Approche multi-processeurs homogènes sur System-on-Chip pour le traitement d'image

Lionel Damez

N° d'ordre : 2004
EDSPIC : 472

UNIVERSITÉ BLAISE PASCAL - CLERMONT II

École Doctorale
Sciences Pour l'Ingénieur de Clermont-Ferrand

Thèse présentée par :
Lionel DAMEZ

Formation Doctorale CSTI :
Composants et Systèmes pour le Traitement de l'Information

en vue de l'obtention du grade de

DOCTEUR D'UNIVERSITÉ

spécialité: Vision pour la Robotique

Approche multi-processeurs homogènes sur *System-On-Chip* pour le traitement d'image.

Soutenue publiquement le : 17 Décembre 2009 devant le jury :

M. ou Mme	
Jean-Pierre DÉRUTIN	Directeur de thèse
Alexis LANDRAULT	Président du Jury
Dominique GINHAC	Rapporteur
Dominique HOUZET	Rapporteur
Virginie FRESSE	Examinateur

Remerciements

Je tiens tout d'abord à remercier Jean Pierre Dérutin pour la confiance qu'il m'a accordé pendant toute la durée de la thèse et pour avoir su me communiquer un peu de sa persévérance.

A Dominique Ginhac et Dominique Houzet pour avoir accepté de juger ce travail malgré les contraintes temporelles.

Je remercie Virginie Fresse qui m'a fait l'honneur de participer au Jury de soutenance. A Alexis Landrault pour l'honneur qu'il m'a fait en présidant ce jury, je le remercie également pour le soutien qu'il m'a apporté aux moments les plus difficiles de mon travail.

Je remercie les divers stagiaires que j'ai pu encadrer pendant la thèse en particulier Adrien, Marie, Dorin, Rémi, Sylvain et Benoit.

Je remercie François, Laurence et Florent qui ont partagé avec moi le bureau 3009, et particulièrement Loïc pour ses nombreux coups de mains.

Un grand merci à toute la bande des thésards, en particulier Eric, Lucie, Joel, Fabio Noel et Pierre.

Je remercie aussi toute l'équipe du DEPECA qui m'a accueilli à Alcalà de Henares. Je remercie José Luis Lázaro Galilea, pour toute l'assistance matérielle auquel j'ai eu accès grâce à lui, ainsi que Raùl, Alvaro et Pedro et les compères Olivier, Dani et Santi.

Je remercie mes amis de longue date, les Christophe et Franck, qui me donnent le sentiment d'avoir une deuxième famille.

Enfin, je remercie ma famille pour sa patience, et pour les encouragements et le soutien aussi bien matériel que moral qu'elle m'a apporté.

Résumé

La conception de prototypes de systèmes de vision en temps réel embarqué est sujet à de multiples contraintes sévères et fortement contradictoires. Dans le cas de capteurs dits "intelligents", tout ou parti des traitements sont effectué à proximité de la rétine, il est nécessaire de fournir une puissance de traitement suffisante pour exécuter les algorithmes à la cadence des capteurs d'images avec un dispositif de taille minimale et consommant peu d'énergie. La densité d'intégration des transistors permet de nos jours de concentrer l'essentiel, voire l'intégralité de l'application dans un seul composant. La conception d'un tel composant (appelé système monopuce ou SoC) et l'implantation d'algorithmes de plus en plus complexes pose problème si on veut l'associer avec une approche de prototypage rapide d'applications scientifiques.

Afin de réduire de manière significative le temps et les différents coûts de conception, le procédé de conception est fortement automatisé. La conception matérielle est basée sur la dérivation d'un modèle d'architecture multiprocesseur générique de manière à répondre aux besoins de capacité de traitement et de communication spécifiques à l'application visée. Les principales étapes manuelles se réduisent au choix et au paramétrage des différents composants matériels synthétisables disponibles. La conception logicielle consiste en la parallélisation des algorithmes, qui est facilitée par l'homogénéité et la régularité de l'architecture de traitement parallèle et la possibilité d'employer des outils d'aide à la parallélisation.

Avec l'approche de conception sont présentés les premiers éléments constitutifs qui permettent de la mettre en oeuvre. Ceux ci portent essentiellement sur les aspects de conception matérielle. Ils sont de nature diverse, outils d'aide à la conception permettant la génération automatisée d'un réseau de processeurs à mémoire distribuée, ou composants matériels (IP) pour les communications dont notamment un routeur de paquets. L'approche proposée est illustrée par l'implantation d'un traitement de stabilisation temps réel vidéo sur technologie SoPC.

Mots-clés : Architectures de vision, Système monopuce, FPGA, architecture parallèle, mémoire distribuée, passage de message.

Abstract

The conception of a real time embedded vision system imposes multiple and severe design constraints, which could be conflicting. Smart camera applications usually requires integration of sofisticated processing near the transducer, with sufficient processing power to run the algorithms at the information flow rate, and using a system of minimal size that consumes little power. Today, transistor integration density enables concentration of the main part, or even all, of a complete system on a sole component (System on a Chip - SoC).

A strongly automated design flow is proposed, it reduces the design effort and conception costs, in order to enable fast implementation of complex algorithms into a SoC. Our overall hardware implementation method is based upon meeting algorithm processing power requirement and communication needs with refinement of a generic parallel architecture model. Actual hardware implementation is done by the choice and parameterization of readily available reconfigurable hardware modules and customizable commercially available IPs. Software conception is based upon parallelisation of the algoritms. An homogeneous and regular hardware architecture model is chosen which enables to use parallelisation tools.

With the design method, most of the works presented in this thesis are focused on enabling a automated hardware design enviromnent. This includes various works, tools enabling automated generation of a homogeneous network of communicating processors, or hardware components (IP) for communication network, including a paquet router. The presented approach is illustrated with the embedding a real time image stabilization algorithm on SoPC technology.

Keywords : Vision architectures, System on a Chip, FPGA, Parallel architecture, Distributed Memory, Message Passing.

