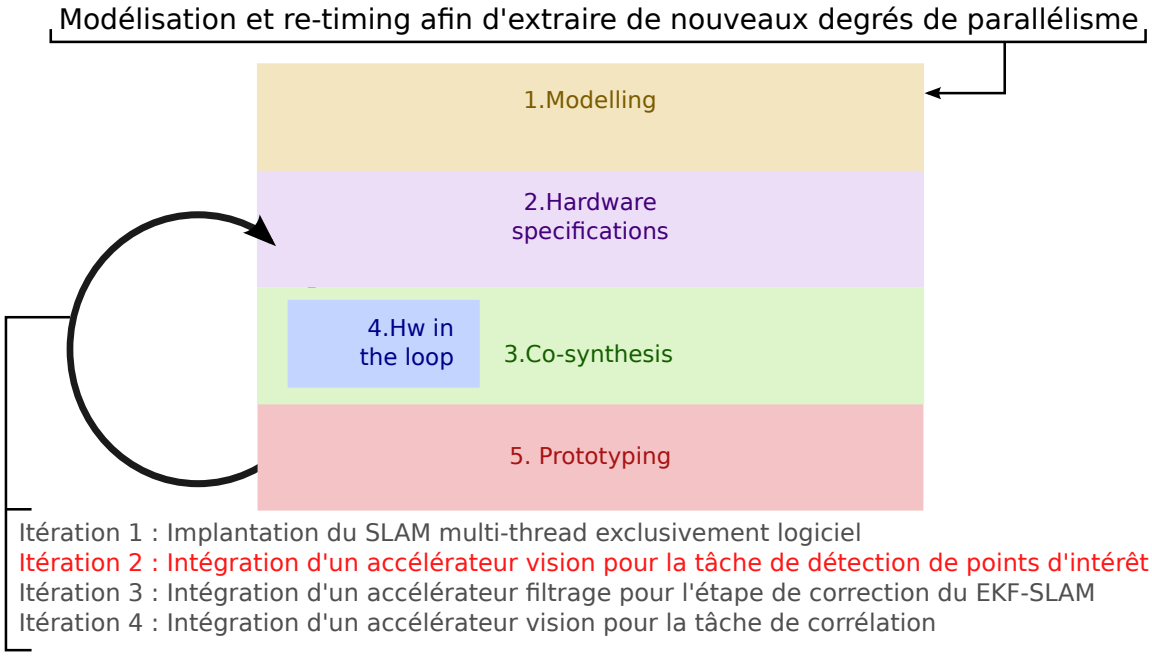

Figure 5.1 – Les différentes itérations de la méthodologie de co-design effectuées avant d'obtenir un système qui satisfait les contraintes imposées.

Cette étape de la conception correspond à la **seconde** itération de la méthodologie de co-design, figure 5.1. Après avoir optimisé notre système en travaillant uniquement sur la

1. Hardware-In-The-Loop

partie logicielle (section 4.1), nous avons remarqué que la fonction la plus coûteuse en temps processeur était la détection de points d'intérêt (69.65% du temps de calcul total, rappel dans le tableau 5.1).

Cette section est consacrée à l'accélération front-end d'un détecteur de points d'intérêt. Nous allons décrire le processus de développement d'accélération sur logique FPGA d'un détecteur matériel original de points d'intérêt ainsi que l'implantation de la stratégie de répartition des points d'intérêt dans l'image.

Itérations n^o	I (HARRIS SW)
Taches C-SLAM	Taux d'occupation CPU [%]
Communication	5.97
Sélection des amers	0.4
Prédiction	1.04
Correction	11.94
Initialisation	0.15
Caméra	6.47
Corrélation	4.38
Détection de coins	69.65
Taux d'img/s [Hz]	5

TABLE 5.1 – Taux d'occupation CPU des tâches de C-SLAM.

5.1.1 Détecteur de points d'intérêt FAST

Comme nous l'avons mis en avant dans l'état de l'art présenté au paragraphe 2.1.6, le détecteur FAST est précis, rapide et à priori adapté pour une implantation sur architecture hardware. Nous présentons dans cette section les étapes de conception et d'implantation d'un IP basé sur [Kraft et al., 2008] et optimisé pour réaliser la détection de points d'intérêt dans une image sur FPGA présenté dans [Brenot et al., 2015].

5.1.1.1 Principe de fonctionnement

L'algorithme FAST est composée de deux parties principales. Une première qui calcule et attribue un score au pixel traité P - figure 5.2. La seconde est une partie de validation du coin.

Pour cela, FAST se base sur un cercle de Bresenham de 16 pixels centré autour du pixel traité afin de déterminer si ce dernier est un coin. Chaque pixel de ce cercle est numéroté de 1 à 16 dans le sens horaire. Un point est validé comme étant un coin, si :

— Un ensemble de N pixels contigus du cercle sont tous plus **clairs** (ou **brighter**) que l'intensité du pixel candidat P (noté I_p) auquel on ajoute une valeur de seuil t (par exemple $t = 10$) ;

— Un ensemble de N pixels contigus du cercle sont tous plus **foncés** (ou **darker**) que l'intensité du pixel candidat P moins la valeur de seuil t.

En parallèle de cette validation, un module attribue grâce à deux chaînes de traitement deux scores au pixel courant. Il utilise les informations données par le module **thresholder** et calcule :

— un score appelé *bright* en prenant en compte tous les pixels plus clairs ;
— un score appelé *dark* en prenant en compte les pixels plus foncés.

Enfin, on attribue au candidat P le score plus élevé entre le score bright et le score dark.

Le score de coin est calculé selon l'équation 5.1 où t est un seuil, $I_{p\to x}$ est l'intensité du pixel de test de segment et I_p est l'intensité du pixel candidat.

$$V = max(\sum_{x\in Sbright} |I_{p\to x} - I_p| - t, \sum_{x\in SDark} |I_p - I_{p\to x}| - t) \tag{5.1}$$

La fonction V de calcul du score, donnée dans l'équation 5.1, est définie comme étant la somme des différences absolues entre l'intensité du pixel central et les intensités des pixels sur le cercle Bresenham .

La figure 5.2 montre l'exemple d'un coin avec le cercle de Bresenham permettant de calculer un score et valider le coin.

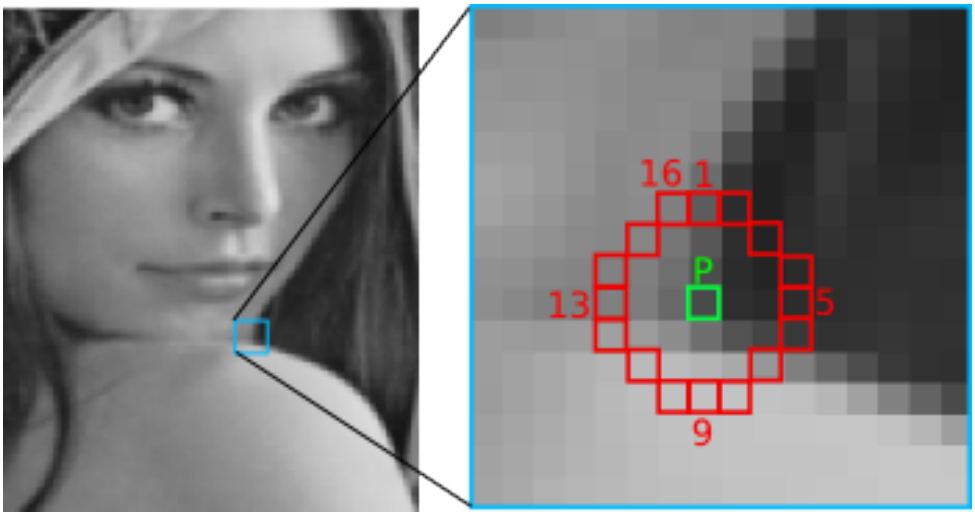

FIGURE 5.2 – Un exemple de segment de test FAST. Le pixel candidat courant est noté P, les pixels du cercle de Bresenham sont numérotés dans le sens horaire.

Dans l'exemple donné en figure 5.2, les pixels 1 à 5 ont une valeur proche du pixel candidat P. Les intensités des pixels 6 à 16 sont plus claires que l'intensité de P, I_p. Comme ils sont tous contigus, cela signifie que le pixel P est validé comme étant un coin.

5.1.1.2 Implantation et validation logicielle

Les contraintes de temps de traitement de notre chaîne SLAM sont de 30 img/s, le bloc d'accélération matériel FAST doit être capable de traiter une image toutes les 1/30 secondes. Pour une résolution d'images de 640x480 pixels cela représente une fréquence pixels de 9.216Mhz. Nous souhaitons donc obtenir une fréquence de fonctionnement **supérieure** 9.216**MHz**. Pour calculer la contraite de latence maximum, nous nous sommes basé sur la datasheet d'un cap-

teur d'image utilisé au LAAS : le MT9V034 [2] de la société Aptina® [3]. Les caractéristiques de ce dernier indiquent qu'il y a, entre deux image, une latence incompréssible de 19 lignes. Afin de ne pas ralentir l'acquisition des images, l'accélérateur FAST ne doit pas dépasser ces **19 lignes de latence** car nous souhaitons commencer l'étape d'initialisation des amers avant le début de la prochaine acquisition d'image.

Nous avons tout d'abord remplacé le détecteur Harris par FAST dans C-SLAM afin de valider son comportement logiciel dans notre système. Une version *C* est disponible (C-FAST [4]), provenant de l'article [Kraft et al., 2008]. Suite à cette implantation, nous n'avons constaté aucun changement de précision dans le SLAM. Ceci est dû à notre politique de répartition des points dans l'image, la tessellation de l'image (section 2.1.7). En effet, l'image est divisée en cases et nous ne sélectionnons que le meilleur coin de chacune d'entre elles. Ainsi, Harris et FAST délivrent tous deux un seul coin facile à retrouver (car ayant un très bon score, il se détache bien de l'arrière plan) lors de la mise en correspondance. En revanche, nous avons pu constater une amélioration du temps de traitement. Le diagramme 5.3 illustre cette amélioration en la comparant à une exécution C-SLAM avec le détecteur Harris et C-SLAM avec FAST en logiciel. Le tableau 5.2 détaille les taux d'occupation processeur (ARM cortex A9) en fonction des tâches.

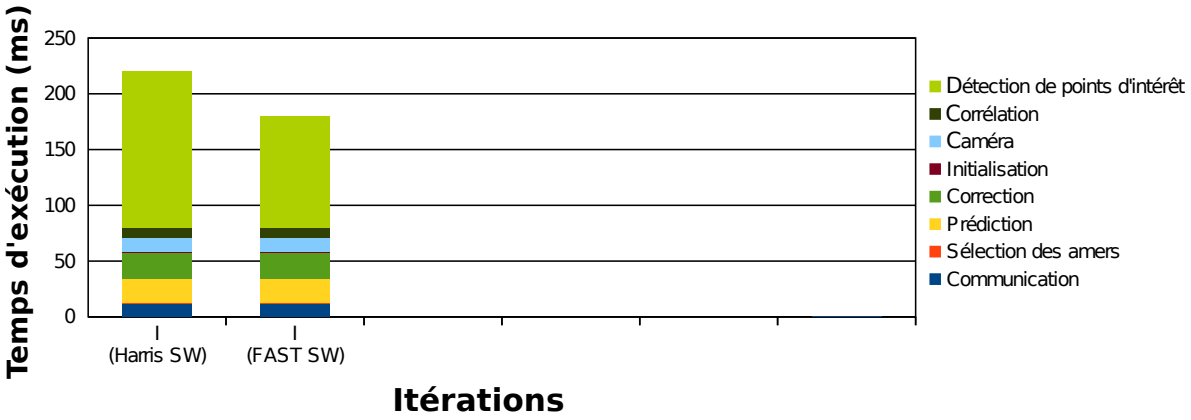

FIGURE 5.3 – Comparaison du temps de calcul des tâches de C-SLAM avec le détecteur Harris et FAST.

Cette validation logicielle nous a conforté dans notre choix de l'algorithme FAST et nous allons maintenant présenter les étapes de conception que nous avons effectuées afin de déporter le détecteur de points d'intérêt vers une fonction matérielle dédiée.

5.1.1.3 Gestion du flux de pixels

En matériel les traitements sont effectués sur un flux 1D continu de pixels. Les pixels sont donc stockés dans une mémoire (BRAM) tampon afin d'alimenter le bloc détection FAST par un cercle de pixel Bresenham, centré autour du pixel traité. La figure 5.4 illustre le mode de

2. Capteur paramétrable à 60 img/s ou 30 img/s avec une résolution de 752x480 pixels. http ://www.onsemi.com/pub_link/Collateral/MT9V034-D.PDF
3. http ://www.onsemi.com/
4. http ://www.edwardrosten.com/work/fast.html

Itération n^o	I (HARRIS SW)	I (FAST SW)
Taches C-SLAM	Taux d'occupation CPU [%]	
Communication	5.97	7.45
Sélection des amers	0.4	0.5
Prédiction	1.04	1.3
Correction	11.94	14.91
Initialisation	0.15	0.19
Caméra	6.47	8.07
Corrélation	4.38	5.47
Détection de coins	69.65	62.11
Taux d'img/s [Hz]	5	6.2

TABLE 5.2 – Taux d'occupation CPU (ARM Cortex A9) des tâches de C-SLAM.

stockage du flux de pixels de la caméra afin de conserver constamment une fenêtre 7x7 pixels disponible en registre d'où est extrait le cercle de Bresenham. Le nombre de lignes stocké dépend de la taille du cercle de Bresenham : pour un cercle de diamètre N pixels, il faut stocker N lignes.

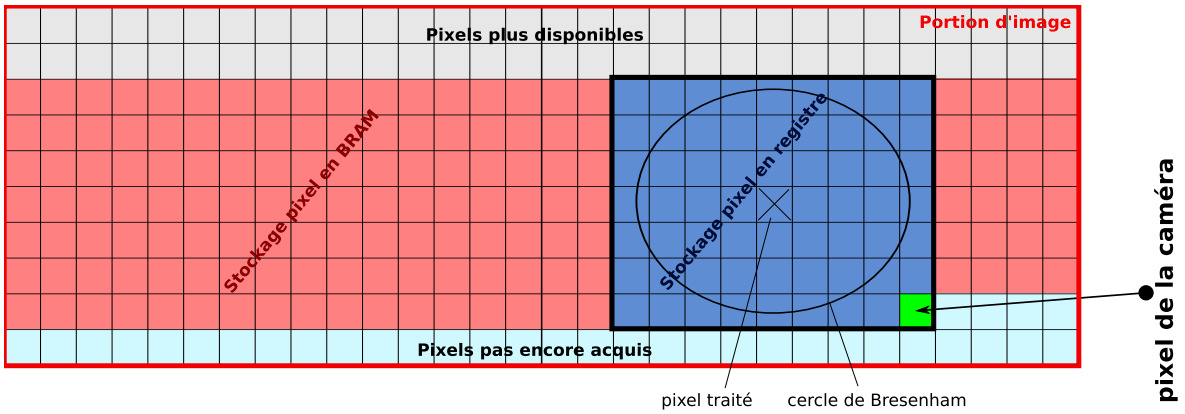

FIGURE 5.4 – Stockage du flux 1D de pixels en BRAM et mise à jour d'une fenêtre de 7x7 pixels glissante.

Ce stockage est géré par le bloc *Block* $N \times N$ dans la figure 5.5. Ce bloc effectue trois opérations en parallèle. Il stocke un buffer glissant de $(N-1)_{lignes} + (N-1)_{pixels}$ en BRAM qui est constamment mis à jour au fur et à mesure de l'acquisition des nouveaux pixels (pixels en rouge dans la figure 5.4). D'après ces données, il met à jour une fenêtre de $N \times N$ pixels stockée en registre (pixels en bleu dans la figure 5.4). Enfin, il prélève dans cette fenêtre les pixels du cercle de Bresenham et le pixel central P (le pixel traité) afin de les rendre disponibles pour la chaîne de traitement FAST.

5.1.1.4 Architecture matérielle d'extracteur de points d'intérêt FAST

La figure 5.5 présente l'architecture matérielle globale du détecteur de coins de type FAST.

Dans cette figure :

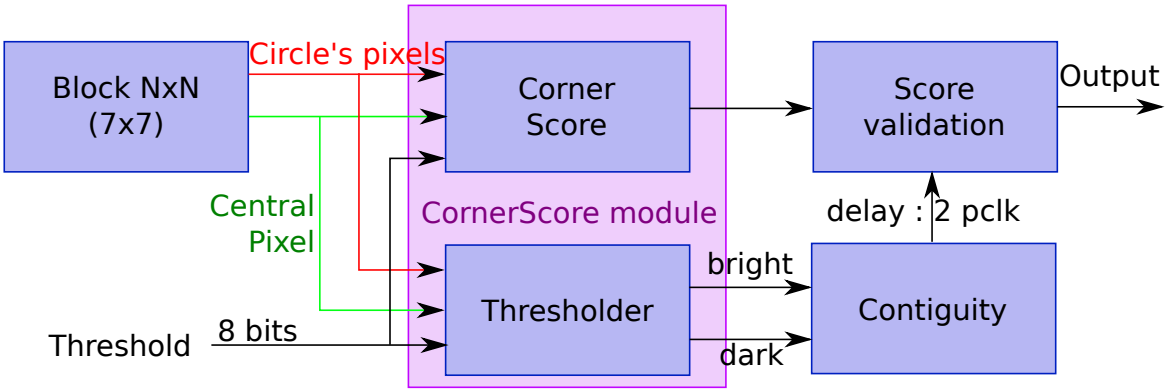

FIGURE 5.5 – Architecture matérielle du détecteur de points d'intérêt FAST.

— *Thresholder* : Compare chaque pixel du cercle avec le pixel central et fournit deux vecteurs de 16 bits. Un premier vecteur contient les résultats de chaque comparaison : 1 si le pixel du cercle est plus clair que le pixel central, 0 dans le cas contraire. Le deuxième vecteur est construit de la même manière mais correspond aux tests vérifiant si les pixels du cercle sont plus foncés que le pixel central ;
— *Corner Score* : Le module de score de coin permet de calculer le score, équation 5.1. Ce module utilise les données fournies par le module *Thresholder* (2 vecteurs de 16 bits) ;
— *Contiguity* : Ce module permet de savoir si au moins 9 pixels contigus du cercle de Bresenham sont tous plus foncés ou plus clairs que le pixel central (+/- un seuil).

Enfin, un module Non maximum suppression permet de supprimer les coins détectés par FAST qui sont inférieurs à un seuil.

Le module *Contiguity* teste la contiguïté des pixels en effectuant 9 comparaisons. Ainsi, huit comparateurs sont utilisés pour tester des arcs de 9 pixels contigus sur le cercle de Bresenham. Ces vecteurs 9 pixels sont donnés par V tel que $V[m] = [1 + m \rightarrow 9 + m]$ avec $m \in [0 \rightarrow 7]$. Ces 9 comparaisons sont traitées en parallèle. Si une d'entre elles satisfait les conditions (tous les pixels sont plus foncés ou clairs que le pixel central), le score est validé.

L'architecture du bloc *thresholder* est présentée dans la figure 5.6 et celle du bloc *Corner Score* est décrite par la figure 5.7.

L'accélération matérielle est obtenue par l'instanciation d'arbres d'additions qui calculent les additions en pipeline (*adder tree*, dans la figure 5.7). Comme expliqué dans 2.2.2.2, les optimisations matérielles peuvent tirer parti du pipelining comme illustré dans la figure 5.8. Chaque étage du pipeline est séparé par une bascule D (ou flip flop). Dans notre cas, nous avons 8 entrées (4 additionneurs à 2 entrées).

1. Au premier coup d'horloge, les 4 résultats d'additions, $r1_1$ à $r1_4$, sont stockés dans des bascules (1er étages de l'arbre) ;
2. Au deuxième coup d'horloge, ces 4 résultats ($r1_1$ à $r1_4$) sont additionnés entre eux, générant 2 valeurs $r2_1$ et $r2_2$ (étage 2 de l'arbre). Pendant ce temps, 8 nouvelles valeurs peuvent être injectées au premier étage de l'arbre ;
3. Au troisième coup d'horloge, $r2_1$ est additionné à $r2_2$ afin de donner le résultat final $r3$ (troisième étage de l'arbre).

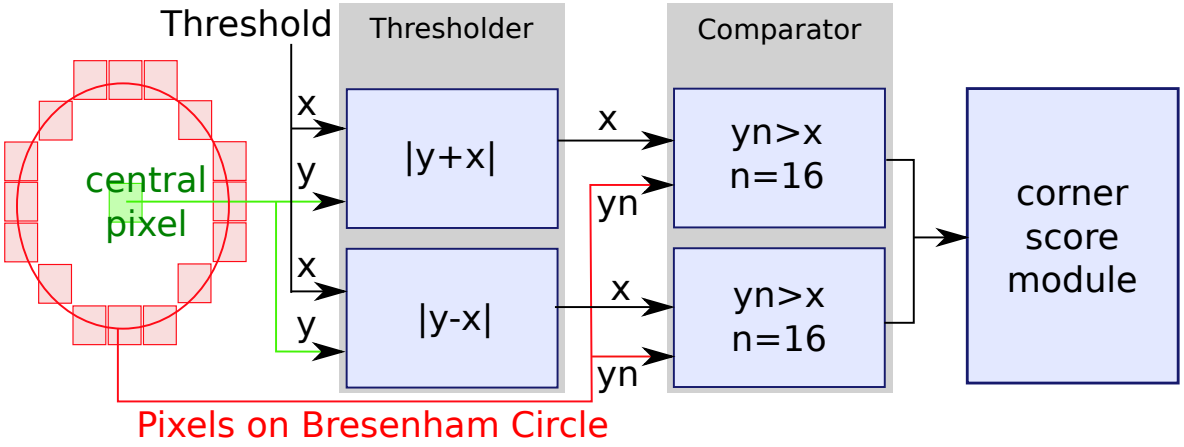

FIGURE 5.6 – Module thresholder et comparator permettant la détection/validation des points d'intérêt.

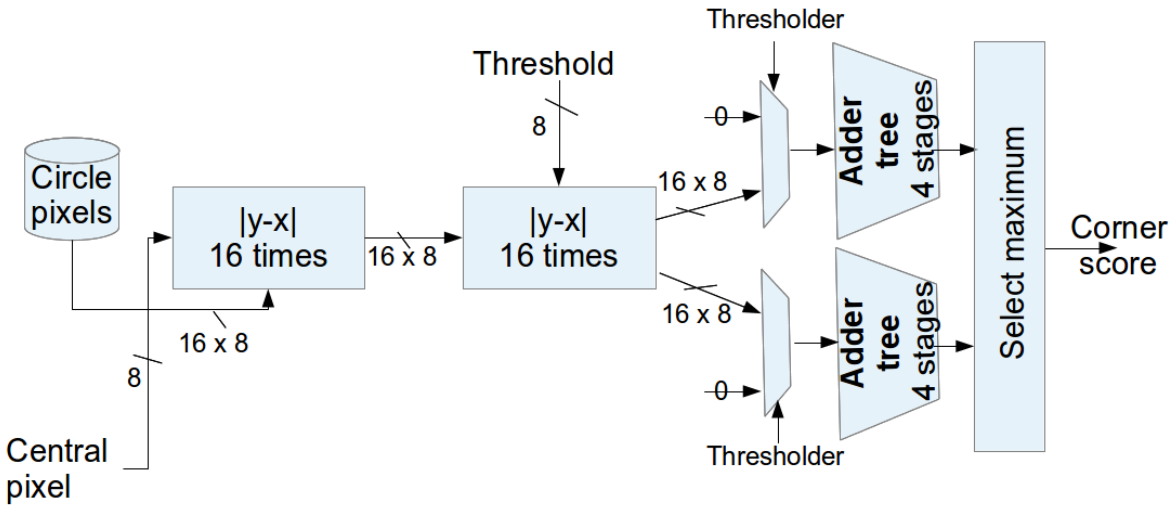

FIGURE 5.7 – Module de calcul du score de FAST.

Ainsi, l'arbre peut générer un résultat de 8 additions par cycle d'horloge (avec une latence de 2 cycles d'horloge).

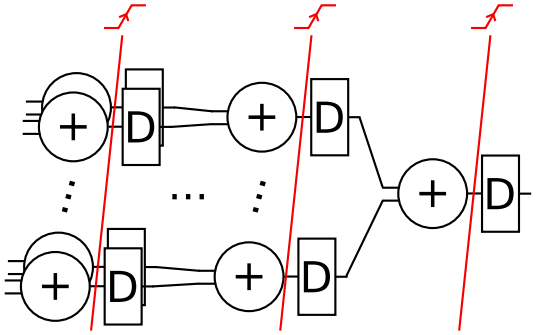

FIGURE 5.8 – Exemple d'arbre d'addition.

5.1.1.5 Configuration

L'architecture matérielle globale que nous avons développée pour l'algorithme FAST est générique. La taille du cercle peut être facilement paramétrée, tout comme le seuil t et le nombre de pixels devant être contigus (grâce à un *généric* dans le VHDL). le système peut également être implanté sur un FPGA Xilinx ou Altera (aucun IP propriétaire n'a été utilisé pour le développement). Nous avons fixé la taille du cercle Bresenham 16 pixels et le test de contiguïté à un arc de 9 pixels. Cette configuration est la plus efficace d'après [Rosten and Drummond, 2005].

5.1.1.6 Validation

Après avoir validé le comportement de notre architecture en SystemC, nous l'avons développée en VHDL afin de l'implanter. Pour cela, nous avons validé le code VHDL en simulation grâce à un banc d'essai présenté sur la figure 5.9. Dans ce dernier, un module VHDL, *virtual camera*, permet de simuler les signaux de synchronisation (hsync, vsync et pclk) et données (pixel) d'une caméra à partir d'une image stockée sur disque. Un deuxième composant, *image writer*, permet de collecter et mettre en forme les résultats de FAST.

Une fois cette validation par la simulation effectuée, nous avons implanté le module FAST sur FPGA (ZynQ® 7020). Pour cela, nous avons utilisé la distribution Xillinux. L'intégration est présentée dans la figure 5.11. Afin de valider le bloc FAST, nous avons tout d'abord utilisé une image de synthèse (fond blanc avec des formes rectangles noirs) puis nous l'avons testé avec

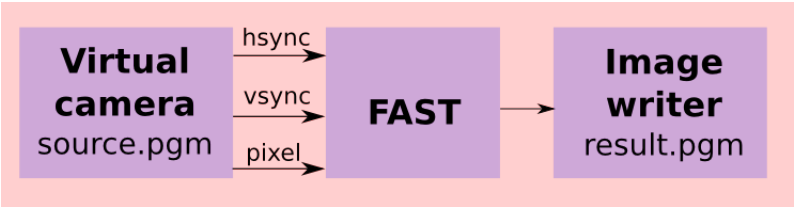

FIGURE 5.9 – Banc d'essai de simulation de l'architecture matérielle FAST.

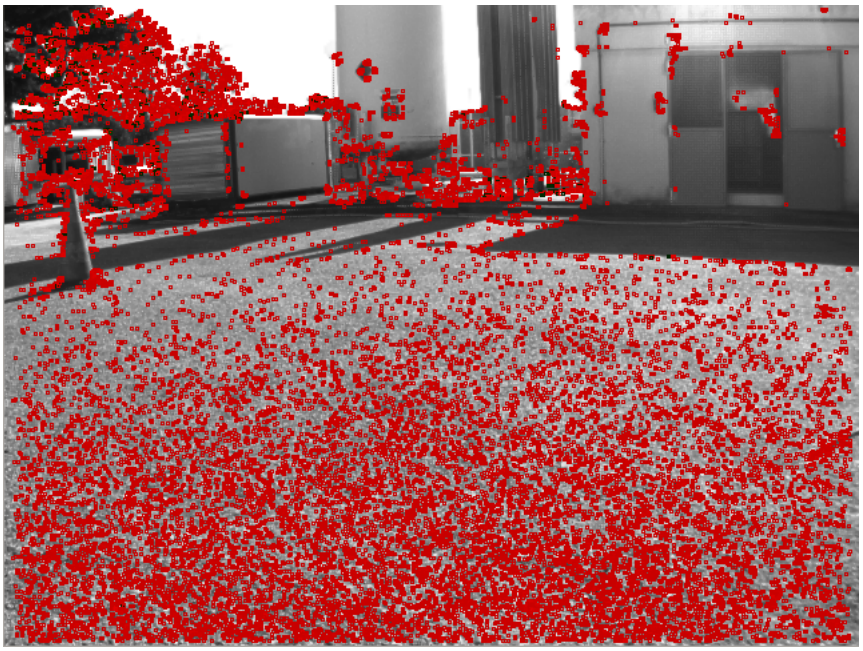

FIGURE 5.10 – Résultats du détecteur de coin FAST sur architecture matérielle.

une image réelle afin de valider fonctionnellement son comportement. La figure 5.10 montre les résultats obtenus après le traitement d'une image par notre module matériel FAST sur FPGA. Cette image provient d'une séquence d'images acquise au LAAS. Chaque coin détecté apparaît comme entouré d'un carré (généré par une fonction OpenCV). Cet exemple montre que notre architecture fournit des coins répartis dans toute l'image même si, habituellement, la surface en graviers du sol au premier plan complique grandement la détection de points d'intérêt.

Nous avons constaté que FAST en matériel et en logiciel détecte les mêmes points d'intérêt (ce qui est logique puisque les algorithmes sont identiques). En revanche, nous avons pu noter une faible différence sur le nombre de points total. En effet, en moyenne nous obtenons -1,6% de points d'intérêt en matériel. Cette différence est due à la gestion des bords de l'image. En matériel, nous avons choisi de ne pas calculer de score tant que le cercle de Bresenham n'est pas complet (pour un cercle 7x7, nous ignorons les 7 premières/dernières lignes et les 7 premières/dernières colonnes de l'image, présentée en figure 5.10). En sachant que les bords des images sont les plus déformés par l'optique de caméra, ces pertes sont minimes. En outre, dans le cas du SLAM, utiliser de tel points d'intérêt risque d'intégrer des mesures imprécises et difficilement ré-observables.

Sur notre cible FPGA (Xilinx® ZynQ® zc7z20), nous obtenons une fréquence maximale de 160 MHz (17, 36 fois supérieur à la contrainte unitaire fixée à 9.216Mhz). Le bloc matériel FAST peut traiter un maximum de 160 Mpixels/s avec une latence de 7 lignes (2 fois inférieur à la contrainte unitaire fixée à 19 lignes de latence maximum). Cette bande passante correspond à 77 images/seconde pour une image de résolution HD (1920x1080 pixels) avec une consommation maximale de 2,5 Watts (estimée par l'outil d'analyse Xilinx® Xpower Analyser®). Ces résultats mettent en évidence le très bon rapport puissance de calcul/consommation et valide cet IP de détection de points dintérêt FAST. Le tableau 5.1.1.6 résume l'utilisation des ressources du FPGA.

Slice Logic Utilization	Used	Available	Utilization
Slice LUTs	5963	53200	11%
BRAM (18k/36K)	0/1	140 (38K)	0.5%
Slice registers	8281	106400	7%

TABLE 5.3 – Utilisation des ressources logiques du Zynq xc7z20-1clg484.

En comparaison l'architecture proposée dans [Kraft et al., 2008] utilise 12 BRAMs, 2368 LUTs et 1547 registres d'un FPGA Spartan 3 et fonctionne à une vitesse de 130 Mhz.

5.1.1.7 Intégration dans la chaîne SLAM

Afin d'intégrer notre accélérateur de détection de points d'intérêt dans notre application SLAM, nous avons utilisé les moyens de communications fournies par Xillybus. Cette intégration est présenté dans la figure 5.11 qui illustre l'architecture du Zynq. Le nombre de points d'intérêt n'étant pas fixe, il dépend des paramètres de résolution d'image et de tesselation (voir section 5.2.1), nous avons utilisé des FIFOs pour transférer les résultats (les coins détectés) de la partie matérielle vers le CPU (*FAST FIFO*, cf figure 5.11). La position des coins est codée sur 2x10 bits et le score associé est codé sur 12 bits. Nous avons donc utilisé une FIFO de 32bits de large pour transférer les données. Une autre FIFO a été utilisée pour transmettre les pixels de l'image du CPU vers la logique (*CAMERA FIFO*, cf figure 5.11).

De plus, nous avons contrôlé le nombre de données transférées en implantant en parallèle une mémoire partagée (*ctrl registers*, cf figure 5.11). Cette dernière est mise à jour par le matériel en fin de tâche, Il y stocke le nombre de points détectés. Le logiciel récupère cette valeur pour vérifier le nombre de données contenu dans la FIFO puis la remet à zéro pour indiquer que tous les points d'intérêt ont été collectés.

Pour intégrer cet accélérateur dans la chaîne SLAM, nous avons désactivé la détection de points d'intérêt logicielle et l'avons remplacée par une séquence qui, dans cet ordre, synchronisent le matériel à chaque nouvelle image :

1. Activation du module matériel (via les mémoires partagées) ;
2. Envoi des pixels de l'image vers la logique ;
3. Attente de la fin du traitement (attente d'un nombre n de points d'intérêt différent de 0 en mémoire partagée) ;
4. Récupération des n données dans la FIFO ;
5. Mise en forme et stockage des n éléments dans les structures de données du SLAM.

5.1.1.8 Validation du prototype SLAM

Après l'intégration de cet algorithme, nous obtenons le même ordonnancement des tâches de notre 3D EKF SLAM que prévu à l'étape de partitionnement de la méthodologie de co-design. Ce dernier est illustré en figure 5.12.

Suite à l'intégration du module matériel FAST dans le SLAM, nous avons procédé à une validation du comportement. Aucune dégradation significative des performances de la chaîne SLAM n'a été remarquée après l'accélération matérielle de la détection de points d'intérêt

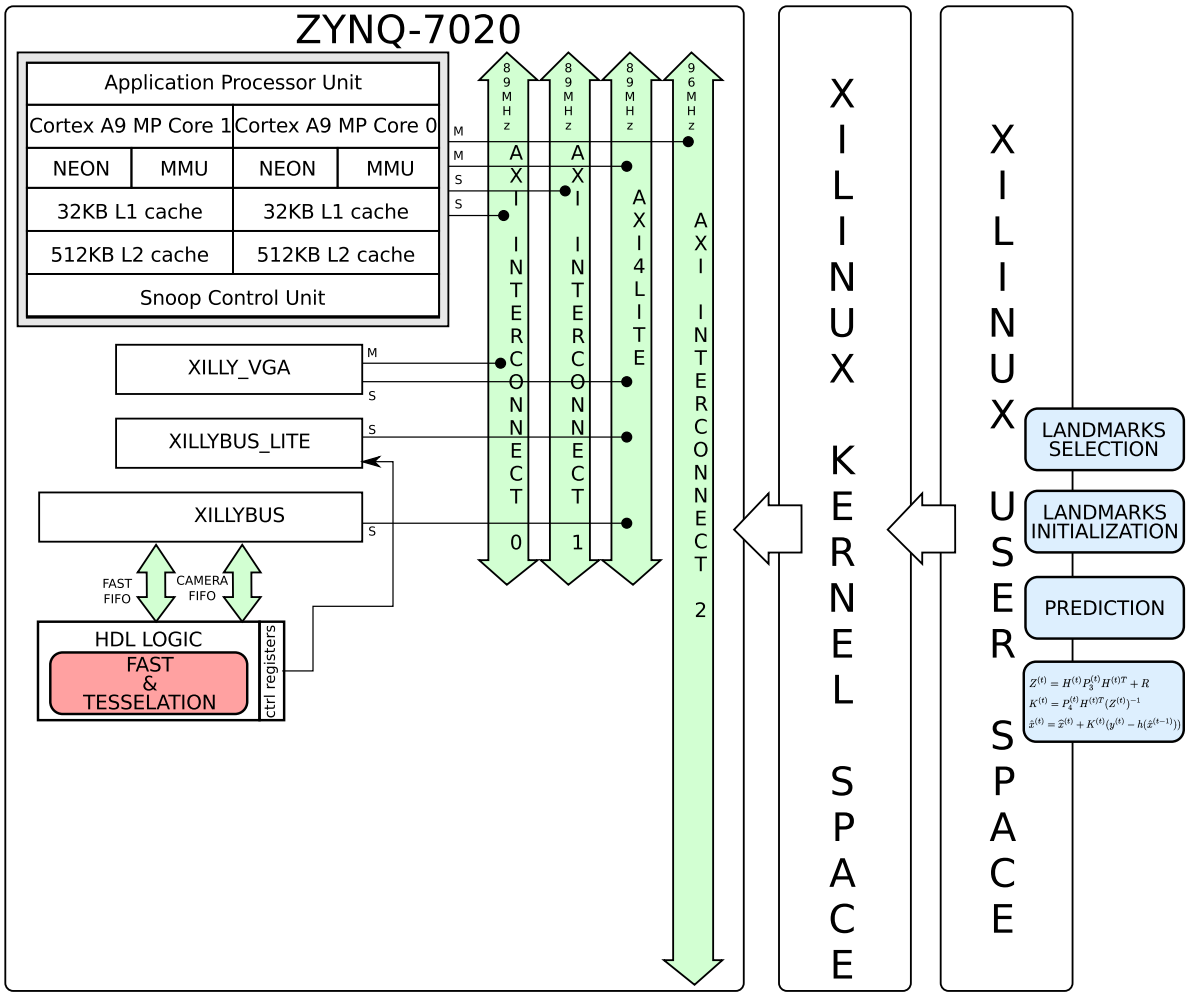

FIGURE 5.11 – Intégration de l'architecture matérielle dédiée dans la chaîne EKF-SLAM.