Table des matières

Introduction **1**

1 Architectures pour le traitement d'image **5**
1.1 Approches conventionnelles 5
1.2 Architectures dédiés à la vision 6
1.2.1 Les caméras intelligentes 6
1.2.2 Les rétines artificielles 7
1.2.3 Approches bio-inspirées 8
1.2.4 Architectures parallèles pour la vision 9
1.3 Exemples de réalisations 11
1.3.1 La camera Philips inca+ 11
1.3.2 La plateforme SeeMOS 11
1.3.3 La rétine programmable PVLSAR 12
1.3.4 Camera intelligente haute fréquence haute résolution . . . 13
1.4 Intérêt des systèmes monopuces 13

2 Des systèmes monopuces aux multiprocesseurs embarqués. **15**
2.1 Architecture d'un système sur puce 16
2.1.1 Processeurs généraux (CPU) 16
2.1.2 Accélérateurs matériels 17
2.1.3 Composants de mémorisation 17
2.1.4 Systèmes de communication 19
2.2 Approches de conception des systèmes monopuces 26
2.2.1 Conception logicielle/matérielle par une approche codesign 26
2.2.2 Réutilisation des IPs 26
2.2.3 Réutilisation d'une plateforme de référence 27
2.2.4 Conception matérielle par la synthèse de haut niveau . . . 27
2.2.5 Conception d'un ASIP 28
2.2.6 Conception de MP-SOC 29

3 Une méthode de conception de MP-SoC homogènes et réguliers pour des applications parallèles 31
3.1 Objectifs visés et principe 32
3.2 Le concept de Réseau Homogène de Processeurs Communiquants 32
3.3 Description globale du flot de conception 33
3.3.1 Etape de parallélisation 35
3.3.2 Etape de conception matérielle 39
3.4 Conclusion 45

4 Contributions pour l'implantation matérielle de RHPC sur SoPC Xilinx 47
4.1 Etude préliminaire 47
4.1.1 Ressources disponibles dans un SOPC 47
4.1.2 Estimation des coûts du processeur 48
4.1.3 Estimation des ressources mémoires 50
4.2 Description des composants matériels 52
4.2.1 Communication point à point par canal FIFO 52
4.2.2 Communication point à point par canal DMA 53
4.2.3 Communications réseau avec un routeur 54
4.2.4 Communications réseau avec routeur et interface réseau DMA 58
4.2.5 Composants d'entrée/sortie vidéo dédiés 61
4.3 Programmation des communications 63
4.3.1 Communications point-à-points FIFO 63
4.3.2 Communications point-à-points DMA 64
4.3.3 Communication réseau avec le routeur sans DMA 64
4.3.4 Communication réseau avec le routeur avec DMA 65
4.3.5 Vers une programmation homogène des communications . 65
4.4 Résultats d'implantation de RHPC 66
4.4.1 Système avec lien de communication point à point FIFO . 66
4.4.2 Système avec lien de communication point à point DMA . 68
4.4.3 Système avec routeur sans DMA 69
4.5 Conclusions 70

5 Application de stabilisation vidéo temps réel. 71
5.1 La stabilisation électronique d'images 72
5.1.1 Mise en Correspondance 73
5.1.2 Estimation du Déplacement 73
5.1.3 Compensation des mouvements 74

5.2 Notre Méthode de Stabilisation 74
5.2.1 Descriptif général 74
5.2.2 Construction de l'image intégrale 75
5.2.3 Détection de primitives 77
5.2.4 Construction des images sous-échantillonnées 78
5.2.5 Mise en Correspondance de primitives 78
5.2.6 Estimation du mouvement 79
5.2.7 Accumulation et filtrage des paramètres 80
5.3 Qualification de la Stabilisation 81
5.4 Présentation de l'architecture Babylon 84
5.5 Implantation parallèle 87
5.5.1 Analyse de l'algorithme séquentiel 87
5.5.2 Description de l'implantation parallèle 88
5.6 Résultats 91
5.6.1 Implantation sur Babylone 91
5.6.2 Implantation sur SoPC 97
5.7 Conclusion 101

Conclusion et perspectives **103**

Bibliographie **104**

Table des figures

1.1 La camera Philips inca 12
1.2 La plateforme SeeMos 12
1.3 Schema synoptique de la plateforme de vision PVLSAR 13
1.4 Banc de traitement PVLSAR 14

2.1 Le standard de bus AMBA. 20
2.2 Le standard de bus coreconnect. 21
2.3 Couches réseau et élements d'un NoC correspondant. 23
2.4 Exemples de topologies de réseaux directs. 24

3.1 Réseau Homogène de Processeurs Communiquants : exemple de topologie Hypercube 8 noeuds avec dispositif d'entrée vidéo dédié. 32
3.2 Flot de conception simplifié 36
3.3 Graphe de processus communicants dans une application parallèle : PIPE(SCM) 37
3.4 Flot de conception automatisé d'un réseau homogène et régulier de processeurs sur SoPC 43
3.5 Interface graphique de l'outil CubeGen 44

4.1 Architecture du processeur MicroBlaze 48
4.2 Evolution de la mémoire disponible par noeud et de la consommation mémoire par noeud en fonction du nombre de noeuds. . . . 51
4.3 Connection point à point FIFO entre 2 processeurs. 52
4.4 Connection point à point avec DMA entre 2 processeurs. 53
4.5 Réseau de 4 processeur avec routeur, sans interface DMA (à gauche) et avec interface DMA (à droite). 54
4.6 Format de paquet manipulé par le routeur. 55
4.7 Format du premier flit d'entête. 56
4.8 Format du second flit d'entête. 56
4.9 Blocs fonctionnels du routeur. 57

4.10 Blocs fonctionnels de l'interface réseau DMA. 60
4.11 Deux approches d'interface d'Entrée/Sortie vidéo dédiée, illustrées avec des RHPC de 4 noeuds. 62
4.12 Fonctions élémentaires d'accès au ports d'entrée/sortie FSL. . . . 63

5.1 Schéma synoptique d'une itération de l'algorithme de stabilisation. 75
5.2 Image 10x10 (à gauche) et son image intégrale (à droite). 76
5.3 A gauche, primitives sélectionnées par les ondelettes de Harr sur une séquence d'images réelle. A droite, schéma de la mise en correspondance des primitives entre deux images. 78
5.4 Les masques des trois types d'ondelettes employés. De gauche à droite : ondelettes verticale, horizontale et diagonale. 79
5.5 Séquence d'images synthétiques instable (en haut) et séquence après stabilisation (en bas). 81
5.6 Instruction C Altivec à deux opérandes : **VR = vec_add(VA, VB)**. 85
5.7 Instruction C Altivec de type DSP : **VD = vec_madd(VA, VB, VC)**. 86
5.8 Schéma de parallélisation de l'algorithme. 90
5.9 Schéma d'implantation parallèle sur SoPC. 92