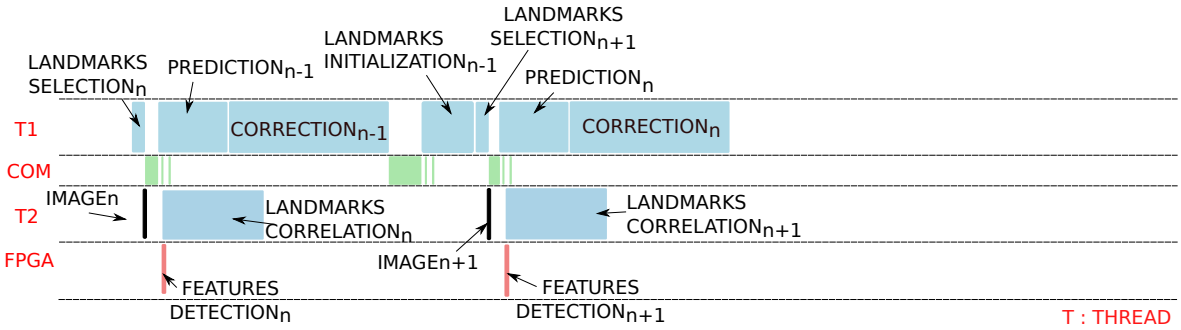

FIGURE 5.12 – Ordonnancement des tâches de l'application SLAM après la seconde itération de la méthodologie de co-design (accélération de la détection de points d'intérêt).

en utilisant la séquence d'images de référence. En revanche, nous avons constaté une nette amélioration de la fréquence de fonctionnement globale du SLAM. Le tableau 5.1.1.8 détaille les taux d'occupation processeur (ARM cortex A9) en fonction des tâches. On peut voir la nette amélioration en performance de la fonction de détection. Le diagramme 5.13 illustre cette amélioration en la comparant une exécution C-SLAM avec le détecteur Harris et FAST en logiciel avec FAST en matériel.

Itération n^o	I (HARRIS SW)	I (FAST SW)	II (FAST HW)
Taches C-SLAM	Taux d'occupation CPU [%]		
Communication	5.97	7.45	17.83
Sélection des amers	0.4	0.5	1.19
Prédiction	1.04	1.3	3.12
Correction	11.94	14.91	35.66
Initialisation	0.15	0.19	0.45
Caméra	6.47	8.07	19.32
Corrélation	4.38	5.47	13.08
Détection de coins	69.65	62.11	9.36
Taux d'img/s [Hz]	5	6.2	14.9

TABLE 5.4 – Taux d'occupation CPU des tâches de C-SLAM.

Ces résultats encourageants ne sont pas suffisants puisque notre système ne satisfait toujours pas les contraintes temps réel que nous nous sommes fixées (>30Hz). La méthodologie de co-design a donc été réitérée afin d'accélérer la fonction de correction (présentée en section 4.2) (Travaux effectués par Daniel Torteï et présentés dans [Tertei et al., 2014]).

5.2 Politique de répartition des points d'intérêt dans l'image

Cette section présente une méthode de répartition des points d'intérêt dans l'image, la tessellation, qui a été conçue au moment de l'implantation matérielle de l'algorithme FAST (décrite dans le paragraphe précédent, 5.1). Puis, nous exposerons une étude réalisée en parallèle du prototypage du SLAM sur un accélérateur matériel de Non-Maxima Suppression (NMS) ou de raffinement de points d'intérêt en section 5.2.2.

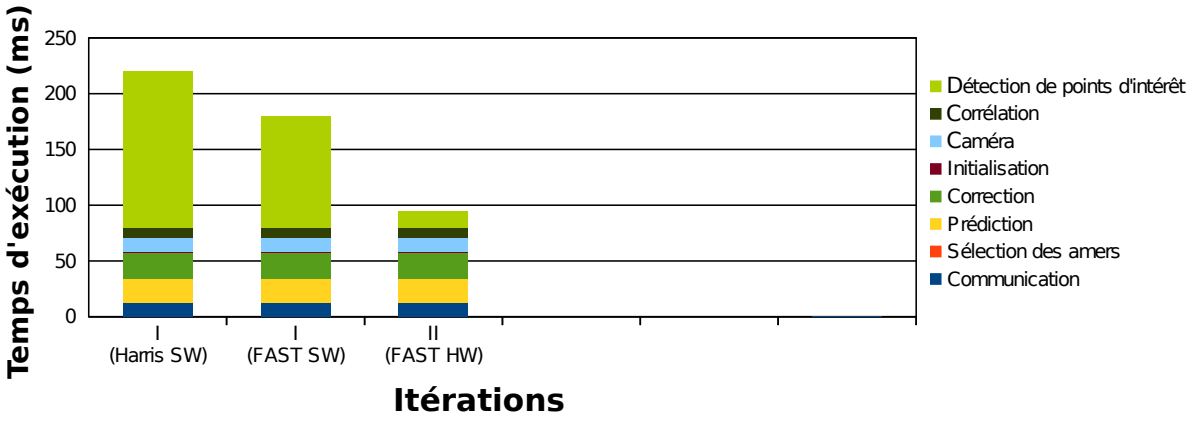

FIGURE 5.13 – Comparaison du temps d'exécution de C-SLAM purement logiciel et C-SLAM avec l'accélérateur matériel de détection FAST.

Afin de maximiser l'observabilité de l'état du robot et, ainsi, rendre le système le plus robuste possible, il est préférable de répartir correctement les points d'intérêt dans l'image. En effet, dans le cas d'une scène avec, dans le champ de vision, l'horizon et le sol, l'observation d'amers à l'horizon donne une bonne information sur la rotation du mobile. Les amers sur le sol eux renseignent sur la translation. Il faut donc détecter des points d'intérêt régulièrement répartis dans l'image pour offrir au SLAM une meilleure observation de la scène et du mouvement. Ceci est d'autant plus important que la quantité d'amers observables est limitée par les capacités de calcul. Pour cela nous allons exposer deux stratégies : la tessellation (implantée dans le SLAM) et la Non-Maxima suppression (NMS).

5.2.1 Tessellation

5.2.1.1 Principe

La tessellation est une méthode de partitionement d'un espace bi- ou tridimensionnel. Dans notre cas, nous divisons l'image en plusieurs portions d'image. Dans le cadre d'une détection de points d'intérêt, ceci permet de sélectionner une région, puis, d'y détecter le meilleur coin (celui qui a le meilleur score). C'est la stratégie adoptée dans C-SLAM pour limiter la demande en calcul de la fonction *détection*.

Cette division offre deux avantages :

— En segmentant ainsi l'image, nous obtenons une grille d'occupation mémorisant les projections dans l'image des amers déjà présents dans la carte. Ainsi, nous connaissons les secteurs de l'image des amers suivis et nous pouvons sélectionner une région qui en est dépourvue pour l'initialisation d'amers (cf. figure 5.14) ;

— Le second avantage est que la détection d'un point d'intérêt dans une région sera beaucoup plus rapide que le traitement de l'image en entier.

Même si, comme nous le verrons dans la suite de cette section, l'économie de calcul n'est plus nécessaire dans le cadre d'une détection matérielle dédiée (les points d'intérêt sont recherchés dans toutes les cases), la répartition des amers pour une meilleure observabilité reste vraie. Elle permet, de plus, de limiter le nombre d'informations qui transitent entre le CPU et la logique FPGA. Ce nombre dépend de la finesse de la grille choisie, dans notre cas 12x16

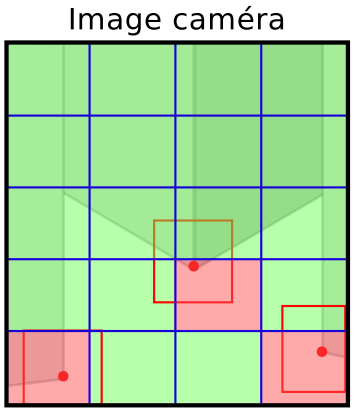

FIGURE 5.14 – Exemple de tesselation de l'image permettant la répartition des points d'intérêt dans l'image. Les cases rouges sont occupées, un point d'intérêt est déjà observé dans cette région et les cases vertes sont disponibles pour l'initialisation de nouveaux points.

cases donc 192 points d'intérêt maximum.

5.2.1.2 Conception matérielle

Comme pour le bloc FAST, les contraintes unitaires de ce bloc matériel de tessellation sont fixées à (cf. section 5.1.1.2) : une fréquence de traitement **supérieure** 9.216**MHz** et une latence inférieure à 19 lignes.

En matériel, la tessellation a été connectée directement à la sortie du détecteur de coin FAST. Elle stocke en mémoire BRAM le meilleur score (sa position et son score) dans chaque case de la grille. Ce traitement se fait sur un flux de points d'intérêt FAST. Deux compteurs de lignes et colonnes permettent de savoir à quelle case appartient chaque coin reçu. Un seul point d'intérêt est conservé par case (celui qui a le meilleur score). Ces maximum sont stockés en BRAM et mis à jour au fur et à mesure du parcours de l'image. La consommation mémoire, (Mem), de ce module de tessellation dépend :

— du nombre de colonnes de la grille, $Cols$;
— du nombre de bits nécessaires pour coder les positions (x et y) des points d'intérêt, $nbrbitY/nbrbitX$;
— du nombre de bits requis pour coder le score, $nbrbitscore$.

L'équation 5.2 permet de calculer la consommation mémoire :

$$Mem = Cols * (nbrbitY + nbrbitX + nbrbitscore) \tag{5.2}$$

Dans notre cas, $Cols = 16$, $nbrbitY = 10$ bits, $nbrbitX = 10$ bits and $nbrbitscore = 12$ bits. La tessellation matérielle ne requière donc que $Mem = 512$ bits.

Nous avons instancié deux compteurs afin de savoir à quelle case le coin reçu appartient. Cette position permet aussi d'interdire la sélection de points d'intérêt près des bords de l'image. Ceci évite de sélectionner des amers qui ne peuvent pas être décrits par le descripteur (présenté en section 5.3) si les pixels voisins sont en dehors de l'image. Dans l'architecture cette taille

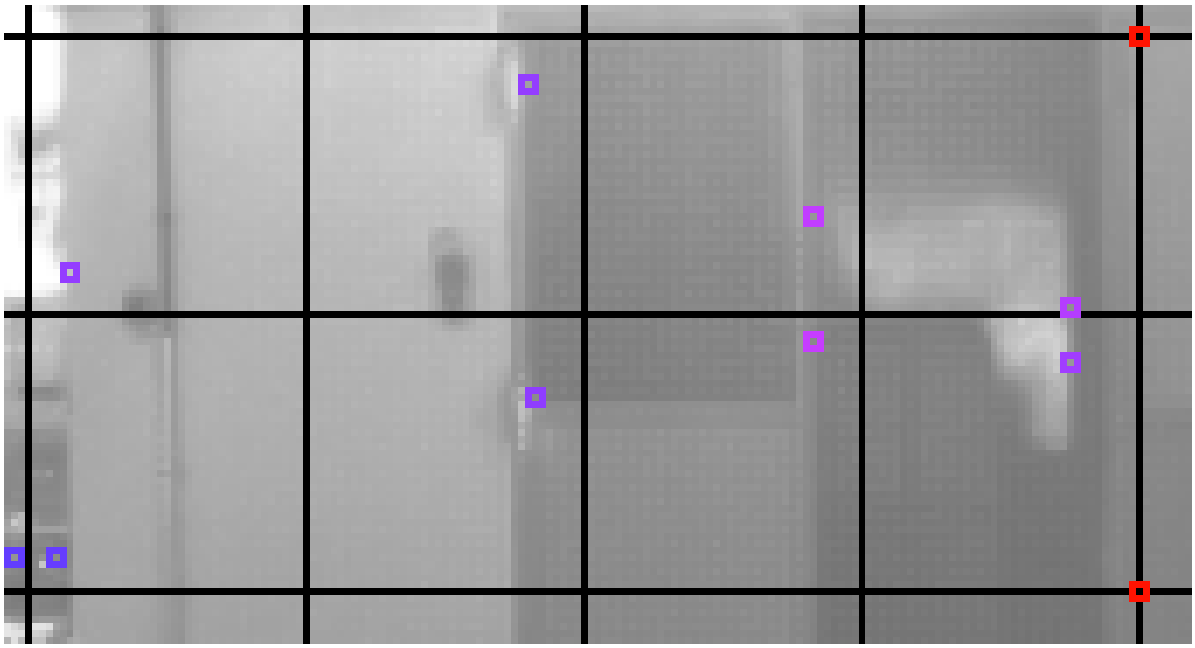

FIGURE 5.15 – Méthode de tessellation sur les résultats d'un détecteur de points d'intérêt FAST sur architecture matérielle.

est paramétrable (paramétré à 4 pixels dans notre cas [5]). La Figure 5.15 présente une portion d'image où l'on peut voir la tessellation (grille) en noir. Dans cette dernière, nous avons choisi un dégradé de couleurs représentant la valeur du score des coins détectés. De bleu pour un très bon score (un très bon point d'intérêt) à rouge pour une absence de détection.

Cette chaîne produit un nombre fixe de données, $Data$, dépendant des paramètres suivants :

— le nombre de colonnes de la tessellation, $Rows$;
— le nombre de lignes de la tessellation, $Cols$;
— le nombre de bit nécessaires pour coder les positions (x et y) des points d'intérêt, $nbrbitY/nbrbitX$;
— le nombre de bit requis pour coder le score, $nbrbitscore$.

$$Data = (Cols * Rows) * (nbrbitY + nbrbitX + nbrbitscore) \tag{5.3}$$

Dans notre configuration, $Cols = 16$, $Rows = 12$, $nbrbitY = 10$ bits, $nbrbitX = 10$ bits et $nbrbitscore = 12$ bits. Cette chaîne retourne toujours au microprocesseur $Data = 6144$ bits de données.

5.2.1.3 Validation

Nous avons développé le bloc de tesselation en VHDL afin de l'implanter. Pour cela, nous avons validé le code VHDL en simulation grâce au banc d'essai que présenté sur la figure 5.16. Dans ce dernier, un module VHDL *Camera* permet de simuler les signaux de synchronisation

5. Notre descripteur BRIEF nécessite une fenêtre de voisinage de 7×7 pixels, centrée sur le pixel traité, cf. section 5.3.

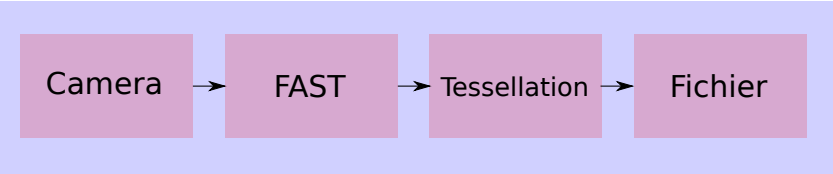

FIGURE 5.16 – Banc d'essai de simulation de l'architecture matérielle FAST et tessellation.

(hsync, vsync et pclk) et données (pixel) d'une caméra à partir d'une image stockée sur disque. Un deuxième composant *Fichier* permet de collecter, mettre en forme les résultats de FAST associé au bloc de tesselation et les stocker dans un fichier.

Une fois cette validation par la simulation effectuée, nous avons implanté cette chaîne FAST et tesselation sur FPGA (ZynQ® 7020). Pour cela, nous avons utilisé la distribution Xillinux. L'intégration est présentée dans la figure 5.11. Afin de valider cette chaîne, nous avons tout d'abord utilisé une image de synthèse (fond blanc avec des formes rectangles noirs) afin de tester les frontières des différentes cases de l'images puis nous l'avons testé avec une image réelle afin de valider fonctionnellement son comportement. Nous nous sommes de plus assuré que le coin de chaque case était bien celui qui avait le score le plus grand et que sa position était exacte. En terme de performances, l'association FAST et tessellation présente une fréqence de fonctionnement de 160MHz et utilise 0.04% de LUTs, 0.2 de registres et 0 BRAMs de plus que l'architecture FAST seule. La latence reste identique : 7 lignes. Les contraintes unitaire de fréquence de traitement et de latence sont toujours respectées.

5.2.2 Non Maxima Suppression (NMS)

Nous avons mis au point une méthode de suppression de non-maxima locaux (ou NMS pour Non-Maxima Suppression). Partant du constat que tous les détecteurs basiques de coins comme Harris ou FAST retournent des amas (régions) de points d'intéret autour des coins de l'image, il peut être intéressant de les raffiner. En logiciel, toute l'image étant disponible en mémoire, ce genre de problème peut être assez facilement résolu en explorant chaque région et en y sélectionnant le meilleur score. D'autres techniques peuvent raffiner la position d'un point d'intérêt au niveau sub-pixel en extrapolant les scores des pixels(de chaque région). En FPGA, cette fonction est plus complexe car l'exploration des amas de scores doit se faire sur un flux de données. Le problème est d'autant plus difficile à résoudre que la taille des régions de scores est variable. Nous avons élaboré une technique originale de raffinement des coins sur architecture matérielle résolvant ce problème. Ce travail a été présenté dans [Brenot et al., 2015].

5.2.2.1 Principe

Notre algorithme de Non-Maxima Suppression (NMS) peut explorer les amas de pixels candidats en prenant en compte un voisinage de 4 ou 8 pixels connexes. La figure 5.17, illustre ces deux configurations. En 4 pixels connexes, les régions 1 et 2 sont distinctes contrairement au cas de la configuration 8 pixels connexes.

Dans la suite de ce document, nous allons présenter la configuration 4 connexes. Notre accélérateur de NMS réalise un traitement de type pipeline où les pixels (ou points d'intérêt

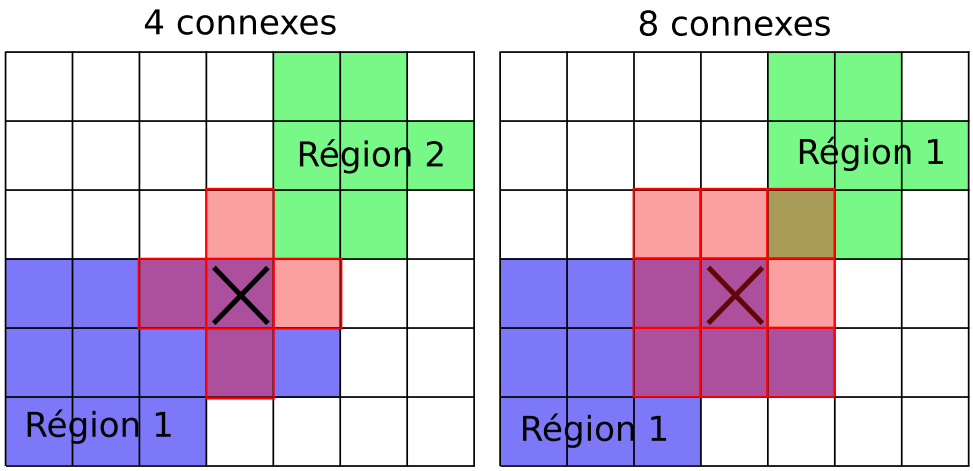

FIGURE 5.17 – Configurations du Non-Maxima Suppression en 4 et 8 pixels connexes.

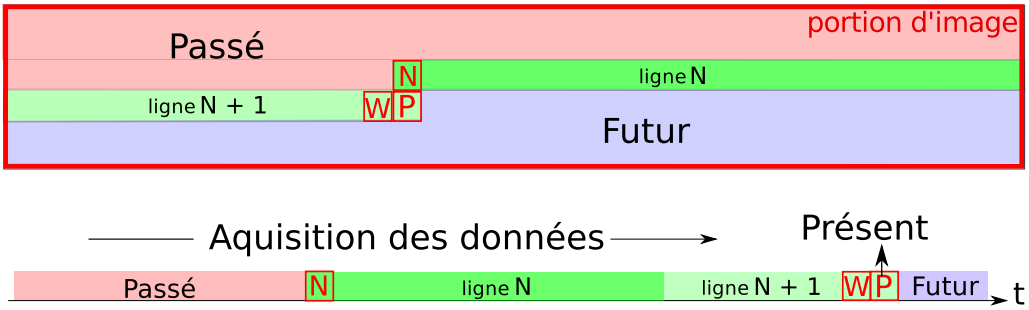

FIGURE 5.18 – Traitement des pixels en flux avec le pattern NMS.

dans notre cas) sont acquis et traités les uns à la suite des autres. La figure 5.18 illustre ce procédé utilisant un pattern en forme de "L". Dans cette figure, P est le pixel traité qui vient d'être acquis, W est son voisin de gauche (W pour West) et N le voisin du dessus (N pour North).

Nous verrons par la suite que les lignes N et $N+1$ ne sont pas un simple buffer de pixels (ou de scores de points d'intérêt) mais qu'elles contiennent en réalité des informations de plus haut niveau sur les régions en cours d'exploration.

5.2.2.2 Conception

Notre algorithme NMS pipeline est basé sur une machine à états présentée en figure 5.19. Dans cette dernière, l'état *Background* correspond à une absence de région. Lorsqu'un nouveau pixel est disponible, trois cas principaux peuvent être identifiés, soit :

— une nouvelle région (*New region*), dans ce cas une nouvelle allocation de mémoire est créée afin de stocker le score maximal de cette région ;

— une région déjà connue, (*Known region*), (définie comme une région non-fermée ou une région ouverte), dans ce cas le score maximum de la ligne précédente (N) appartenant à cette zone est lu en mémoire et est mis à jour si besoin ;

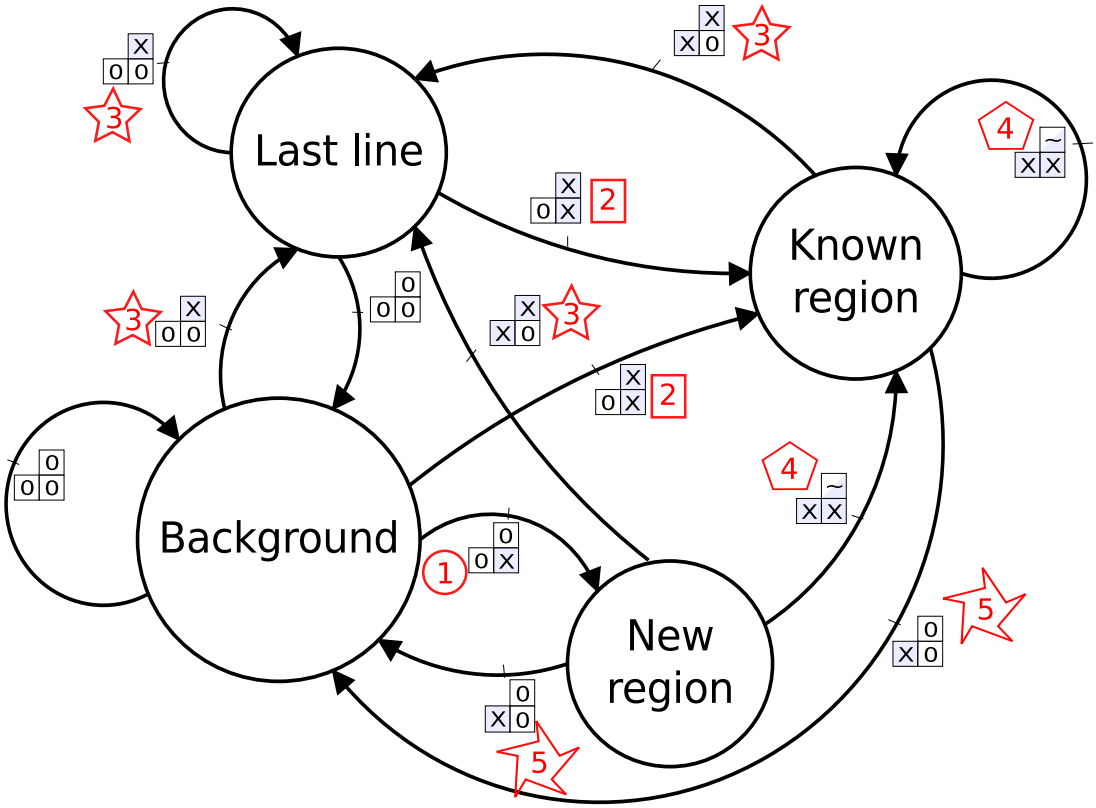

FIGURE 5.19 – Machine à état de l'algorithme de Non-Maxima Suppression.

— une région connue est fermée (*Last line*), dans ce cas le score maximum et définitif de cette zone est trouvé.

La figure 5.20 montre une vue d'ensemble de l'architecture proposée. On y trouve deux blocs, un *memory manager* et un *maximum comparator*, que nous allons maintenant décrire.

Maximum comparator Ce bloc permet de détecter les régions de points d'intérêt candidats. Ce traitement met en oeuvre un pattern. Ce dernier permet d'explorer les scores de pixel des régions pour y trouver le maximum. Ce pattern prend en compte 3 pixels. Il est représenté à la figure 5.21 où N est le score maximum trouvé dans les lignes précédentes de la région courante, P est le score courant envoyé par FAST et W est le score maximal connu de l'ensemble de la région (y compris les lignes précédentes). Ces scores sont comparés et le maximum est mis à jour dans W.

Cinq cas principaux déterminent les conditions de transition d'état (se référer aux chiffres en rouge dans la figure 5.19 et la notation en figure 5.21). Ces transitions sont basées sur l'état du pattern :

1. $P \neq 0$ ($P = X$), P a un score non nul. Cela signifie que le score courant fait partie d'une nouvelle région. Si, dans le futur, il apparait que cette région est déjà connue, les deux régions seront fusionnées ;

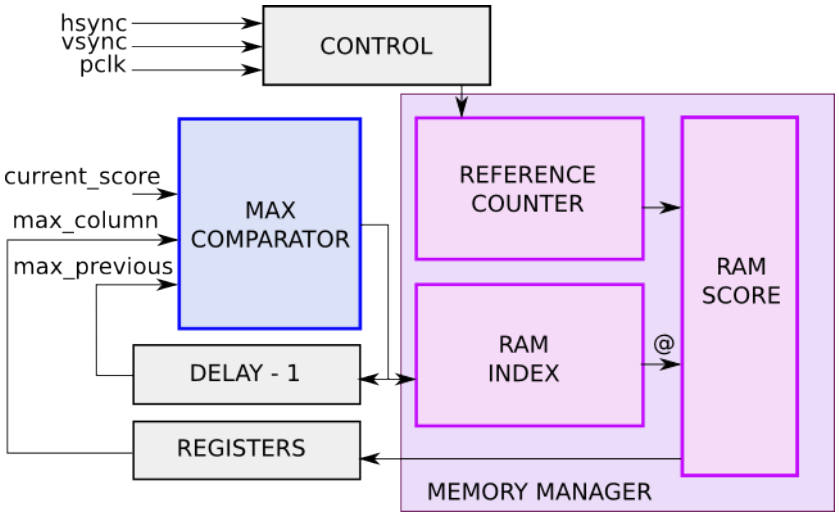

FIGURE 5.20 – Architecture matérielle de l'algorithme de Non-Maxima Suppression.

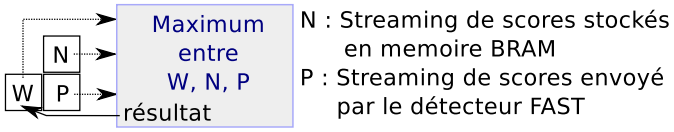

FIGURE 5.21 – Pattern utilisé pour une configuration 4 connexes de la Non-Maxima Suppression.

2. Si $N \neq 0$ et $P \neq 0$, on peut supposer que la région a déjà été découverte (ouverte) dans les précédentes lignes. Le score le plus élevé entre N et P est mis à jour dans W. En parallèle, un compteur de référence permet de garder en mémoire la largeur en pixels de cette zone ;
3. Le pattern est entrain d'explorer la ligne en dessous de la zone. On dit qu'il est entrain de "fermer" la zone. Pendant ce processus, le compteur de référence est décrémenté afin de mettre à jour la taille en largeur de la zone (à chaque fois que $N \neq 0$ et $P = 0$). Si ce compteur tombe à '0', la région a été entièrement explorée, elle peut être fermée et le score maximum de cette dernière relâché ;
4. Cet état montre que le pattern est dans une région connue (donc ouverte). Si N et C appartiennent à deux régions différentes, les régions sont fusionnées pour n'en faire qu'une ;
5. Cette condition signifie que le score maximum contenu dans W peut être mis à jour en mémoire BRAM.

Memory manager Afin d'expliquer le bloc *memory manager*, prenons $Img_{size} = H \times L$ la taille de l'image avec sa hauteur H et sa largeur L. Les coordonnées des scores dans l'image sont données par (l, c) avec l pour la ligne et c pour la colonne. $P(l, c)$ correspond à la position du score courant P.

Nous avons choisi de stocker des données afin de garder une trace du passé. Premièrement, le module *memory manager* utilise une mémoire (BRAM), RAM_{score} (cf. figure 5.20) qui

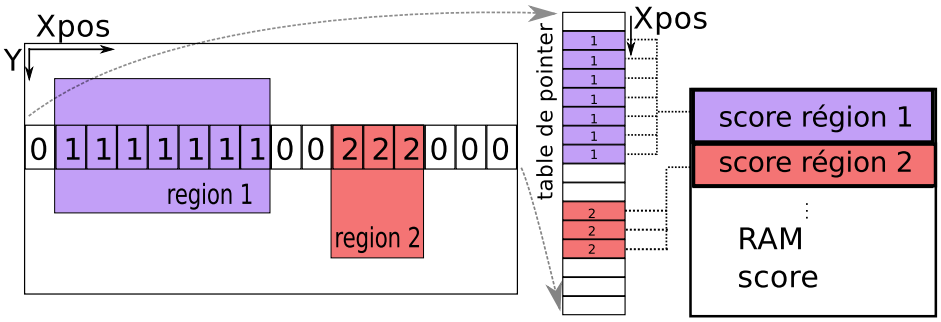

FIGURE 5.22 – Gestion des mémoires dans Non-Maxima Suppression.

FIGURE 5.23 – Configuration du pattern de la Non-Maxima Suppression.

stocke le score le plus élevé de chaque région ouverte. Pour chacun de ces scores, nous stockons leurs positions (l, c). De plus, pour chaque score, un label est stocké dans une seconde mémoire en fonction de la région à laquelle il appartient. Nous avons appelé cette mémoire, RAM_{index}. Une troisième mémoire $ref\_counter$ stocke la taille en largeur de chaque région, c'est un compteur de référence.

La mémoire RAM_{index} (figure 5.20) stocke une ligne entière de pointeurs comme on peut le voir dans la figure 5.22. Les numéros de colonnes de l'image sont les adresses de la mémoire afin de savoir à quelle région appartient chaque pixel de la ligne courante. Par exemple, dans la figure 5.22, la largeur de la région 1 est de 7 pixels et de celle la région 2 est de 3 pixels de large sur la ligne en cours. On a donc en mémoire 7 cases mémoire qui pointent vers la région 1 de la mémoire *RAM score* et 3 vers la région 2.

Le compteur de référence ($ref\_counter$) est mis à jour pour mémoriser le nombre de points qui appartiennent à chaque région. Quand il tombe à zéro, l'allocation de mémoire dans RAM_{score} peut être libéré, la région a été entièrement explorée.

Enfin, la lecture et la mise à jour des mémoires RAM_{index} et RAM_{score} conduit à une latence. Il faut 2 cycles d'horloge pour mettre à jour un score. Afin d'être en mesure de mettre à jour et de lire un score en un cycle d'horloge, nous avons utilisé des registres pour alimenter le W du pattern afin d'anticiper les lectures mémoire. Lors de la mise à jour d'un score, la mémoire et les registres sont mis à jour.

L'architecture NMS peut être facilement configurée afin d'utiliser une connectivité de 4 ou 8 pixels. Ce changement n'affecte que le module de comparaison (et plus précisément le pattern utilisé). Pour une configuration 8 connexes le module *MAX comparator* compare le score courant P avec W, N et NE (pour North-East) comme le montre la figure 5.23.

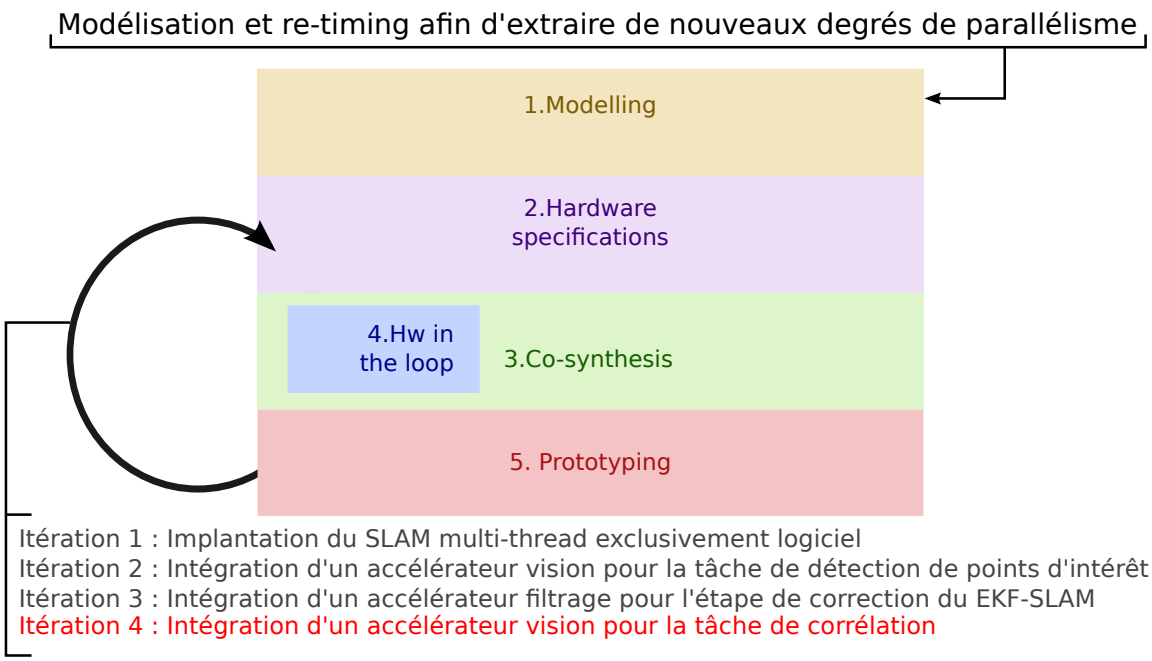

FIGURE 5.24 – Quatrième itération de méthodologie de co-design effectuée avant d'obtenir un système EKF-SLAM qui satisfait les contraintes imposées.

5.3 Descripteur de points d'intérêt BRIEF et la corrélation

Dans les sections 5.3 et 5.4, nous allons présenter le travail effectué dans la quatrième et dernière itération de la méthodologie de co-design (cf. figure 5.24). Il correspond à l'intégration d'un accélérateur matériel original de la tâche de corrélation. Pour cela nous allons intégrer tout d'abord un accélérateur de description de points d'intérêt BRIEF (en section 5.3) puis une bloc de corrélation (en section 5.4). Cette dernière intégration va permettre d'exécuter la partie vision (front-end) entièrement en matériel. Ainsi, nous pourrons connecter directement la caméra sur la logique et supprimer le transfert des pixels du CPU vers la logique.

Le descripteur BRIEF présente des propriétés intéressantes en vue de son implantation en matériel. Le calcul de ce dernier ne requiert que des comparaisons entre pixels codés sur 8 bits. De plus, la corrélation de deux descripteurs BRIEF se calcule par une distance de Hamming. En matériel, cette opération peut être effectuée par un arbre d'additionneurs. Dans cette section 5.3, nous allons présenter le principe de fonctionnement (cf. section 5.3.1), l'implantation du descripteur BRIEF en matériel (cf. section 5.3.2) et enfin la validation unitaire de ce bloc (cf. section 5.3.3).

5.3.1 Principe

BRIEF (Binary Robuste Independent Element Feature) [Calonder et al., 2010] est un descripteur binaire qui décrit la distribution des gradients dans le voisinage du pixel traité. Un patch $N \times N$ centré sur le pixel d'intérêt est extrait. M paires de pixels dans le patch sont aléatoirement sélectionnées. Pour chaque paire $p0_m$, $p1_m$ un bit est généré avec le règle suivante :

$$b_m = \begin{cases} 1 & \text{if } p0_m < p1_m \ avec \ m[0 \rightarrow M] \\ 0 \end{cases} \tag{5.4}$$

Il en résulte un descripteur binaire de longueur M. La figure 5.25 illustre la constitution du vecteur BRIEF sur un exemple simple descripteur BRIEF de longueur 8 bits.

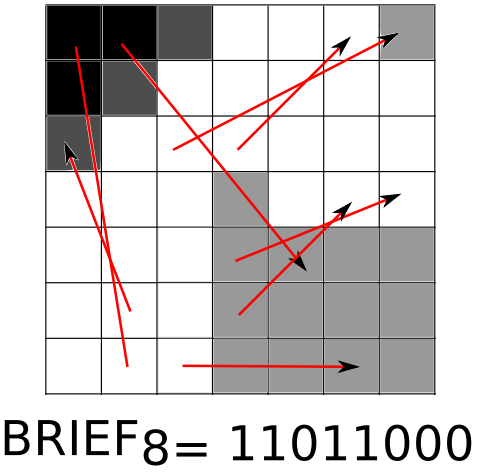

FIGURE 5.25 – Exemple de cacul d'un descripteur BRIEF 8 bits.

5.3.2 Conception matérielle

Comme pour le bloc FAST, les contraintes unitaires de ce bloc matériel de desciption BRIEF sont fixées à (cf. section 5.1.1.2) : une fréquence de traitement **supérieure** 9.216**MHz** et une latence inférieure à 19 lignes.

Nous avons, tout d'abord, instancié le descripteur BRIEF en version logiciel dans notre EKF SLAM afin de valider son comportement dans notre système. De plus, nous avons expérimenté 3 tailles de descripteurs BRIEF (64, 128 et 256 bits de longueur) ainsi que des tailles différentes de la fenêtre (ou patch) de description (fenêtres 5×5, 9×9 et 15×15) afin choisir les bons paramètres. Nous avons pu remarquer qu'une taille de fenêtre de 9×9 pixels et une longueur de descripteur de 128 bits suffisent à assurer la stabilité du tracking, tout en minimisant les ressources utilisées.

Comme pour les précédentes implantations de traitement vision (FAST, NMS, tessellation), nous avons créé une architecture capable de décrire un pixel en fonction de son voisinage, en opérant directement sur le flux de pixels envoyé par la caméra (il n'y a aucun stockage d'image mis à part un buffer de n lignes). De la même manière, le bloc de description de points d'intérêt BRIEF sera cadencé par l'horloge pixel de la caméra (indépendante de l'horloge système).