Liste des tableaux

4.1 Espace disponible et ressources dans des composants Virtex-4 et Virtex-5 de grande taille. 48
4.2 Impact de quelques unités de traitement optionnelles sur la surface occupée et l'utilisation de ressources FPGA pour l'implantation d'un processeur MicroBlaze. 49
4.3 Resources SoPC utilisées par le routeur en terme de surface, de registres et de fonctions logiques. 57
4.4 Ressources SoPC utilisées par un interface DMA en terme de surface, de registres et de fonctions logiques. 60
4.5 Ressources SoPC utilisées pour un système de 4, 8, 16, et 32 processeurs, topologie hypercube avec des liens point-à-point FSL. . 67
4.6 Ressources SoPC utilisées pour un système de 4, 8, 16, et 32 processeurs, topologie hypercube avec des liens point-à-point DMA. . 68
4.7 Ressources SoPC utilisées par un système comprenant 8 noeuds (1 processeur, 1 routeur et 16ko de mémoire locale.) 69

5.1 Caractéristiques des différentes étapes de l'application. 87
5.2 Importance relative des différentes étapes en fonction du format vidéo et de la distance de recherche. 88
5.3 Caractéristiques des systèmes utilisés. 93
5.4 Performances temporelles de [*arch_1*] et [*arch_2*] - images 320x240 et paramètres *bench_1* ($n = 24$ primitives, $T = 15$ pixels). 93
5.5 Performances temporelles de [*arch_3*] - images 320x240 et paramètres *bench_1* ($n = 24$ primitives, $T = 15$ pixels). 94
5.6 Performances temporelles de [*arch_3*] - images 640x480 et paramètres *bench_2* ($n = 84$, $T = 30$). 94
5.7 Performances temporelles de [*arch_3*] - images 640x480 et paramètres *bench_2* ($n = 84$, $T = 30$). 95

5.8 Performances temporelles de [*arch_3*] - images 1280x960 et paramètres *bench_3* ($n = 84$, $T = 60$). 95
5.9 Performances temporelles de [*arch_3*] - images 1280x960 et paramètres *bench_3* ($n = 84$, $T = 60$). 96
5.10 Performances temporelles de [*arch_3*] - images 2560x1920 et paramètres *bench_4* ($n = 84$, $T = 120$). 96
5.11 Performances temporelles de [*arch_3*] - images 2560x1920 et paramètres *bench_4* ($n = 84$, $T = 120$). 97
5.12 Performances temporelles de [*arch_3*] - images 5120x3840 et paramètres *bench_5* ($n = 84$, $T = 240$). 97
5.13 Performances temporelles de [*arch_3*] - images 5120x3840 et paramètres *bench_5* ($n = 84$, $T = 240$). 98
5.14 ressources SoPC consommées pour 4, 8, 16, et 32 processeurs avec des connections point à point FSL sur Virtex4. 98
5.15 ressources SoPC consommées pour 4, 8, 16 et 32 processeurs avec des connections point à point FSL sur Virtex5. 99
5.16 Variation des paramètres d'application parallèle en fonction du nombre de noeuds. 99
5.17 Comparaison de la consommation mémoire de l'application et de celle disponible par noeud (Ko) en fonction du nombre de processeur (1 à 64). 100
5.18 Temps d'exécution (ms) de l'application pour 1 à 64 processeurs. 100
5.19 Accélération de l'application pour 1 à 64 processeurs. 100

Introduction

Le principal champ d'application de la robotique mobile est de permettre la réalisation de tâches que l'homme ne peut pas accomplir. Soit parce que le milieu dans lequel l'action du robot doit être accomplie est difficilement voire complètement inaccessible à l'homme (missions spatiales ou sous-marines, interventions chirurgicales à l'intérieur du corps humain), ou parce que le milieu est dangereux (zone irradiée, applications militaires). Un robot mobile peut également servir à automatiser une tâche couramment accomplie par le plus grand nombre, mais aussi à effectuer la réalisation de ces tâches pour une minorité (handicapés, personnes âgées).

Par exemple, un véhicule intelligent qui automatise la fonction de transport, ouvre la possibilité d'offrir un service de transport public individualisé avec une couverture plus large que le transport en commun, aussi bien en terme de zones desservies que d'horaires de circulation et ce pour un prix de reviens inférieur au taxi. Dans le cas d'une zone de circulation structurée mais ouverte comme une zone urbaine, les défis à relever sont nombreux pour permettre un contrôle automatique ou fortement assisté du véhicule. En plus des différents problèmes liés à la sécurité, se pose par exemple le problème de la localisation précise et référencée du véhicule. Pour éviter d'avoir à équiper l'infrastructure, des dispositifs de localisation sont embarqués dans le véhicule. Utiliser un GPS différentiel permet une localisation directe, avec une précision suffisante (quelques cm) et une fréquence d'actualisation des données généralement satisfaisantes (quelques Hz) mais ne constitue pas une solution fiable. Une forte réduction de la précision effective de localisation, est provoquée par l'éventualité d'un trajet multiple du signal GPS dès que le véhicule ne se situe pas en milieu complètement ouvert. De plus à cause de l'effet canyon, le signal GPS peut être dévié par un obstacle et ne plus être reçu. Des dispositifs de vision embarqués sont alors utilisés pour remplacer à moindre coût et de manière fiable la localisation par GPS.

Comme l'information fournie par une caméra standard est très éloignée de la

position du véhicule dans son environnement, un certain nombre de traitement d'images doivent être effectuées pour interpréter le flux vidéo fourni par la camera. Un tel véhicule intelligent ou plus généralement un robot autonome équipé d'une (ou plusieurs) caméra, doit ainsi toujours embarquer un système de traitement d'images afin de permettre les fonctions requises (localisation, détection d'obstacle, ...)

La réalisation de telles fonctions implique l'utilisation d'algorithmes de traitements d'images souvent complexes, et dont l'exécution en temps réel nécessite une architecture matérielle qui non seulement est dotée de capacités de calculs importantes mais respecte aussi des contraintes sévères de taille et de consommation d'énergie.

Le principe d'une caméra intelligente est d'intégrer dans un volume réduit, toute la chaîne de traitement d'image, depuis le capteur jusqu'aux dispositifs de traitements, en passant par les entrées/sorties nécessaires à l'application.

Les contraintes d'application nécessitent en général la mise en oeuvre d'architectures dédiées. La conception d'un tel système dédié pose de nombreux problèmes. Un premier problème est de permettre un cycle rapide de conception - implantation - validation du système. Il faut aussi d'une part pouvoir proposer une architecture adéquate à chaque nouvelle évolution des algorithmes. Il faut encore d'autre part ouvrir la possibilité d'étendre l'architecture matérielle ou de pouvoir la mettre à jour facilement au niveau technologique.