Étant donné que le système de traitement est basé sur un flux en continu de pixels, le calcul du descripteur exige de bufferiser une fenêtre de 9×9 glissante du flux de pixels d'entrée. Cette mémoire cache utilise suffisamment de mémoire pour stocker 9 lignes de l'image en BRAM et des registres pour stocker la fenêtre glissante (soit $9 \times 9 = 81$ pixels) (Figure 5.26). Cette fenêtre de 9×9 est utilisée pour le calcul de BRIEF avec un pattern de comparaisons statique (généré

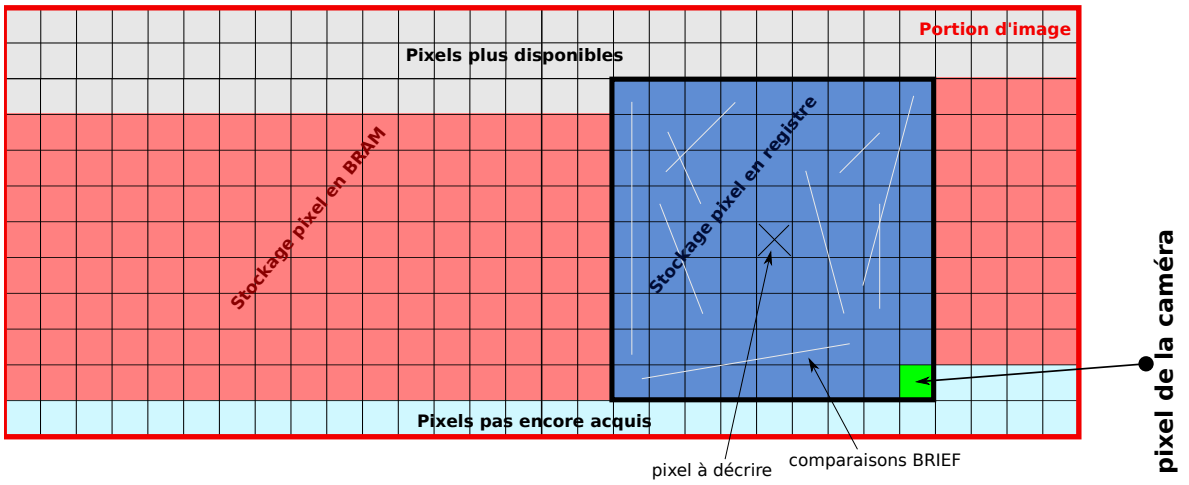

FIGURE 5.26 – Gestion du flux de pixel dans l'architecture du descripteur de points d'intérêt BRIEF. Le buffer rouge permet d'indiquer les pixels stockés en BRAM et le buffer bleu indique les pixels stockés en registre.

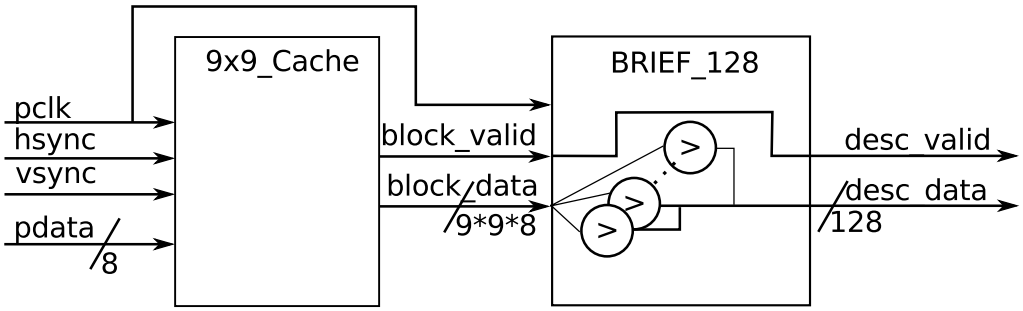

FIGURE 5.27 – Architecture d'extraction du descripteur BRIEF 128 bit.

hors ligne) de 128 paires de pixels (cf. figure 5.27). Ce modèle (ou pattern) de comparaison étant généré de manière statique, le synthétiseur VHDL peut optimiser la logique de comparaison.

Le descripteur est constamment calculé à la vitesse du flux de pixels. A chaque fois qu'un coin est détecté (par le détecteur FAST dans notre système), son descripteur BRIEF lui est associé en mémoire. Il est intéressant de noter que la même fenêtre 9×9 pixels peut être utilisée pour la détection de points d'intérêt (FAST utilise une fenêtre 7×7 pixels). La fenêtre glissante peut donc être partagée entre les deux fonctions pour économiser de la mémoire BRAM et des registres.

5.3.3 Validation unitaire

Après l'avoir implantée, nous avons validé notre chaîne FAST + BRIEF matérielle en la comparant à l'implantation logicielle. Pour cela nous avons modifié l'algorithme logiciel BRIEF afin de rendre le pattern de comparaisons de BRIEF statique (de base ce pattern est tiré aléatoirement à la première exécution de la fonction de description). Nous avons pu donc rentrer le même pattern de comparaisons BRIEF dans la fonction logiciel et dans le bloc matériel afin de pouvoir comparer les résultats. Nous avons obtenu strictement les mêmes vecteurs

BRIEF entre l'implantation logicielle et matérielle. En revanche, nous avons pu constater la rapidité de calcul de l'algorithme en logique capable de fonctionner à une fréquence maximale de 140 Mhz soit un traitement d'image de résolution HD (1920×1080 pixels) à un taux de 60 images/seconde. Une fois cette validation faite, nous avons implanté la corrélation de descripteur BRIEF matériel avant d'intégrer la chaîne vision complète (FAST, BRIEF et corrélation) dans notre SLAM.

5.4 Conception d'un accélérateur pour la mise en correspondance de points d'intérêt

Dans cette section, nous allons décrire l'ultime étape de la quatrième itération de la méthodologie de co-design, l'intégration de la corrélation de points d'intérêt décrits par descripteurs BRIEF (cf. section 5.3) - [Brenot et al., 2016]. Nous allons décrire le principe de la recherche active (cf. section 5.4.1), la conception matérielle du bloc de corrélation (cf. section 5.4.2 et 5.4.3), sa validation unitaire (cf. section 5.4.4) et son intégration dans la chaîne SLAM (cf. section 5.4.5). Enfin, nous terminerons par la validation de la chaîne complète BRIEF + corrélation dans notre application SLAM en présentant les performances du protype final de cette implantation de chaîne 3D EKF SLAM (cf. section 5.4.6).

5.4.1 Recherche active

Dans le processus de correction du SLAM, certains amers sont sélectionnés afin d'être corrigés. Au moment de l'étape de prédiction on estime la position de ces derniers dans l'image (on obtient des points d'intérêt). Ces points d'intérêt sont mis en correspondance grâce à une opération de recherche active (ou Active Search, cf. section 2.1.7). Cette dernière effectue une correlation de descripteurs BRIEF (cf. section 5.3) dans une zone de recherche (ou ROI [6]) donnée par la matrice covariance de l'innovation Z_i (cf. section 2.1.2, équation 2.11). Dans la figure 5.28, pour 3 amers de la carte globale, la matrice Z_i permet de déterminer la dimension des ROIs. L'objectif de la mise en correspondance est de parcourir cette ROI afin d'y retrouver le point pour lequel la ressemblance avec le point d'intérêt que l'on recherche est maximale. Cette ressemblance est donnée par la corrélation de deux descripteurs BRIEF et un calcul de la distance de Hamming donne un score de ressemblance.

5.4.2 Calcul de la distance de Hamming

La distance de Hamming est définie comme étant le nombre de bits qui diffèrent entre une paire de mots de taille N bits. Ce calcul nécessite d'abord d'effectuer un XOR bit à bit des deux mots, puis compter le nombre bit à $'1'$ dans le vecteur résultant (recensement de population). En matériel, l'opération XOR est simple puisque combinatoire. En revanche, le calcul de la distance de Hamming nécessite plus d'effort de conception afin d'être exécuté d'une manière efficace (en terme de fréquence de calcul et de consommation de ressources). Compter le nombre de $'1'$ dans un mot peut se faire de deux manières différentes :

1. Comptage séquentiel de $'1'$: balayer le mot et incrémenter un compteur chaque fois qu'un $'1'$ est détecté ;

6. Region Of Interest.

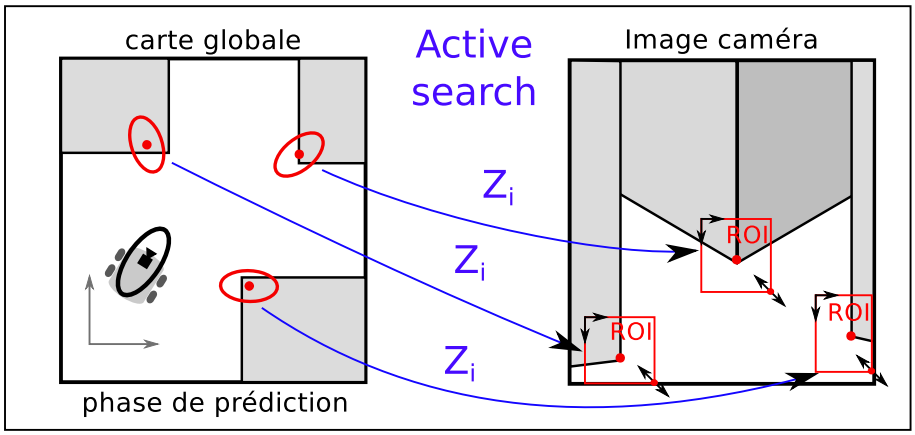

FIGURE 5.28 – La fonction active search définit la position et la dimension des ROIs en fonction des données de prédiction.

2. Comptage parallèle de $'1'$: Calculer des distances de Hamming partielles en parallèle puis les ajouter de manière itérative.

Le comptage séquentiel de $'1'$ se fait en utilisant une cascade d'additionneurs. Avec cette technique, le comptage d'un mot de N bits nécessite $N/2$ compteurs en cascade. Cette technique donne des performances de comptage en temps de traitement très faible puisque le plus long chemin combinatoire passe par N additionneurs (figure 5.29). La performance de fréquence peut être améliorée par l'intégration d'un pipeline à chaque étage de la cascade. Le calcul de la distance de Hamming d'un mot d'une longueur de 128 bits, en utilisant un comptage séquentiel de $'1'$, nécessitera 64 étages.

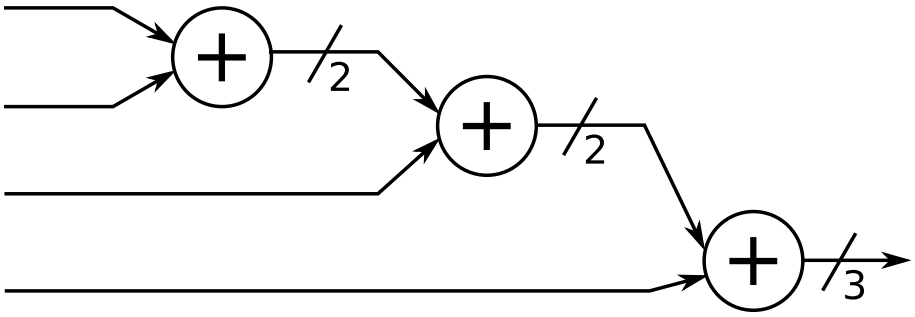

FIGURE 5.29 – Cascade d'additionneurs pour le comptage de $'1'$ séquentiel dans le calcul de la distance de Hamming.

Le comptage parallèle de $'1'$ peut être réalisé à l'aide d'un arbre d'additionneurs. Cet arbre aura $ceil(log_2(N))$ étages constitués chacun de $N/(etage_{index}+1)$ additionneurs de $2^{(etage_{index}+1)}$ de largeur (figure 5.30). L'arbre d'additionneurs peut être entièrement combinatoire avec un grand impact sur la fréquence maximale du système. Il peut aussi être pipeliné moyennant une forte consommation de ressources pour obtenir de meilleures performances en fréquence. Pour le calcul de la distance de Hamming d'un mot de longueur 128 bits, l'arbre additionneur aura donc 7 étages de profondeur.

Le nombre d'additionneurs à chaque étage a principalement un impact sur le nombre de res-

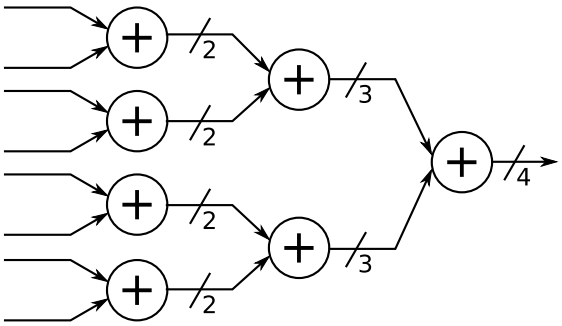

FIGURE 5.30 – Arbre d'additionneurs pour le comptage de $'1'$ parallèle dans le calcul de la distance de Hamming.

sources utilisé. La profondeur de l'arbre d'additionneurs aura un impact sur la fréquence maximale du système (et donc réduire l'horloge pixel maximale que peut supporter notre système) car elle conditionne directement la longueur maximale du chemin combinatoire. Pour implanter le calcul de la distance de Hamming sur un système piloté par l'horloge pixel (la distance doit être calculée à chaque cycle d'horloge pixel) avec des performances en fréquence maximales et une consommation en ressource minimale, nous avons fait une étude sur l'approche parallèle. Dans ce travail, nous avons exploré différentes solutions répondant au problème de comptage de bits à $'1'$ en parallèle. Ceci permet de trouver le meilleur compromis entre la fréquence de fonctionnement maximale et l'utilisation des ressources. Nous avons orienté nos recherches sur les solutions qui profitent des ressources spécifiques du FPGA : les LUTs et les blocs RAMs.

5.4.2.1 Calcul de la distance de Hamming basé LUT

la mise en œuvre la plus naïve de l'arbre d'additionneurs est d'utiliser des additionneurs de 1 bit au premier étage de l'arbre. Cela signifie que ce premier étage doit comporter $N/2$ additionneurs de 1 bit. Dans [Sklyarov et al., 2015], les auteurs proposent de mettre en œuvre la distance de Hamming en utilisant des LUT. L'addition de 2 bits peut être considérée comme une fonction combinatoire comme le montre la table de vérité suivante :

b_1	b_0	r_1	r_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Une table de vérité peut être mappée sur les MUX et LUT dans le FPGA. Les slices génériques des FPGAs contiennent une ou plusieurs LUTs avec une taille variable d'entrée, en fonction du fournisseur et le modèle du FPGA. Dans la gamme Xilinx®, les FPGAs modernes (à partir de Spartan-6) utilisent des LUTs à 6 entrées et une sortie. Cela signifie qu'au lieu d'utiliser des additionneurs à 1 bit dans le premier étage de l'arbre d'additionneurs, on peut mapper la fonction d'addition aux entrées de trois LUTs pour efficacement calculer la distance de Hamming d'un mot de 6 bits. Pour un mot de taille N bits, on consommera donc $(N/6) \times 3$ LUTs au premier étage de l'arbre et $ceil(log_2(N/6))$ étages d'additionneurs pour calculer la

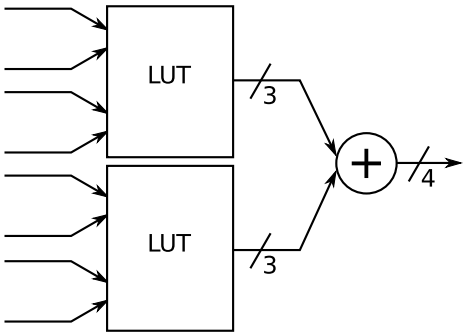

FIGURE 5.31 – Exemple du calcul de la distance de Hamming d'un mot 8 bits basé LUT.

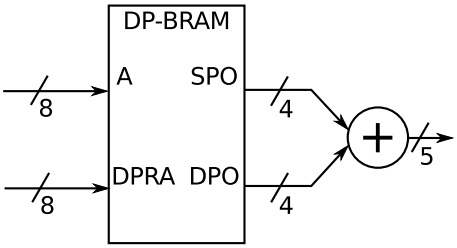

FIGURE 5.32 – Exemple du calcul de la distance de Hamming d'un mot 16 bits basé BRAMs.

distance de Hamming (Figure 5.31). La réduction de la profondeur de l'arbre d'additionneurs a un impact direct sur la fréquence maximale du processeur de distance Hamming. Pour le calcul de la distance de Hamming sur un mot de 128 bits, nous avons donc besoin d'un arbre d'additionneurs de profondeur 5 avec un premier étage composé de (3×22) LUTs à 6 entrées.

5.4.2.2 Calcul de la distance de Hamming basé BRAM

La méthode précédemment mentionnée propose de mapper le premier étage sur des LUTs à 6 entrées pour limiter la profondeur de l'arbre d'additionneurs et augmenter la fréquence maximale du système. Des BRAMs double port sont aussi des ressources habituellement disponibles dans les FPGA modernes. Ces BRAMs permettent un double accès simultané en lecture grâce à deux ports indépendants. Une BRAM étant une mémoire, elle peut également être considérée comme une grande LUT avec les entrées correspondantes à l'adresse et le contenu à la sortie de la LUT. Une BRAM avec une $Address_{width}$ et $Data_{width}$ peut être utilisée comme une LUT à $Address_{width}$ entrées et $Data_{width}$ sorties. L'utilisation d'une BRAM comme une LUT de comptage de $Address_{width}$ bits nécessitera $Data_{width} = ceil(Address_{width/2})$. Par exemple, une BRAM avec un BUS d'adresse de 8 bits devra contenir 4 bits de données. Cela signifie que nous pouvons remplacer le premier étage de notre arbre d'additionneurs avec des BRAMs (une BRAM peut être utilisée pour deux additions grâce aux doubles ports). On consommera $(N/8)/2$ BRAMs pour le premier étage, puis $ceil(log_2((N/8)/2))$ étages d'additions pour calculer la distance de Hamming (figure 5.32).

Pour le calcul de la distance de Hamming sur notre mot 128 bits nous aurons besoin d'un arbre d'additionneurs de profondeur 3 avec un premier étage composé de 8 BRAMs double port avec des bus d'adressages 8 bits et des bus de données à 4 bits.

Toutes ces méthodes décrites sont de bonnes solutions au problème de comptage de la

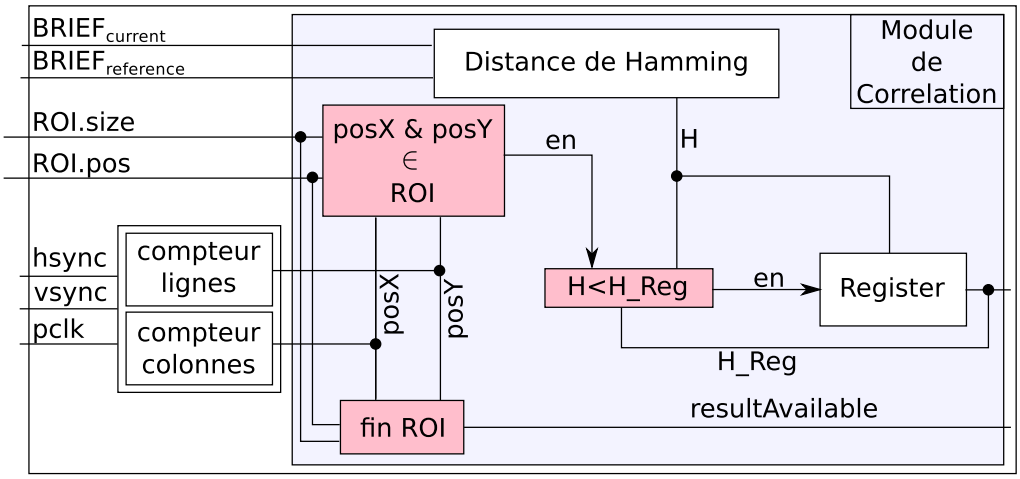

FIGURE 5.33 – Architecture d'un processeur de corrélation.

population. La mise en œuvre de LUT donne le meilleur rapport performance/ ressources, mais, au vu des ressources disponibles dans notre architecture, nous avons choisi d'utiliser la méthode basée sur des BRAMs.

5.4.3 Architecture d'un processeur de corrélation

Comme pour le bloc FAST, les contraintes unitaires de ce bloc matériel corrélation sont fixées à (cf. section 5.1.1.2) : une fréquence de traitement **supérieure** 9.216**MHz** et une latence inférieure à 19 lignes.

Dans notre EKF-SLAM, l'observation des amers est effectuée par la recherche active de descripteurs de points d'intérêt dans une séquence d'images. Cette implantation SLAM utilise une carte court terme contenant un maximum de 20 amers. L'observation de 6 points d'intérêt maximum permet de corriger leurs positions à chaque itération. Nous avons développé un accélérateur maintenant capable de suivre 20 points d'intérêt en parallèle dans la séquence d'images, permettant ainsi d'observer chaque amer de la carte. Un processeur de corrélation est chargé avec les descripteurs BRIEF ainsi que la taille et la position de la zone de recherche correspondante. Il est ensuite piloté par l'horloge pixel et déclenché en fonction de la position courante dans l'image. Une mémoire permet de garder la position et le descripteur du points d'intérêt qui a la distance de Hamming minimale. La figure 5.33 illustre l'architecture d'un coeur de corrélation.

Pour pouvoir prendre en charge la corrélation de 20 points dans l'image, nous avons dû faire face au problème de superposition de ROIs. La figure 5.34 illustre un exemple de recouvrement de ROIs. Nous avons implanté 20 fois ce processeur en parallèle afin de permettre d'explorer 20 ROI en même temps (dans le pire cas, les 20 ROIs se recouvrent dans l'image). Un scheduler gère l'activation et la dés-activation de chacun des 20 coeurs de corrélation.

5.4.4 Validation unitaire

Afin de valider l'architecture de corrélation, nous avons tout d'abord simulé le fonctionnement d'un processeur de calcul de la distance de Hamming. Le banc de test utilisé est illustré

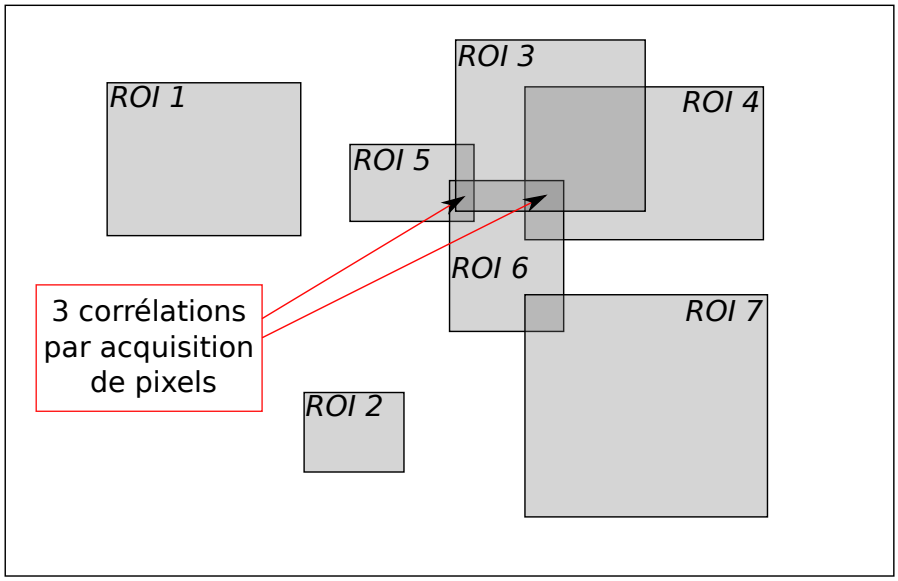

FIGURE 5.34 – Exemple de recouvrement de région d'interet

par la figure 5.35 où le module *file_reader* permet de fournir des vecteurs BRIEF à corréler et *file_writer* stocke les résultats des distances Hamming dans un fichier.

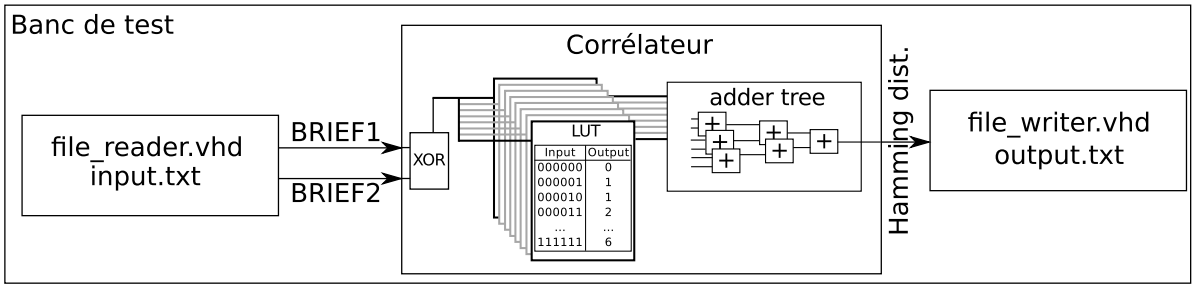

FIGURE 5.35 – Banc de test de validation en simulation d'un corrélateur

Une fois les résultats analysés et le processeur de corrélation validé, l'architecture a été dupliquée 20 fois puis implantée sur FPGA afin de procéder à une validation hardware-in-the-loop (cf. 3.1.4) à l'aide d'une architecture Xillybus. Cette validation HIL calcule la corrélation en logiciel et en matériel de 20 descripteurs et compare les résultats entre eux. La taille des zones de recherche est fixée à 10x10 pixels autour des points d'intérêt. Les modules de FAST et BRIEF ont été utilisés pour détecter et décrire les pixels. L'architecture est schématisée sur la figure 5.37. Voici le processus de validation utilisé (cf. figure 5.36) :

1. Une image de test est envoyée à la partie logique, $sendImgToHW$;
2. FAST et BRIEF sont éxécutés sur le matériel, $FAST$ & $BRIEF$;
3. Le logiciel récupère les résultats FAST et BRIEF ;
4. Une sélection aléatoire de 20 points est réalisée parmi tous les points d'intérêt obtenus, $randSelect$;

5. Les 20 corrélateurs HW sont configurés avec des zones de recherche de 20x20 autour des points d'intérêt précédemment sélectionnés. On transmet les vecteurs BRIEF de ces points d'intérêt (que l'on cherche à corréler), *setupCorrelators* ;

6. La même image de test est envoyée à la partie logique, *sendImgToHW* ;

7. La partie matérielle exécute la corrélation. Le logiciel récupère les résultats de corrélation HW, *correlation* ;

8. Le logiciel exécute la même corrélation SW (*correlSW*) et la compare à la corrélation HW, *compareHW_SW*.

La figure 5.36 met en évidence l'ordonnancement des tâches précédemment énumérées dans le temps (décrites par le pseudo-code précédent). On peut voir dans ce chronogramme la nécessité de configurer les corrélateurs matériels avant l'envoi de l'image.

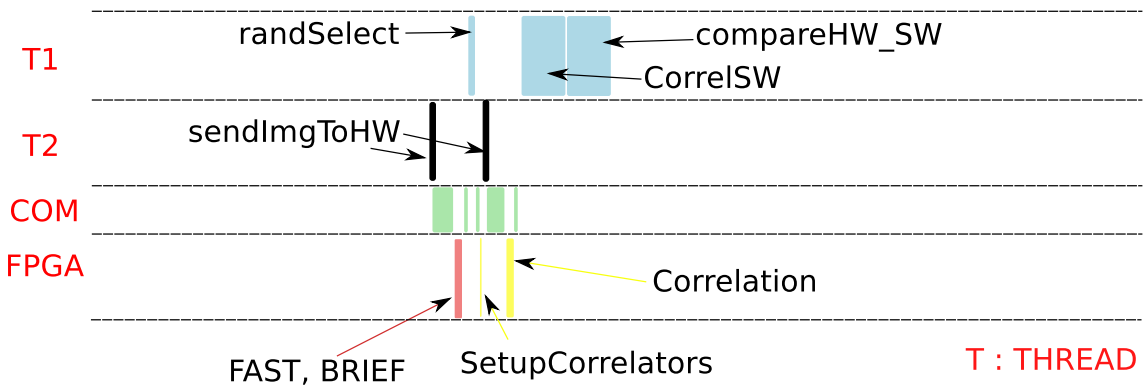

FIGURE 5.36 – Ordonnancement des tâches dans l'architecture de validation HIL du module de corrélation.

L'implantation d'un processeur de corrélation basé cascade d'additionneurs sur ZynQ® 7020 donne une utilisation équilibrée des ressources et est capable d'atteindre une fréquence de traitement maximum de 150.921MHz. Le compteur de bits basé LUT montre la meilleure performance pour une utilisation minimale de ressources avec une fréquence max de 160.436 MHz tout en réduisant de 20% l'utilisation des ressources LUTs par rapport à la méthode cascade d'additionneurs. L'approche basée BRAM montre de bonnes performances en fréquence (140MHz) et utilise 50% de LUTs en moins mais utilise un trop grand nombre de BRAMs disponibles sur le ZynQ®. De plus, la logique de série 7 de Xilinx® intègre des blocs RAM de 18kbits à 36kbits avec des largeur de données 1, 2, 4, 9, 18 bits. En utilisant ces blocs RAM avec des adresses 8 bits et 4 bits de donnée nous sous-utilisons largement chaque bloc RAM .

En augmentant le nombre de corrélateurs à 20, la fréquence n'a pas été affectée grâce à la faible utilisation des ressources. La méthode basée LUT atteint une meilleure fréquence d'horloge pixel et consomme 20% de moins de ressources que l'arbre d'additionneurs en cascade. Pour réduire au minimum l'utilisation des ressources de notre l'application SLAM, nous avons choisi d'utiliser la méthode basée LUT. Le rapport de synthèse pour les 20 corrélateurs à base LUT est présenté dans le tableau 5.5.

Type	Available	Used	Ratio
Slice Registers	106400	6850	6%
Slice LUTs	53200	9864	18%
LUT Flip Flop pairs :		12511	
-With an unused Flip Flop		5661	45%
-With an unused LUT		2647	21%
-Fully used LUT-FF pairs		4203	33%

TABLE 5.5 – Rapport de synthèse pour 20 coeurs de corrélation basés sur des LUTs.

5.4.5 Intégration dans la chaîne SLAM

Une fois la validation unitaire passée, ce bloc de 20 corrélateurs matériels a été intégré au SLAM. La figure 5.37 présente l'architecture du SLAM avec les accélérateurs de détection, description et corrélation de points d'intérêt. Notre SLAM a été modifié afin d'utiliser l'accélérateur de corrélation. Des fonctions de configuration [7] et de réception des résultats [8] de corrélation ont été intégrées et la corrélation logicielle a été supprimée. Des fonctions de mise en forme des résultats ont été ajoutées dans le logiciel pour correspondre aux structures de variables déjà existantes du SLAM.

Le diagramme de l'exécution de notre prototype de EKF SLAM en figure 5.38 représente l'ordonnancement des données transmises entre le processeur et le FGPA et les fonctions traitées par les différentes architectures. On remarque que les fonctions de corrélation, FAST et BRIEF sont exécutées sur FPGA. Ainsi, tous les traitements bas niveau (nécessitant les données pixel) sont assurés par la partie matérielle et les données haut niveau sur microprocesseur (Landmarks selection, prediction, landmarks initialization).

5.4.6 Validation et performances de notre prototype final de chaîne complète 3D EKF SLAM embarquée

Nous avons comparé le SLAM avec la corrélation exécutée par le microprocesseur puis la corrélation accélérée sur FPGA. Le tableau 5.6 récapitule les performances atteintes par ce nouveau prototype. La figure 5.39 illustre ce gain en performance en la mettant en parallèle avec les autres prototypes du SLAM.

Nous avons donc constaté un gain en fréquence mais aussi une très faible perte de précision. Cette baisse de performance est due à la différence entre les algorithmes logiciels et matériels. La corrélation logicielle est subpixelique [9] et les mesures effectuées s'en trouvent plus précises et corrigent mieux l'état du filtre.

En revanche, une fois accéléré, le temps d'exécution de la corrélation a un coût indépendant du nombre de ROIs (max. 20) parcourues et de leurs tailles respectives (cf. figure 5.41). Cette modification permet d'éviter le phénomène d'amplification de la dérive du SLAM illustrée par la figure 5.40. En effet, nous avons remarqué qu'en utilisant la corrélation en version logicielle,

7. Remplissage de la mémoire BRAM partagée entre le HW et le SW (cf. figure 5.37) avec les descripteurs BRIEF de chaque points d'intérêt à correler et leur ROI (avec leur position et leur taille) associée.

8. Lecture des résulats dans la mémoire BRAM partagée (cf. figure 5.37).

9. Les positions X, Y des points appariés sont données en précision décimale. Elle est calculée en fonction des distances de Hamming des voisins du pixel ayant la meilleure corrélation.

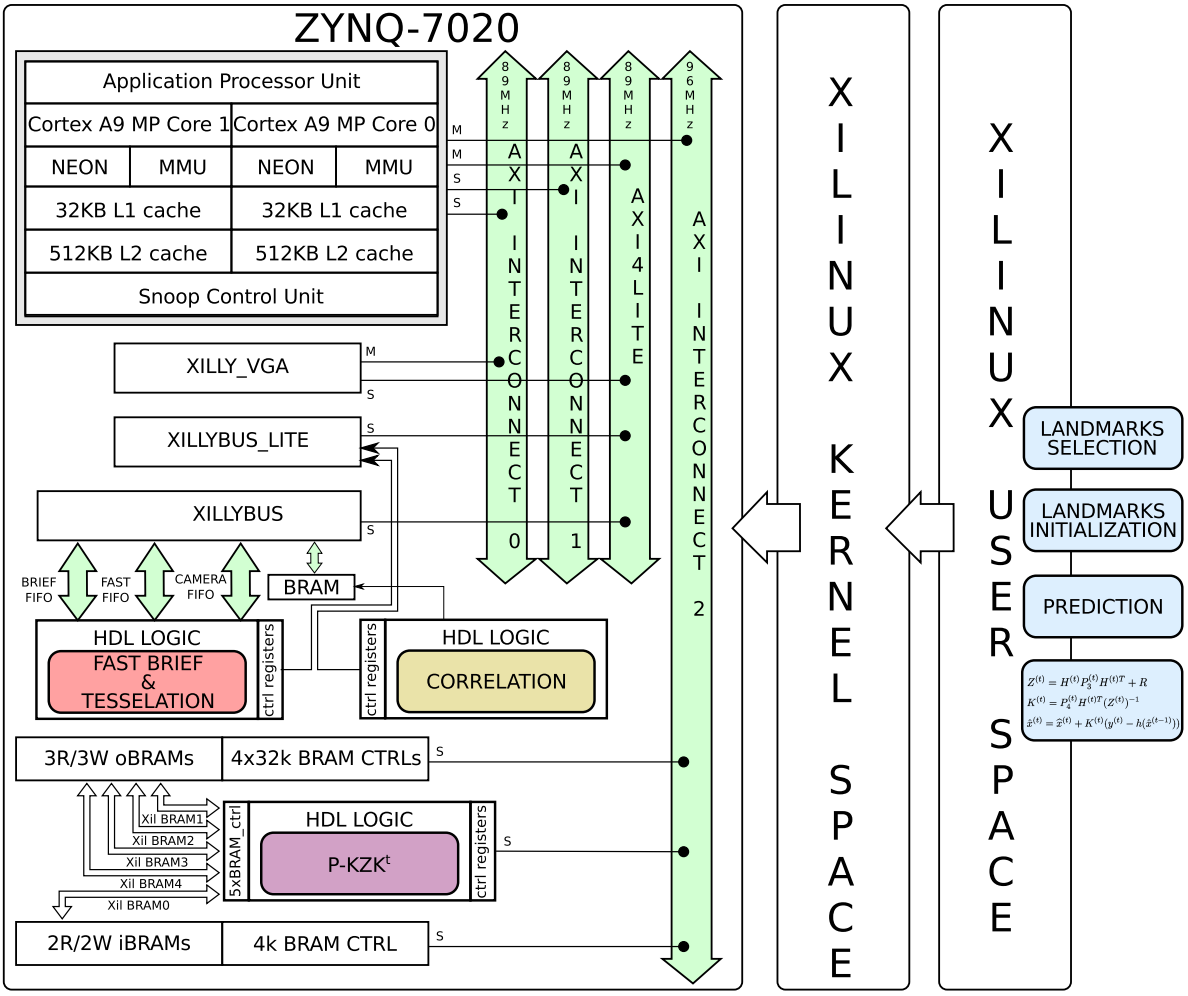

FIGURE 5.37 – Architecture du SLAM embarqué avec accélération FAST, BRIEF et corrélation sur Zynq.

quand le SLAM commence à dériver fortement, il rentre dans une boucle dans laquelle les données qu'il intègre sont de plus en plus erronées (cf. figure 5.40) :

1. les incertitudes sur la position des amers
2. la taille des zones de recherche dans lesquelles le SLAM effectue la corrélation augmentent ;
3. le temps de traitement augmente donc le taux d'img/s diminue ;
4. les différences de pose entre 2 images augmentent
5. la dérive du SLAM devient encore plus importante.

En revanche notre version matérielle permet de rendre le temps de traitement constant et indépendant du nombre de points à carréler. Nous obtenons un taux de 25 images par seconde constant même en cas de dérive.