Les capacités d'intégration des composants microélectroniques actuels sont telles qu'il est possible de concentrer l'essentiel, voire la totalité d'un système complet dans une même puce. La conception de ce type de composant, système monopuce ou SoC (*System on a Chip*) est classiquement menée avec une approche de type co-design, où les différentes fonctions du systèmes doivent être partitionnées entre fonctions logicielles et fonction matérielles.

L'approche proposée dans ce manuscrit n'est pas une approche de codesign classique mais une approche de conception alternative, basée sur la parallélisation des algorithmes de visions et leur placement sur un réseau homogène à topologie fixe de processeurs communicants.

Notre objectif est de proposer une approche apportant un compromis entre la recherche d'une solution matérielle la plus adéquate possible aux algorithmes à implanter, et la réduction de l'effort de conception nécessaire.

L'approche proposée est aussi une méthode le prototypage rapide, permettant d'obtenir en un délai bref un prototype testable en vérité de terrain.

La cible technologique visée est de type système monopuce sur composant

reconfigurable ou SoPC (*System on a Programable Chip*).

La réduction des temps de conception se base sur une forte automatisation des outils de conceptions, qui sont d'une part des outils d'aide à la parallélisation, et d'autre part des outils de génération d'architecture taillée sur mesure, à partir de bibliothèque d'IP (standards ou dédiés).

Les travaux présentés dans ce manuscrit s'organisent de la manière suivante :

- Dans le premier chapitre un panorama des différentes approches de conception d'architectures de vision sont proposées.
- Dans le second chapitre nous nous intéresseront aux problèmes de conceptions de systèmes monopuces, et aux différentes méthodes employées pour les adresser.
- Le troisième chapitre est consacrée à la description de la méthode proposée. Une première partie de ce chapitre est consacrée au problème de la parallélisation, et décrit les outils d'aide à la parallélisation employés. Une seconde partie de ce chapitre décrit le problème de la recherche d'une architecture matérielle adéquate, et les outils proposés pour l'exploration d'architectures.
- Le quatrième chapitre décris les différents travaux effectués pour mettre en oeuvre l'exploration d'architecture. L'accent est notamment mis sur les aspects communications avec la description de divers dispositifs maternel de communications.
- Dans le cinquième chapitre enfin, nous confrontons la validité de l'approche proposée à travers l'implantation d'un traitement de stabilisation temps réel vidéo.

Chapitre 1

Architectures pour le traitement d'image

Traditionnellement, la réalisation d'un système de vision est envisagée en reliant une caméra à un ordinateur par un canal de communication physique.

Le volume de données impliqué dans les traitements d'images, et le besoin de puissance de traitement pour obtenir les résultats en un temps de traitement bref ont cependant amené dès les années 1970 à s'intéresser à des architectures dédiées à ce type d'applications.

Depuis, les efforts constants en terme de miniaturisation ont amené la possibilité d'intégrer un système de traitement d'image complet dans un boitier de camera aussi réduit qu'un boitier de camera de vidéo-surveillance.

L'objectif de ce chapitre est de montrer la grande diversité d'approches envisageables pour la mise en oeuvre d'un système de traitement d'image embarqué. Les avantages et les inconvénients des diverses approches sont discutés suivant les préoccupations de cette thèse. Après la présentation des principaux concepts quelques exemples de réalisations sont données.

1.1 Approches conventionnelles

Classiquement, un système de vision est considéré en deux étapes principales, acquisition puis traitement.

Pour des applications suffisamment peu contraignantes, il est possible ne faire appel qu'à des composants standards (une ou plusieurs camera(s), connectée(s) à un ordinateur par un bus vidéo). Cette approche a l'avantage d'un certain confort de mise en oeuvre, avec des coûts relativement modérés.

Le rôle de la caméra est alors uniquement de transformer le signal lumineux en information sous forme de signal électrique, l'ordinateur hôte servant à la fois d'unité de traitement, et de plateforme de développement.

Les traitements peuvent être effectués sous forme de logiciel avec un fort niveau d'abstraction de l'architecture. La présence d'outils de développement de haut niveau et de bibliothèques permettent de réduire les temps de mise au point.

Dans le cadre d'applications en robotique mobile, l'application nécessite d'utiliser des composants moins standards voire complètement spécifiques quand il s'agit de travailler à une cadence vidéo ou des résolutions particulièrement élevées, ou que des contrainte de temps de traitements sévères doivent être respectées ou encore s'il est nécessaire de produire un système de vision compact, avec une consommation d'énergie réduite.

1.2 Architectures dédiés à la vision

Réaliser un système dédié à une application offre l'opportunité de redéfinir complètement l'organisation matérielle de façon à améliorer l'efficacité du dispositif. Parmi les optimisations les plus notables il est possible de procéder à une délocalisation de la fonction de traitement, depuis l'extrémité finale de la chaîne de perception, vers une intégration au sein des dispositifs dédiés. Selon le niveau de délocalisation du traitement, intégré dans le boitier même de la caméra voire directement intégré à proximité de l'imageur, on parlera respectivement de *caméra intelligente*, ou de *rétine artificielle*.

1.2.1 Les caméras intelligentes

Une camera intelligente intègre dans le même boîtier un capteur vidéo et une chaîne de traitement numérique (consistant généralement en des étages conversion analogique/numérique et un ou plusieurs calculateurs (processeur, FPGA[1], DSP[2], ...).

- technologie CCD - Les capteurs employés sont souvent de type CCD[3] car ceux ci présentent des avantages importants : ils possèdent un meilleur ren-

[1]Field Programmable Gate Array : réseaux logiques programmables utilisés en général pour implémenter des traitements à base d'opérateurs logiques câblés, cf chapitre 3

[2]Digital Signal Processor : processeur spécialisé dans le traitement numérique du signal.

[3]Charge Coupled Device : technologie de capteur lumineux avec dispositif à transfert de charges.

dement des cellules dû à un facteur de remplissage élevé[4] (plus de 98%), une meilleure dynamique de signal, une grande immunité au bruit et une meilleure qualité globale d'image. Le mode de transmission des registres à transfert de charge ne permet pas d'accéder aux pixels dans un mode aléatoire.