Dans ce chapitre 5 nous avons présenté l'intégration de plusieurs accélérateurs matériels en faisant deux itérations de la méthodologie de co-design. Une première pour implanter un

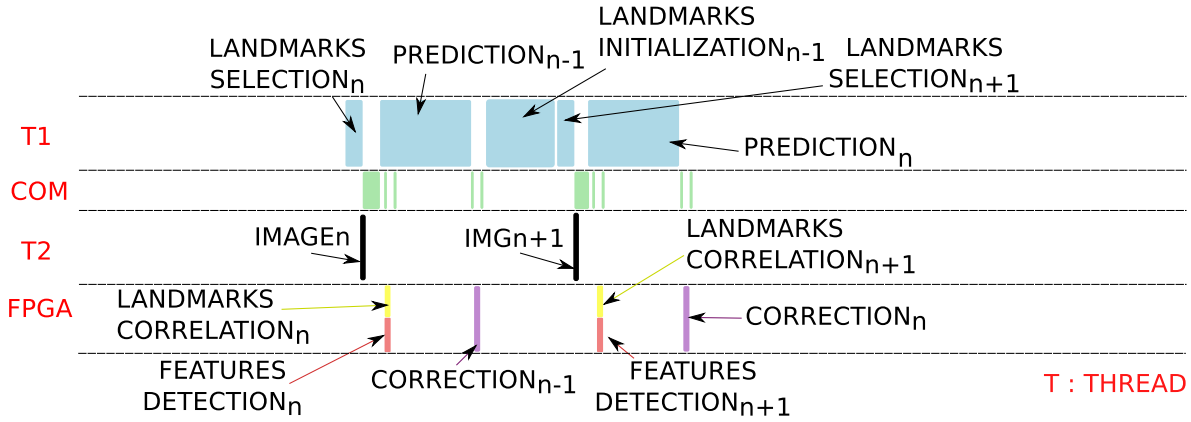

FIGURE 5.38 – Ordonnancement des données du SLAM avec l'accélération de FAST, BRIEF, tessellation et corrélation.

Itération n^o	I	II	III	IV (vision HW + $P^t - KZK^t$ en HW)
Taches C-SLAM	Taux d'occupation CPU [%]			
Communication	5.97	17.83	24.84	29.93
Sélection des amers	0.4	1.19	1.66	2
Prédiction	1.04	3.12	4.35	5.24
Correction	11.94	35.66	10.35	12.47
Initialisation	0.15	0.45	0.62	0.75
Caméra	6.47	19.32	26.92	32.42
Corrélation	4.38	13.08	18.22	1.5
Détection de coins	69.65	9.36	13.04	15.71
Taux d'img/s [Hz]	5	14.9	20.7	24.9

TABLE 5.6 – Taux d'occupation CPU des tâches de C-SLAM.

détecteur de point d'intérêt FAST (cf. section 5.1) couplé à une architecture de tessellation (cf. section 5.2) permettant de diviser l'image en cases et de retrouver le meilleur score de coins de chacune d'entre elles. Une deuxième itération nous a permis d'implanter un descripteur de points d'intérêt BRIEF (cf. section 5.3) ainsi qu'une architecture de corrélation de descripteurs de points d'intérêt (cf. section 5.4). Ces deux itérations nous ont permis d'obtenir un prototype fonctionnel capable de traiter toutes les fonctions de vision bas niveau (front-end) sur architecture matérielle. Le gain en performance est significatif puisque le système est capable de traiter 24.9 images/seconde.

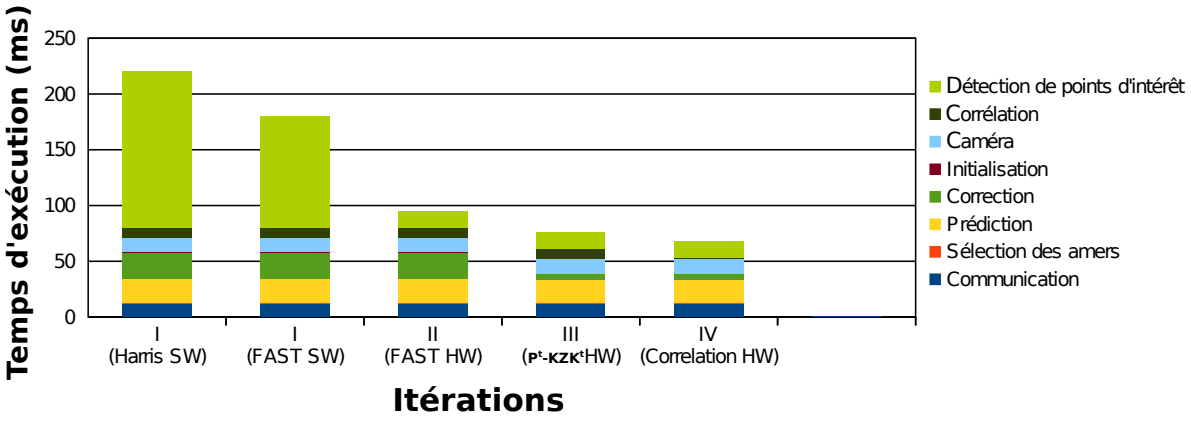

FIGURE 5.39 – Comparaison du temps d'exécution des prototypes C-SLAM précédents et C-SLAM avec l'accélérateur matériel de corrélation.

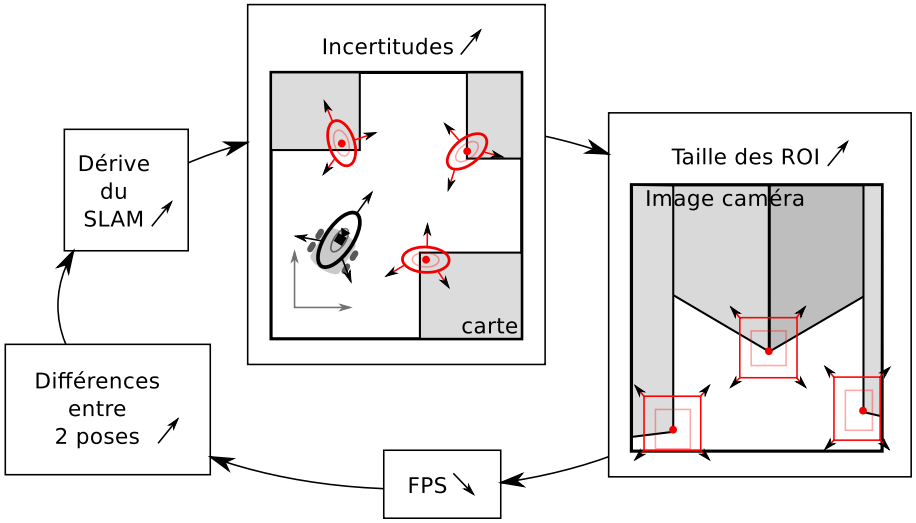

FIGURE 5.40 – Effet de l'augmentation du temps de calcul de la corrélation sur le SLAM.

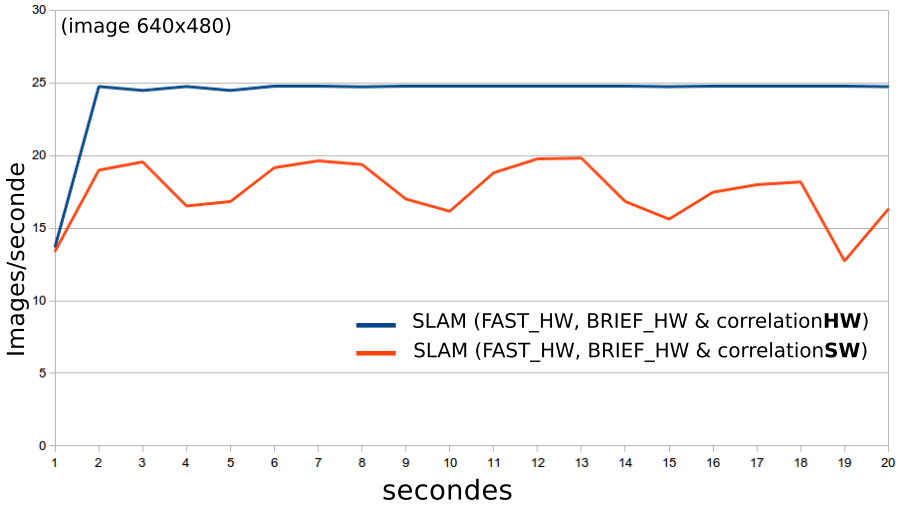

FIGURE 5.41 – Comparaison du taux d'images par seconde du SLAM avec une corrélation en logiciel et le SLAM avec une corrélation matérielle.

Chapitre 6

Conclusions et perspectives

Les constructeurs automobiles proposent de nos jours de plus en plus de systèmes avancés d'assistances au conducteur. Ces systèmes se sont multipliés ces dernières années grâce à la diminution du coût de la puissance de calcul mais aussi du fait de la diminution du coût de certains capteurs. L'amélioration en performances et la baisse du coût des caméras promettent des progrès significatifs du fait de la richesse de cette source d'information. Cependant, peu d'assistances usent de caméras car la complexité calculatoire qu'implique le traitement des images se heurte aux contraintes de temps réel du monde automobile. Dans cette thèse, nous présentons la définition et l'implantation d'une chaine 3D EKF SLAM opérationnelle sur une architecture embarquée hétérogène (processeur + FPGA) originale, avec une haute cadence et une faible latence de traitement. Les informations disponibles en sortie de l'architecture doivent permettre la conception d'assistances à la conduite nécessitant une localisation à court terme précise et une estimation fiable de l'état (vitesse, attitude) du véhicule. Ces assistances vont de la détection d'obstacles mobiles sans a priori d'apparence, à l'autonomie partielle ou complète du véhicule. L'enveloppe énergétique (5W) associée à notre prototype est de plus parfaitement compatible avec le contexte automobile. Notre architecture tire parti d'accélérateurs spécifiques aux opérations de perception du SLAM que sont la détection, la description et la mise en correspondance de points d'intérêt (ou coins) dans une séquence d'images.

Le premier accélérateur proposé permet la détection de coins en utilisant la méthode FAST. Cette détection de coins est utilisée dans nombre de méthodes d'odométrie visuelle ou de flot optique épars. L'implantation proposée s'avère meilleure que l'état de l'art, tant en ressources matérielles utilisées qu'en cadence de traitement. La fréquence maximum de fonctionnement atteinte permet d'envisager l'utilisation d'une caméra rapide (plusieurs centaines d'images par seconde) ou de haute résolution.

Le deuxième accélérateur implante l'extraction du descripteur BRIEF associé à la détection d'un coin. Ce descripteur binaire du voisinage d'un pixel d'intérêt présente à ce jour un très bon compromis entre la puissance nécessaire au calcul du descripteur et sa robustesse. D'autres travaux que les nôtres ont traité de l'accéleration de l'extraction de descripteur SIFT dont le calcul en logiciel est très coûteux mais qui présente d'excellentes performances (robustesse). Cependant ce descripteur nécessite une quantité de ressources importantes. L'implémentation du descripteur BRIEF proposée ici nécessite très peu de resources et permet le calcul du descripteur directement sur le flux de pixel (l'horloge pixel cadençant l'accélérateur).

Le troisième accélérateur se charge de corréler une liste de descripteurs BRIEF dans le flux d'images. Notre implémentation originale instancie 20 coeurs de corrélation en parallèle pour permettre l'observation de la totalité des amers de la carte de notre EKF-SLAM. Chacun des coeurs de corrélation implante le calcul de la distance de Hamming entre deux descripteurs binaires BRIEF. Nous avons étudié plusieurs implantations de la distance de Hamming utilisant : un arbre d'additionneurs, des LUTs ou des BRAMs. Elles varient peu en terme de performances (fréquence, latence, ressources) mais laissent au concepteur le choix du type de

ressources à utiliser.

L'ensemble des accélérateurs proposés constitue une implémentation matérielle intégrale des opérations de vision de notre chaine 3D EKF-SLAM. L'architecture définie grâce à notre démarche de co-design ne présente pas les performances de l'implantation finale du fait des choix d'environnement de prototypage (communication Xillybus, émulation du flot de pixels à partir de fichiers). Afin de garantir les contraintes temps réel définies dans la section 1.1, il est nécessaire de redéfinir le modèle de communication entre les accélérateurs et le processeur (mémoire partagée, utilisation du DMA). De plus, le traitement des images en provenance directe d'une caméra réduira considérablement le volume de données circulant entre le processeur et la logique et de ce fait, accélérera très significativement la cadence de traitement. Cependant le traitement direct des données caméra nécessitera une synchronisation fine entre le logiciel et les accélérateurs ce qui pourrait limiter la vitesse de traitement effectif.

Ces accélérateurs étant indépendants de la méthode de SLAM employée, il pourrait être intéressant d'évaluer leur intérêt dans le cadre d'implantations existantes de SLAM par méthode d'optimisation (SVO, ORB). En effet, la plupart des méthodes de SLAM utilisent le suivi de points d'intérêt dans une séquence d'images. La méthode ORB SLAM utilise également la combinaison FAST + BRIEF pour la détection de points d'intérêt. Cependant afin d'améliorer le suivi des points, le descripteur a été rendu robuste, dans l'état de l'art, par l'addition de la détection de l'orientation et de l'échelle pour les points suivis. Afin de s'intégrer dans cette approche, il serait donc nécessaire de modifier notre accélérateur pour intégrer ces attributs.

Dans les évolutions futures du prototype, les images lues depuis le disque dur seront replacées par une caméra. On peut s'attendre à un gain significatif en fréquence puisque le microprocesseur n'aura pas à lire l'image en mémoire et à envoyer les pixels vers la logique. La figure 6.1 illustre l'ordonnancement prévu de ce futur prototype.

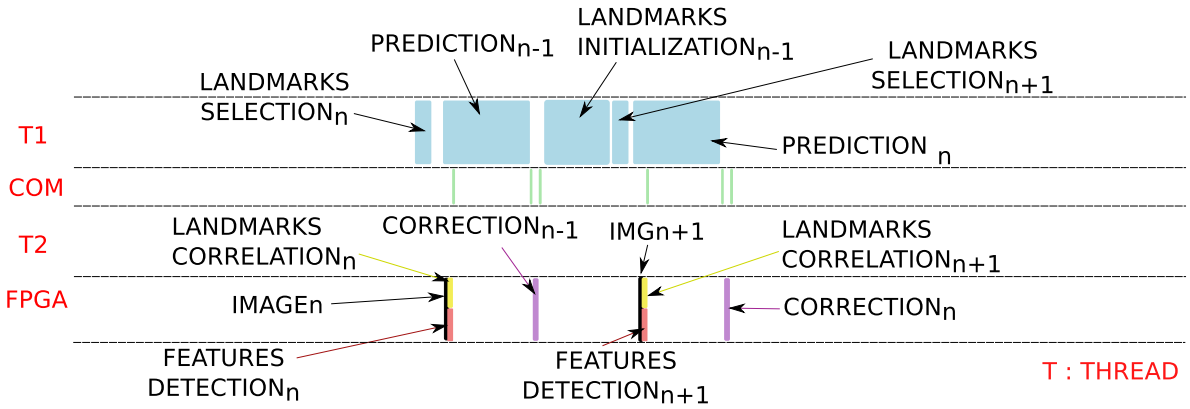

FIGURE 6.1 – Ordonnancement des données du SLAM du futur prototype avec l'accélération de FAST, BRIEF, tessellation, corrélation et avec la caméra directement connectée sur la partie matérielle.

Le SLAM pourra traiter plus d'images à la seconde ce qui minimisera les changements de pose entre 2 images. Ainsi, les variations des pixels entre deux images seront minimes et les descripteurs des points d'intérêt pourront être appariés avec une plus grande certitude. Nous avons fait une estimation des performances que nous pourrions atteindre après cette modification. Le tableau 6.1 récapitule les performances atteintes par ce futur prototype (itération

5). La figure 6.2 illustre le gain potentiel en performance en la mettant en parallèle avec les autres prototypes du SLAM.

Itération n^o	I	II	III	IV	V (camera HW)
Taches C-SLAM	Taux d'occupation CPU [%]				
Communication	5.97	17.83	24.84	29.93	1.31
Sélection des amers	0.4	1.19	1.66	2	5.23
Prédiction	1.04	3.12	4.35	5.24	13.73
Correction	11.94	35.66	10.35	12.47	32.68
Initialisation	0.15	0.45	0.62	0.75	1.96
Caméra	6.47	19.32	26.92	32.42	0
Corrélation	4.38	13.08	18.22	1.5	3.92
Détection de coins	69.65	9.36	13.04	15.71	41.18
Taux d'img/s [Hz]	5	14.9	20.7	24.9	65.4

TABLE 6.1 – Taux d'occupation CPU des tâches de C-SLAM.

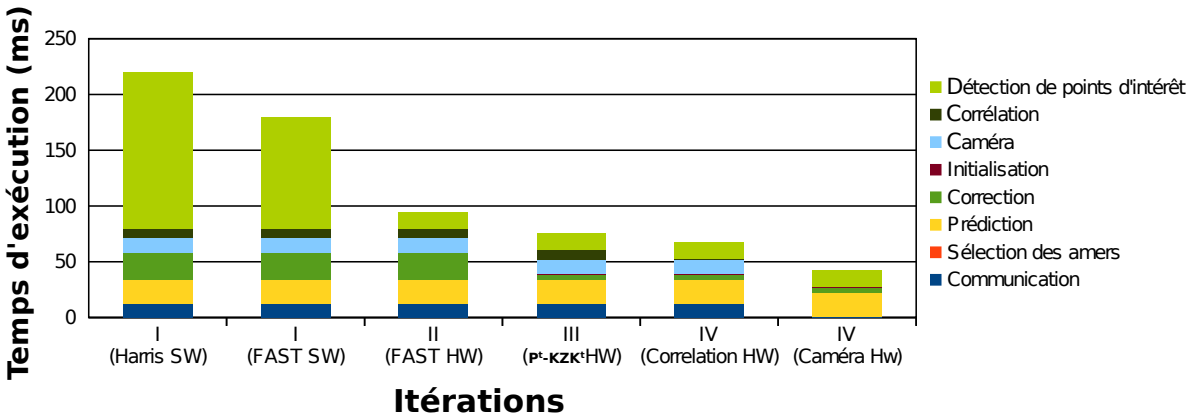

FIGURE 6.2 – Comparaison du temps d'exécution des prototypes C-SLAM précédents et C-SLAM avec la caméra connectée directement sur la logique FPGA.

Pour conclure, j'aimerais ajouter que nous avons été les premiers à implanter une chaîne EKF-SLAM opérationnelle sur une architecture hétérogène et je suis persuadé qu'une partie de la communauté pourrait s'inspirer de la démarche afin de faciliter le déploiement de ses solutions sur systèmes embarqués. En effet, même si beaucoup de monde considère le développement FPGA comme de l'ingénierie, je reste convaincu que l'implantation matérielle a sa place dans la recherche afin de proposer des solutions innovantes à fortes capacités de traitement et faible consommation.

Enfin, le travail de recherche sur cette application EKF-SLAM embarquée n'est pas encore fini et il reste encore de multiples façons de l'optimiser et de l'améliorer. Je pense notamment à la fusion des méthodes de SLAM par filtrage et par optimisation, le SLAM hiérarchique. Cette solution permet, en effet, de tirer parti des avantages de chacune des méthodes et de compenser les défauts de l'une par l'autre.

Annexe A

SystemC

A.1 Structure d'un modèle d'écrit en SystemC

La description architecturale en systemC est similaire à la structure du VHDL/Verilog. Les modèles sont divisés en modules pouvant contenir des processus. La communication entre modules est assurée par des canaux. Ces différents modules sont instanciés dans un module : **top_level** qui utilise le mot clé : *sc_main{}*.

A.1.1 Les modules

Un module en SystemC peut être composé d'autres modules, de canaux de communication entre ces modules (signaux, ou canaux plus abstraits) et, éventuellement, de processus (cf diagramme A.1).

A.1.2 Les ports

Un module possède un ou plusieurs ports (point d'entrée ou sortie du module). Ces ports doivent déclarer les fonctions qui seront utilisées pour communiquer à travers eux. Par exemple :

- Un port en entrée destiné à être relié à un signal normal déclare qu'il utilise la fonction read des signaux ;
- Un port bidirectionnel déclare qu'il utilise les fonctions read et write ;
- Un port destiné à être relié à une fifo (un canal de communication abstrait, de haut niveau) déclare, selon le côté de la fifo, qu'il utilise les fonctions *read*, *nb_read* (read non bloquant), *num_available*, *write*, *nb_write*, *num_free* (les signaux classiques d'une FIFO).

La déclaration de ces fonctions que le port utilise est appelée interface. Une interface est une déclaration des fonctions ("méthodes") qui pourront être utilisées à travers les ports d'un module. Une interface ne contient pas de code, c'est seulement une déclaration de fonctions. Les interfaces permettent au compilateur de détecter très tôt (dès la compilation du code SystemC) le branchement d'un port à un canal qui ne lui est pas adapté.

A.1.3 Les canaux (ou channels)

Les canaux sont les moyens de communication entre les modules. Ils peuvent être basiques et concrets (signaux) ou plus évolués / plus abstraits (fifo, réseau Ethernet, ...). Ils peuvent aussi contenir d'autres canaux, voire même des modules si ce sont des canaux de très haut niveau.

Les canaux contiennent (entre autres) l'implantation du code C++ déclaré dans les interfaces. On dit qu'ils implantent une interface.

On branche un canal à un module par l'intermédiaire d'un port, ce port devant déclarer l'interface implantée par le canal. L'intérieur du module peut alors accéder au canal en appelant les fonctions déclarées dans l'interface. L'interface est donc ce qui lie les ports et les canaux en décrivant comment l'information transite par les ports. Le schéma A.1 symbolise, par l'image port/interface, tout ce dont a besoin un module pour pouvoir utiliser un port : connaître l'interface qu'il publie. On peut donc modifier à loisir le contenu d'un canal, le raffiner progressivement, sans avoir à toucher quoi que ce soit d'autre. Il suffit que son interface reste la même.

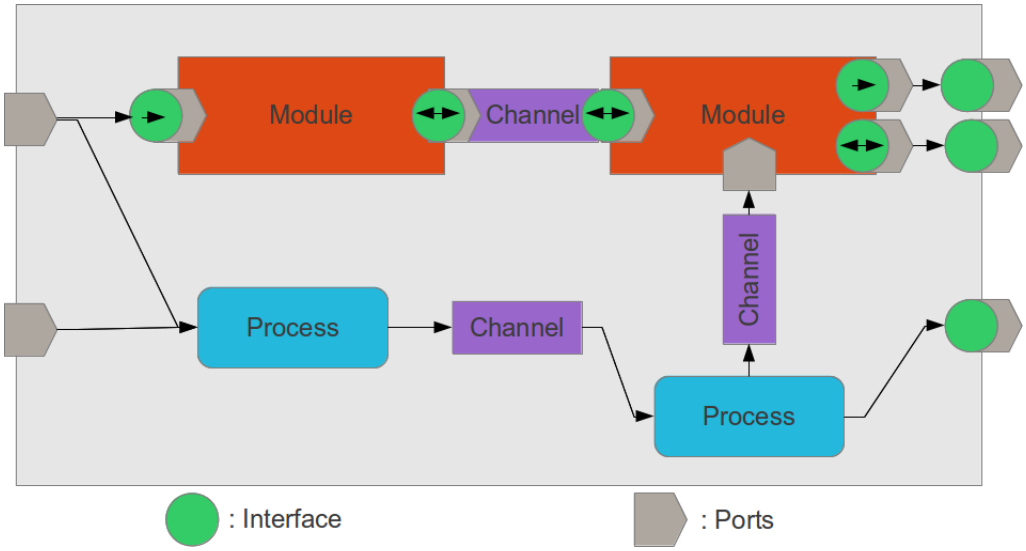

FIGURE A.1 – Diagramme d'un module en SystemC

A.1.4 Les processus

Les processus en SystemC sont similaires à ceux de Verilog et VHDL. Ils décrivent une fonctionnalité, un comportement. Un processus ne doit pas être appelé directement ; c'est le moteur de simulation SystemC qui se charge de l'appeler (le déclencher) sur certains événement particuliers : ceux qui sont dans sa liste des sensibilités (liste des signaux qui déclenchent le processus).

Les processus peuvent éventuellement communiquer directement avec les canaux. Ils n'ont pas besoin de port pour cela, ils appellent directement les méthodes du canal. Il existe 3 types de processus :

— *SC_METHOD* : Comparable au "process" en VHDL. Il se termine toujours et s'exécute en entier lors d'un changement sur les signaux de la liste de sensibilité.
— *SC_THREAD* : Ce type de processus n'a pas de liste de sensibilité. Il peut être comparé à un Thread software classique et peut donc être suspendu et réactivé avec un wait.
— *SC_CTHREAD* : Sa liste de sensibilité n'est constituée que des fronts montants ou descendants d'une horloge. Ce type de processus est classiquement utilisé pour coder des machine à états.

Bibliographie

[Abrate et al., 2007] Abrate, F., Bona, B., and Indri, M. (2007). Experimental EKF-based SLAM for Mini-rovers with IR Sensors Only. In *EMCR*. (Cité en page 42.)

[Adam et al., 1974] Adam, T. L., Chandy, K. M., and Dickson, J. R. (1974). A Comparison of List Schedules for Parallel Processing Systems. *Commun. ACM*, 17 :685–690. (Cité en page 75.)

[Asano et al., 2009] Asano, S., Maruyama, T., and Yamaguchi, Y. (2009). Performance comparison of FPGA, GPU and CPU in image processing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 126–131. (Cité en page 55.)

[Balboni et al., 1996] Balboni, A., Fornaciari, W., and Sciuto, D. (1996). Tosca : A Pragmatic Approach To Co-Design Automation Of Control-Dominated Systems. In De Micheli, G. and Sami, M., editors, *Hardware/Software Co-Design*, volume 310 of *NATO ASI Series*. Springer Netherlands. (Cité en page 70.)

[Bay et al., 2006] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf : Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer. (Cité en pages 47 et 48.)

[Bhattacharya and Bhattacharyya, 2001] Bhattacharya, B. and Bhattacharyya, S. S. (2001). Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49 :2408–2421. (Cité en page 73.)

[Bilsen et al., 1995] Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. A. (1995). Cyclo-static data flow. In *International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3255–3258. (Cité en page 73.)

[Birem and Berry, 2012] Birem, M. and Berry, F. (2012). FPGA-based real time extraction of visual features. pages 3053–3056. IEEE. (Cité en page 47.)

[Birem and Berry, 2014] Birem, M. and Berry, F. (2014). DreamCam : A modular FPGA-based smart camera architecture. *Journal of Systems Architecture*, 60 :519–527. (Cité en page 22.)

[Bonato et al., 2008] Bonato, V., Marques, E., and Constantinides, G. A. (2008). A Parallel Hardware Architecture for Scale and Rotation Invariant Feature Detection. *IEEE Transactions on Circuits and Systems for Video Technology*, 18 :1703–1712. (Cité en page 42.)

[Botero et al., 2012] Botero, Piat, Devy, and Boizard (2012). An FPGA accelerator for multispectral vision-based EKF-SLAM. *Proc. IROS Workshop on Smart CAmeras for roBOTic applications (SCaBot), Vilamoura (Portugal)*. (Cité en pages 42, 44 et 101.)

[Brenot et al., 2015] Brenot, F., Fillatreau, P., and Piat, J. (2015). FPGA based accelerator for visual features detection. In *Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM), 2015 IEEE International Workshop of*, pages 1–6. IEEE. (Cité en pages 34, 114 et 128.)

[Brenot et al., 2016] Brenot, F., Piat, J., and Fillatreau, P. (2016). FPGA based hardware acceleration of a BRIEF correlator module for a monocular SLAM application. (Cité en pages 34 et 136.)

[Calonder et al., 2012] Calonder, M., Lepetit, V., Ozuysal, M., Trzcinski, T., Strecha, C., and Fua, P. (2012). BRIEF : Computing a local binary descriptor very fast. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34 :1281–1298. (Cité en page 49.)

[Calonder et al., 2010] Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). Brief : Binary robust independent elementary features. *Computer Vision–ECCV 2010*, pages 778–792. (Cité en pages 48, 52 et 133.)

[DARPA, 2012] DARPA (2012). Robotics challenge. (Cité en page 22.)

[Davison, 2003] Davison (2003). Real-time simultaneous localisation and mapping with a single camera. pages 1403–1410 vol.2. IEEE. (Cité en page 37.)

[Davison et al., 2007] 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 :1052–1067. (Cité en page 43.)

[Di Stefano et al., 2005] Di Stefano, L., Mattoccia, S., and Tombari, F. (2005). ZNCC-based template matching using bounded partial correlation. *Pattern Recognition Letters*, 26 :2129–2134. (Cité en page 52.)

[Faessler et al., 2016] Faessler, M., Fontana, F., Forster, C., Mueggler, E., Pizzoli, M., and Scaramuzza, D. (2016). Autonomous, Vision-based Flight and Live Dense 3d Mapping with a Quadrotor Micro Aerial Vehicle : Autonomous, Vision-based Flight and Live Dense 3d Mapping. *Journal of Field Robotics*, 33 :431–450. (Cité en page 42.)

[Fischler and Bolles, 1981] Fischler and Bolles (1981). Random Sample Consensus : A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*. (Cité en page 43.)

[Gonzalez, 2012] Gonzalez, A. (2012). *Localisation par vision multi-spectrale. Application aux systèmes embarqués*. PhD thesis. (Cité en pages 38 et 42.)

[Gonzalez et al., 2011] Gonzalez, A., Codol, J.-M., and Devy, M. (2011). A C-embedded algorithm for real-time monocular SLAM. pages 665–668. IEEE. (Cité en pages 44 et 84.)

[Hamming, 1950] Hamming (1950). Error Detecting and Error Correcting Codes. (Cité en page 52.)

[Harris and Stephens, 1988] Harris, C. and Stephens, M. (1988). A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Citeseer. (Cité en pages 43 et 46.)

[Hill et al., 2015] Hill, K., Craciun, S., George, A., and Lam, H. (2015). Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193. (Cité en page 66.)

[Izagirre et al., 2006] Izagirre, A., Arana, N., Garcia, C., Aginagalde, A., and Esnaola, J. A. (2006). Real time stereo computation in sheet hydroforming processes. San Sebastian. (Cité en page 28.)

[Jovanovic and Milutinovic, 2012] Jovanovic, Z. and Milutinovic, V. (2012). FPGA accelerator for floating-point matrix multiplication. *IET Computers & Digital Techniques*, 6 :249. (Cité en page 105.)

[Kahn, 1974] Kahn, G. (1974). The Semantics of Simple Language for Parallel Programming. *ResearchGate*, pages 471–475. (Cité en page 73.)

[Kalman, 1960] Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *J. Basic Eng*, 82 :35–45. (Cité en page 36.)

[Kanade, 1981] Kanade, B. D. L. T. (1981). with an Application to Stereo Vision. (Cité en page 32.)

[Kraft et al., 2008] Kraft, M., Schmidt, A., and Kasinski, A. J. (2008). High-Speed Image Feature Detection Using FPGA Implementation of Fast Algorithm. In *VISAPP (1)*, pages 174–179. (Cité en pages 76, 114, 116 et 122.)

[Kwok and Ahmad, 1998] Kwok, Y.-K. and Ahmad, I. (1998). Benchmarking the task graph scheduling algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 531–537. IEEE. (Cité en page 75.)

[Lapray et al., 2014] Lapray, P.-j., Heyrman, B., and Ginhac, D. (2014). HDR-ARtiSt : An Adaptive Real-time Smart camera for High Dynamic Range imaging. (Cité en page 22.)

[Lee and al, 1987] Lee, E. A. and al, e. (1987). *Synchronous data flow*. (Cité en pages 73, 74 et 89.)

[Lee and Messerschmitt, 1987] Lee, E. A. and Messerschmitt, D. G. (1987). Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on computers*, 36 :24–35. (Cité en pages 74 et 89.)

[Lee and Parks, 1995] Lee, E. A. and Parks, T. (1995). Dataflow Process Networks. In *Proceedings of the IEEE*, pages 773–799. (Cité en page 73.)

[Leutenegger et al., 2011] Leutenegger, S., Chli, M., and Siegwart, R. Y. (2011). BRISK : Binary robust invariant scalable keypoints. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2548–2555. IEEE. (Cité en pages 47 et 48.)

[Lowe, 1999] Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee. (Cité en page 47.)

[Lowe, 2004] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60 :91–110. (Cité en page 48.)

[Mohammad Awrangjeb and Fraser, 2012] Mohammad Awrangjeb, G. L. and Fraser, C. S. (2012). Performance comparisons of contour-based corner detectors. In *IEEE Transactions on Image Processing*. (Cité en page 46.)

[Montiel, 2006] Montiel, J. (2006). Unified inverse depth parametrization for monocular slam. In *Proc. Robotics : Science and Systems (RSS)*. (Cité en page 44.)

[Nikolic et al., 2014] 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. (Cité en page 42.)

[Olaizola Goenaga, 2014] Olaizola Goenaga, G. (2014). Tribological analysis of hot stamping process. (Cité en page 47.)

[Piat et al., 2013] Piat, J., Marquez-Gamez, D. A., and Devy, M. (2013). Embedded vision-based SLAM : A model-driven approach. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 284–289. IEEE. (Cité en page 88.)

[Rosten and Drummond, 2005] Rosten, E. and Drummond, T. (2005). Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1508–1515. IEEE. (Cité en pages 46, 76 et 120.)

[Roussillon et al., 2011] Roussillon, Gonzalez, Sola, Lacroix, and Devy (2011). RT-SLAM : A Generic and Real-Time Visual SLAM Implementation. In *Computer Vision Systems*, volume 6962, pages 31–40. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cité en page 43.)

[Roussillon, 2013] Roussillon, C. (2013). *Une solution opérationnelle de localisation pour des véhicules autonomes basée sur le SLAM*. PhD thesis. (Cité en pages 35 et 43.)

[Rublee et al., 2011] Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). ORB : an efficient alternative to SIFT or SURF. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE. (Cité en pages 47 et 48.)

[Saenz de argandona et al., 2007] Saenz de argandona, E., Astiria, A., Arana, N., Izagirre, A., Pop, R., Terzyk, T., and Fillatreau, P. (2007). Automatic detection of burrs in sheet metal cutting processes by a combination of a sensor based monitoring system, an artificial vision system and an intelligent control system. In *57th CIRP General Assembly*. (Cité en page 28.)

[Savatier, 2015] Savatier, X. (2015). La vision embarquée pour les systèmes mobiles autonomes : état des lieux, enjeux et perspectives. (Cité en pages 20 et 21.)

[Sciuto et al., 2002] Sciuto, Salice, F., Pomante, L., and Fornaciari, W. (2002). Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES)*. (Cité en page 69.)

[Shaout et al., 2009] Shaout, A., El-mousa, A. H., and Mattar, K. (2009). *Specification and Modeling of HW/SW CO-Design for Heterogeneous Embedded Systems*. (Cité en pages 69, 70, 76 et 83.)

[Shi et al., 2010] Shi, B., Chen, S., Huang, F., Wang, C., and Bi, K. (2010). The Parallel Processing Based on CUDA for Convolution Filter FDK Reconstruction of CT. In *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, pages 149–153. (Cité en page 54.)

[Shi and Tomasi, 1994] Shi, J. and Tomasi (1994). Good features to track. pages 593–600. IEEE Comput. Soc. Press. (Cité en page 46.)

[Sinnen, 2007] Sinnen, O. (2007). *Task Scheduling for Parallel Systems*. Wiley edition. (Cité en page 75.)

[Sinnen and Sousa, 2005] Sinnen, O. and Sousa, L. A. (2005). Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16 :503–515. (Cité en page 75.)

[Sklyarov et al., 2015] Sklyarov, V., Skliarova, I., Sudnitson, A., and Kruus, M. (2015). FPGA-based time and cost effective Hamming weight comparators for binary vectors. In *IEEE EUROCON 2015 - International Conference on Computer as a Tool (EUROCON)*, pages 1–6. (Cité en page 138.)

[Smith et al., 1991] Smith, R., Self, M., and Cheeseman, P. (1991). *A stochastic map for uncertain spatial relationships*. (Cité en page 37.)

[Sola, 2007] Sola, J. (2007). *Towards Visual Localization, Mapping and Moving Objects Tracking*. PhD thesis. (Cité en page 44.)

[Sola et al., 2007] Sola, J., Monin, A., and Devy, M. (2007). Bicamslam : Two times mono is more than stereo. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 4795–4800. IEEE. (Cité en page 38.)

[Spivey, 2003] Spivey, J. (2003). Fast, accurate call graph profiling. *Oxford University Computing Laboratory*. (Cité en pages 98 et 105.)

[Stankovic, 1988] Stankovic, J. A. (1988). Misconceptions about real-time computing : a serious problem for next-generation systems. *Computer*, 21 :10–19. (Cité en page 27.)

[Sérot et al., 2013] Sérot, J., Berry, F., and Ahmed, S. (2013). CAPH : a language for implementing stream-processing applications on FPGAs. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer New York. (Cité en page 22.)

[Tertei et al., 2014] Tertei, D. T., Piat, J., and Devy, M. (2014). FPGA design and implementation of a matrix multiplier based accelerator for 3d EKF SLAM. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6. IEEE. (Cité en pages 34, 101, 104, 106 et 124.)

[Thrun et al., 2005] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics*. Intelligent robotics and autonomous agents. MIT Press, Cambridge, Mass. OCLC : ocm58451645. (Cité en page 104.)

[Tobin Buck, 1993] Tobin Buck, J. (1993). *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis. (Cité en page 73.)

[Vincke et al., 2012] Vincke, Elouardi, A., and Lambert, A. (2012). Real time simultaneous localization and mapping : towards low-cost multiprocessor embedded systems. *EURASIP Journal on Embedded Systems*. (Cité en page 69.)

[Wakabayashi, 2004] Wakabayashi, K. (2004). C-based behavioral synthesis and verification analysis on industrial design examples. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 344–348. IEEE Press. (Cité en page 66.)

[Wu and Gajski, 1990] Wu, M.-Y. and Gajski, D. (1990). Hypertool : a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, pages pages 330–343. (Cité en page 75.)

[Yang et al., 2008] Yang, Z., Zhu, Y., and Pu, Y. (2008). Parallel Image Processing Based on CUDA. In *2008 International Conference on Computer Science and Software Engineering*, volume 3, pages 198–201. (Cité en page 57.)

[Zhuo and Prasanna, 2007] Zhuo and Prasanna (2007). Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems. In *IEEE Transactions on Parallele and Distributed Systems, Vol. 18, No. 4*. (Cité en page 105.)