- technologie CMOS - Une alternative à la technologie CCD est d'employer un imageur CMOS. En plus du progrès rapide de cette technologie en terme de qualité d'image, cette technologie apporte de nombreux avantages. Le premier est une plus faible consommation d'énergie que les CCD. Ensuite cette technologie permet d'ajouter d'autres fonctions à coté de l'électronique photosensible. Il ainsi possible de convertir, de corriger le signal voir même ajouter des traitements complexes. pour finir, son grand avantage est de permettre un accès aléatoire au pixel. Certaines caméras intelligentes [49] [20] [61] emploient ainsi des techniques de fenêtrage directement au sein du capteur dans le but réduire le flot de données entre l'imageur et les composants de traitement de la caméra intelligente.

Un dispositif d'imagerie CMOS permet un mode de lecture contrôlé de l'image au sein d'un système de vision active. Un autre avantage des imageurs CMOS est la possibilité technologique de fabriquer des rétines artificielles en rapprochant plus encore les traitements de la source d'information visuelle.

1.2.2 Les rétines artificielles

Les rétines artificielles sont des imageurs CMOS intégrant dans leur électronique (au sein même du pixel) des opérateurs de prétraitement avancé. Le terme de rétine artificielle est d'inspiration biologique, car dans de nombreuses espèces animales on retrouve un ensemble de transformations du signal lumineux perçu directement au sein de la rétine. Le terme de rétine artificielle est souvent aussi employé pour les imageurs intégrant une électronique active[5] pour adapter l'impédance du signal. Nous parlerons ici de circuits entièrement analogiques ou partiellement numériques et analogiques.

La première catégorie emploie des opérateur analogiques permettant de concilier compacité et consommation réduite aux prix d'une fonctionnalité et d'une flexibilité limitée. Les traitement d'images effectués sont généralement de bas niveau (filtrages, extraction de primitives, adaptation de luminosité,..). Ce type

[4] En anglais *fill factor*,rapport entre la surface photosensible de la cellule et la surface opaque.
[5] APS : Active Pixel Sensors

d'architecture permet d'atteindre des fréquences d'acquisition et de traitement très élevées (entre 2000 et 5000 images traitées par seconde sur un senseur 64×64[36]).

Les progrès en matière d'intégration permettent dorénavant de remplacer les opérateur figés par des Processeur Élémentaire (PE). On parle alors de rétine artificielle numérique programmable (RANP). Les PEs communiquent de proche en proche et effectuent une même opération (numérique ou analogique) sur chaque pixel ou groupe de pixel[43][68]. Tous les PEs sont commandés par une unité de contrôle, selon le modèle de traitement parallèle SIMD (cf §1.2.4).

L'objectif est alors d'implanter sur le même circuit l'imageur et l'essentiel des fonctions de traitements nécessaires à l'application visée. Le panel des applications visées est élargi par le caractère programmable de ces systèmes. Par exemple ont été développées des applications de localisation, de reconnaissance des formes, ou des fonctionnalités de suivi rapide[42].

Le principal avantage des rétines programmables est d'éliminer à la base le goulot d'étranglement classiquement rencontré pour les communications entre fonction d'acquisition et fonction de traitement avec une approche conventionnelle. Bien que leurs homologues biologiques bénéficient d'une disposition tridimensionnelle, les rétines artificielles sont contraintes par les technologies de fabrication planaires. Les éléments photosensibles et composants de traitements étant sur le même plan, l'ajout d'électronique active dans la cellule photo-réceptrice a pour conséquence de réduire le facteur de remplissage. Un facteur de remplissage plus faible à pour effet de diminuer la sensibilité du pixel et donc la qualité de l'image obtenue. Ce qui amène un compromis difficile entre qualité d'acquisition et puissance/diversité des traitements. Le facteur de remplissage se situe généralement entre 30% et 50% en fonction de la complexité de l'electronique associée aux éléments photosensibles.

1.2.3 Approches bio-inspirées

Certaines approches s'inspirent des solutions employées dans la nature pour réaliser la fonction visuelle. Parmi les approches s'inspirant de la vision humaine, la *vision active* introduit la notion de rétroaction entre le dispositif de traitement (le "cerveau") et celui d'acquisition des données images ("l'oeil"). Les approches dites *neuromimétiques* implantent des modèles de réseaux de neurones sur circuit électronique.

1.2.3.1 La vision active

Une camera basée sur le concept de vision active est réactive vis à vis de l'information visuelle. Elle met à profit le contrôle du mode de lecture aléatoire des pixels d'un capteur CMOS. Les résultats de l'analyse du signal vidéo sont ainsi utilisés afin d'optimiser dynamiquement le flux d'information. En accédant à l'ensemble des pixels (par exemple une ou des fenêtre d'intérêt) strictement nécessaire à l'application, le goulot d'étranglement de l'information entre le capteur et le système de traitement rencontré avec une approche conventionnelle est évité. Des applications de visions actives ont été réalisées sur la plateforme SeeMOS[20] et la plateforme PVLSAR[68].

1.2.4 Architectures parallèles pour la vision

Très souvent, les approches mettent à profit la nature intrinsèquement parallèle des applications de vision, et emploient des composants capables de traitements parallèles. L'*Adéquation Algorithme Architecture* consiste à étudier simultanément les aspects algorithmiques et architecturaux en prenant en compte leurs interactions afin de réaliser l'implantation optimisée d'un algorithme (en terme de coûts des composants logiciels et matériels et de performances) tout en réduisant les temps de développement.

1.2.4.1 Classification des machines parallèles

La classification la plus utilisée pour les traitements parallèle est celle proposée par Flynn[41]. Elle distingue quatre classes de machine :

- SISD - Simple flot d'Instructions Simple flot de Données : machine séquentielle
- SIMD - Simple flot d'Instructions Multiple flot de Données : machine vectorielle
- MISD - Multiple flot d'Instructions Simple flot de Données : peu utilisé
- MIMD - Multiple flot d'Instructions Multiple flot de Données : grappe de processeurs

1.2.4.2 Evolution des machines parallèles

Les premières machines capables d'effectuer des traitement de vision en temps réel appartenaient dans les années 1980 à la catégorie des super-calculateurs. Les super-calculateurs utilisaient pour la plupart un mode de calcul SIMD vectoriel où une même opération est appliquée simultanément par plusieurs unités de traitement à leur données. Un exemple typique est la *connection machine* CM-2[50]. Ces machines même si elles offraient une solution efficace au problème de la puissance de traitement, se révélaient particulièrement coûteuses et s'adressent uniquement aux problèmes comportant un parallélisme de données.

A partir des années 1990, dans pratiquement tous les domaines du calcul haute-performances, les super-calculateurs furent progressivement remplacés par des clusters, machines parallèles constituées d'unités de calcul indépendantes et peu coûteuses. Le mode de calcul utilisé est alors de type MIMD à mémoire distribuée. Chaque unité de calcul étant un processeur classique possédant sa propre mémoire de données.

Dans les années 2000, ce type d'architecture à connu une seconde jeunesse avec l'introduction de capacités traitement vectoriels dans les architectures de processeurs classiques (c'est à dire aussi bien dans le domaine des stations de travail que pour les ordinateurs grand public). Les extensions de jeu d'instruction tel que MMX,SSE ou AltiVec utilisent ainsi le concept de SWAR[6] pour permettre au processeur d'effectuer plus efficacement des traitements sur l'image, le son ou la vidéo. Depuis, les clusters sont alors des architectures hybrides intégrant plusieurs niveaux de parallélisme, associant un parallélisme gros grain entre processeurs à un parallélisme de moyenne granularité à l'intérieur du processeur. Un exemple récent de cette classe d'architecture est l'architecture BlueGene/L d'IBM[33].

Une forte tendance actuelle est encore d'ajouter à ces processeurs hybrides des composants FPGA servant de coprocesseurs. Ils permettent de mettre en œuvre un parallélisme à grain fin par l'emploi de structures pipelines mais aussi la réplication des unités fonctionnelles. La nature reconfigurable des coprocesseurs apportant une certaine dose de flexibilité par rapport à une architecture figée.

L'architecture Cray XD1[18] permet d'associer à chaque processeur (opteron) un module optionnel comportant un FPGA Xilinx. La technologie SGI RASC avec le RC100[45] permet aussi d'ajouter à leurs serveurs Altix (basés sur le processeur Itanium) un module comportant deux FPGA Xilinx. La tendance est donc à une évolution vers des architectures de plus en plus hétérogènes.

On a retrouvé cette même évolution des tendances générales avec les machines parallèles dédiées à la vision. L'expérience des machines TRANSVISION[32],

[6] SIMD Within A Register.

OSSIAN[67] et BABYLON[40] développées au LASMEA, ont été source de nombre des concepts de cette thèse.

La machine TRANSVISION de Type MIMD-DM basée autour d'un réseau de Transputers T9000 a introduit notamment le concept de noeuds vidéo avec une architecture pourvue de deux bus de communications. La machine OSSIAN conçue pour exécuter des algorithmes de vision et de traitement d'images dans un contexte embarqué, est un cluster de quatre Apple G4 Cube (de faible encombrement) à deux niveau de parallélisme, MIMD inter-noeuds, et SIMD intégré au processeur. La machine BABYLON reprend l'architecture générale d'OSSIAN et ses deux bus de communications : un bus ethernet dédié aux communications et un bus FireWire 1394a dédié au transfert des données vidéos en provenance d'une caméra numérique. Constituée de quinze machine PowerPC G5 biprocesseur, c'est une architecture hybride à trois niveau de parallélisme, MIMD inter-noeuds, SMP inter-processeurs, et SIMD intégré au processeur.

Les architectures de traitement des systèmes de visions embarqués ont évolué de la même manière vers des systèmes hétérogènes, alliant un processeur (typiquement un DSP) à un *accélérateur* (typiquement un processeur SIMD ou un FPGA). On retrouve bien entendu cette tendance à travers l'exemple des camera intelligentes.

1.3 Exemples de réalisations

1.3.1 La camera Philips inca+

La camera Philips inca+ est un prototype de caméra intelligente adapté à partir du modèle commercial inca (figure 1.1). Celle ci est architecturée autour d'un imageur CMOS, d'un processeur massivement parallelle "xetal" fonctionnant en mode SIMD et du processeur DSP trimedia de type VLIW. Un algorithme basé sur un réseau de neurone radial (RBF) est utilisé pour effectuer de la reconnaissance de visage[55].

1.3.2 La plateforme SeeMOS

La plateforme SeeMos[20](figure 1.2) est une camera intelligente conçue pour des applications de vision active. Elle dispose d'un imageur CMOS NC1802 de la société NEURICAM de résolution 640x480, et d'une centrale inertielle pour étendre les capacité de perception. L'architecture de traitement est hétérogène avec un

FIGURE 1.1 – La camera Philips inca

FIGURE 1.2 – La plateforme SeeMos

FPGA (Altera Stratix EP1S60) et un DSP (Texas TMS32066416). Cette plateforme à notamment été utilisée pour des applications de détection de mouvement et de suivi rapide de motif.

1.3.3 La rétine programmable PVLSAR

La rétine artificielle numérique programmable PVLSAR 34 développée à l'ENSTA (cf figure 1.4) est une machine massivement parallèle de 40 000 cellule associant à chaque élément photosensible un convertisseur analogique/numérique, une unité de traitement analogique, et d'un processeur élémentaire numérique. Le processeur est capable d'effectuer des opérations booléennes (''ET'', ''NON'', ''OU'', etc.) et dispose d'une mémoire d'environ 50 bits.

Les noeuds sont interconnectés suivant une topologie grille 2D 200x200, et

communiquent les données entre pixels voisins par partage de données. Le mode de calcul est purement SIMD.

Cette architecture est utilisée pour intégrer des mécanismes d'attention visuelle[68].

La Figure 1.3 fournit un schéma général de l'architecture du système composé de la rétine artificielle programmable et du cortex(contrôleur de la rétine + processeur hôte).

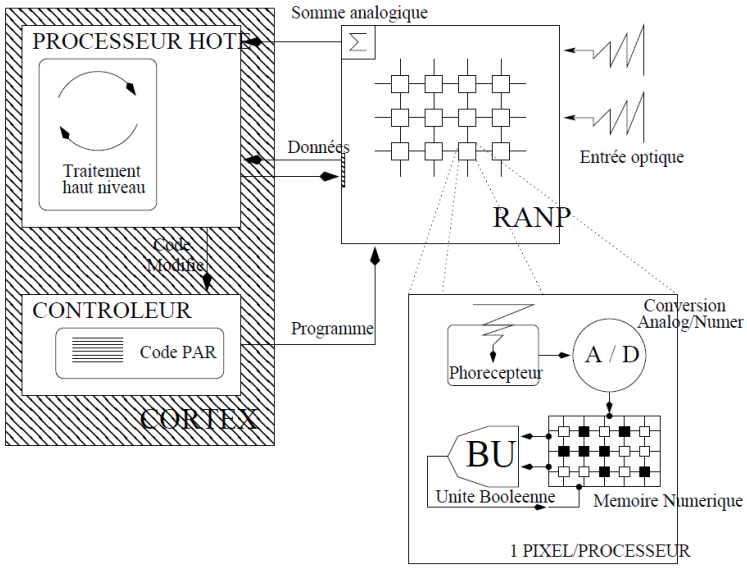


FIGURE 1.3 – Schema synoptique de la plateforme de vision PVLSAR

1.3.4 Camera intelligente haute fréquence haute résolution

Le laboratoire Le2i de Dijon à mis au point une camera intelligente dotée d'un imageur CMOS haute fréquence Micron MTM9M413 (500 images/s en 1280 $\times$ 1024, $\geq$10000 images/s en mode fenêtré 1280 $\times$ 128) lequel est associé à un FPGA Xilinx VIRTEX-II (XC2V3000) pour les traitements. Une interface USB 2.0 permet la visualisation et le stockage sur un système hôte standard (PC). Dans [61], cette camera intelligente est utilisée pour des applications de compression vidéo (JPEG2000, MPEG4) et de segmentation d'images.

1.4 Intérêt des systèmes monopuces

Les avancées en technologies micro-électronique, en particulier la possibilité de concevoir des systèmes intégrant pratiquement la totalité de leurs fonctionnalités

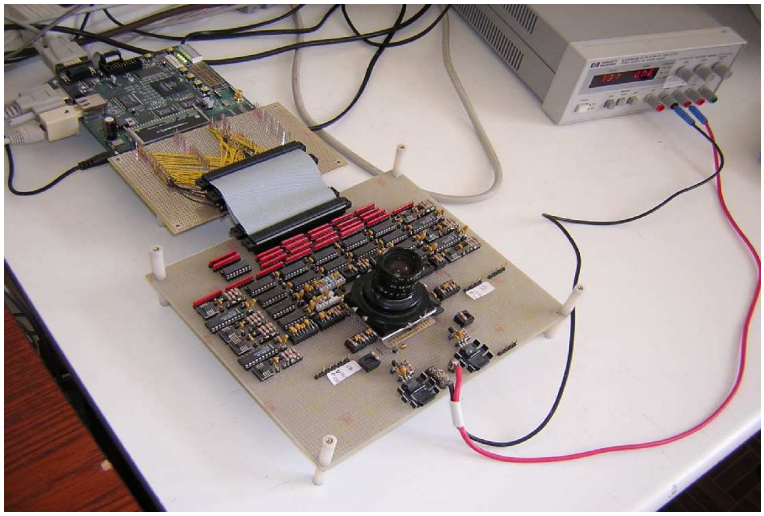


FIGURE 1.4 – Banc de traitement PVLSAR

au sein du même composant, offrent une piste particulièrement intéressante pour mettre en œuvre des capteurs intelligents ou rétines artificielles.

Les méthodes de conception dépendent fortement de l'application visée et du type de technologie employée. On peut par exemple considérer les rétines artificielles utilisant des opérateurs cablés comme appartenant à la catégorie des ASICs[7]. Les solutions programmées (employant des processeurs de type DSP[8], ASIP[9], ...) sont des systèmes embarqués qui sont constitués d'une partie matérielle et d'une partie logicielle.

Une difficulté est notamment de disposer de méthodes et d'outils de conception qui sont d'une part efficaces en terme d'effort de conception et qui d'autre part aboutissent à la réalisation d'architectures dotés d'une certaine souplesse d'utilisation et capables d'évoluer, leur assurant ainsi une certaine pérennité.

Le prochain chapitre présente les problèmes de conception de systèmes monopuces, et les différentes méthodes employées pour les adresser.

[7] Application Specific Integrated Circuit
[8] Digital Signal Processor.
[9] Application Specific Instruction set Processor.

Chapitre 2

Des systèmes monopuces aux multiprocesseurs embarqués.

Héritant des méthodes de conception d'ASIC, la conception de système monopuce (SoC) est l'approche la plus prometteuse pour embarquer des algorithmes de traitement d'images. Un des enjeux majeurs de la conception de SoC est la nécessité de faire évoluer rapidement les méthodes de conception afin de combler le fossé entre productivité des concepteurs et capacité d'intégration. En effet, les méthodologies de conception des systèmes électroniques pour des implantations d'algorithmes sur des systèmes monopuce sont relativement complexes, et la capacité d'intégration des circuits continue d'évoluer plus vite que la productivité des concepteurs. Il est alors de plus en plus difficile de tirer réellement profit de la capacité d'intégration. De plus, un nouveau défi auquel doivent faire face les concepteurs de SoC concerne la mise en oeuvre de système de communication complexe, l'approche classique (interconnection par un bus partagé) constituant un des principaux facteurs limitant les performances du système.

Dans une première moitié de ce chapitre, est rappelée l'architecture interne classique des systèmes sur puces. Un bref aperçu y est donné de l'éventail des possibilités rencontrées à la conception de chaque sous ensemble du système, ainsi qu'une introduction aux problèmes classiquement rencontrés. La question des systèmes de communication intégrés dans les SoC et notamment la notion de réseau intégré (NoC) y est approfondie. Dans une deuxième partie, sont présentées les stratégies employées par les diverses méthodologies afin d'obtenir un système satisfaisant les contraintes d'application tout en permettant d'améliorer la productivité.

2.1 Architecture d'un système sur puce

Les évolutions technologiques successives ont permis l'intégration de plus en plus de fonctions sur le même circuit intégré. Il est maintenant possible d'intégrer ensemble plusieurs processeurs, des mémoires, des accélérateurs matériels et des composants périphériques, le tout étant interconnecté avec un réseau de communication. Ceci permet de rassembler toutes les fonctions nécessaires pour effectuer des traitements complexes sur une puce.

2.1.1 Processeurs généraux (CPU)

La plupart des SoC contiennent maintenant un ou plusieurs processeurs à usage général. Les processeurs commerciaux destinés à être embarqués (ARM, MIPS, PowerPC, ST-Microelectonics, etc.) se distinguent des processeurs ordinaires par une architecture optimisée afin de minimiser la consommation d'énergie. Pour les applications nécessitant de la puissance de traitement, certains SoC utilisent des DSP optimisés pour le traitement du signal. Un exemple classique d'instruction de processeurs DSP est l'instruction MACC : Multiply and ACCumulate qui permet d'effectuer une multiplication et une addition à la fois. Les DSP disposent aussi d'une architecture permettant de traiter plus efficacement les flux de données en facilitant l'accès aux données avec par exemple l'adjonction d'un accès direct mémoire (DMA) ou la séparation du bus en bus d'instruction et bus de données (Architecture Harvard). Les principaux avantages de faire appel à des composants standards se situent dans le confort et la puissance des outils de conceptions qui leur sont associés.

Les processeurs sont disponibles sous formes de propriétés intellectuelles(IP[1]) qui peuvent être plus ou moins souples d'utilisation. On distingue ainsi les coeurs de processeur *hard* des coeurs de processeur *soft* en fonction de leur nature.

On parle de coeur processeur *hard*, quand le processeur est disponible sous forme de masque pour une technologie de type ASIC, ou alors intégré *en dur* dans un FPGA/SOPC ou un circuit mixte comprenant à la fois une partie reconfigurable et non-configurable. L'inconvénient d'utiliser un processeur *hard* est que cette approche est peu flexible. Les choix sont limités par la disponibilité des composants dans le marché. Un processeur choisi sur le marché sera ainsi en général plus puissant que nécessaire, et pourra consommer plus d'énergie ou être plus volumineux qu'un processeur qui aurait été conçu spécifiquement pour l'application (ASIP).

[1] IP provient de l'anglais Intellectual Property.

Les processeurs *soft* (ou processeurs virtuels) par opposition aux processeurs *hard* classiques, sont disponibles sous formes de sources décrites au niveau RTL. Certains processeurs sont aussi configurables et disposent d'unités fonctionnelles optionnelles qui permettent de faire varier leur architecture en fonction du besoin.

Les processeurs *soft* commerciaux proviennent essentiellement des fabriquants de circuits reconfigurables avec le Microblaze de Xilinx[79] et le NIOS d'Altera[9]. ARM propose également des versions *softcore* de ses processeurs ARM[3].

Il existe aussi un nombre relativement élevé d'implantations ouvertes et librement disponibles[65]. La plupart de ces processeurs *soft* sont en fait des *clones* de processeurs commerciaux disposant d'un jeu d'instructions compatible avec l'original. Ainsi les processeurs Openfire[25] et le Leon[53] sont respectivement compatibles avec les processeurs Microblaze et Sparc.

Les processeurs virtuels offrent d'avantage de flexibilité que les processeurs classiques, outre la possibilité de les intégrer sur technologie reconfigurable, ils sont en général paramétrables, avec des unités fonctionnelles optionnelles (Unités arithmétiques, ou de gestion mémoire par exemple), permettant de les adapter aux besoins applicatifs.

2.1.2 Accélérateurs matériels

Les accélérateurs matériels sont des fonctions de traitement intensif (FFT, décodage/encodage vidéo, etc.) ou de contrôle d'entrée/sortie (UART, VGA, PS2, etc.) qui sont effectuées sur un composant matériel dédié à cet usage afin de soulager un processeur (ou microcontrolleur) ou pour en retirer un gain de performance. Ces composants peuvent être des composants matériels conçus à l'occasion ou des propriétés intellectuelles (IP) réutilisables. La tendance est à une utilisation de plus en plus massive d'IPs lors de la conception de système embarqué. Nous évoquerons par la suite les méthodes proposées pour permettre d'augmenter encore la réutilisation des IPs. Les principaux obstacles à la réutilisation étant le problème d'adaptation entre interfaces et protocoles incompatibles, et le besoin de disposer d'IP le plus paramétrable possible.

2.1.3 Composants de mémorisation

2.1.3.1 Technologies de mémoires

Dans les systèmes sur puce, on utilise de préférence des mémoires internes car elles offrent de meilleures performances que les mémoires externes en terme de latence et de bande passante. Les mémoires internes permettent d'accélérer les

accès aux données mais elles occupent une surface de silicium importante. Les éléments de mémorisations sont très diversifiés et on retrouve dans un SoC aussi bien des mémoires linéaires (piles et files) que des mémoires adressables et non adressables (caches) et aussi bien de la mémoire morte (ROM) que de la mémoire vive. Les deux technologies de mémoires sont disponibles, les mémoires statiques (SRAM) volumineuses[2] mais performantes et plus récemment la DRAM enfouie (eDRAM), plus dense [3] mais moins performante que de la SRAM. Enfin le dimensionnement des bancs mémoires et de leurs contrôleurs a aussi un impact sur les performances et l'intégration : l'emploi de petites mémoires permet d'améliorer les performances mais fait augmenter le nombre de contrôleur, à l'inverse utiliser de larges bancs mémoire permet d'améliorer la densité d'intégration mémoire au prix de moins bonnes performances.

2.1.3.2 Architecture mémoire conventionnelle

L'emploi de différents niveaux de mémoires organisées en couches successives s'est généralisé dans les systèmes de microprocesseurs standards pour concilier au mieux performances et capacité de stockage. Typiquement dans une architecture de processeur universel sont identifiés, (du plus rapide au plus volumineux) les registres du processeur, plusieurs niveaux de caches interne (L1 et L2), et à l'extérieur du microprocesseur du cache externe (L3) et la mémoire principale. Cette organisation peut se retrouver aussi dans les systèmes embarqués. Les besoins spécifiques aux SoC, notamment en terme d'efficacité énergétique, nécessitent cependant de repenser les architectures mémoires conventionnelles.

2.1.3.3 Architecture mémoire dédiée

Pour un système spécialisé, la connaissance des besoins exacts de l'application permet d'ajuster finement la capacité mémoire au besoin, et d'élaborer des stratégies d'optimisation de la mémoire, d'une part en optimisant l'allocation et les flux de données dans l'application et d'autre part en dimensionnant précisément la capacité mémoire requise[59] et les flux de données mis en jeu par l'application. En retenant les opérandes en mémoire interne, il est montré[17][69] que la bande passante sera utilisée plus efficacement.

Dans le cas d'une architecture multiprocesseur se pose enfin l'épineux problème du choix entre mémoire partagée et mémoire distribuée. Le grand avantage d'une mémoire partagée est de laisser la possibilité de conserver des techniques de

[2]La mémoire standard SRAM 1 bit utilise 6 Transistors

[3]La mémoire standard DRAM 1 bit utilise 1 transistor et 1 capacité

programmation conventionnelles. Le principal inconvénient de cette architecture de mémoire est que l'accès à la mémoire devient un goulot d'étranglement, quand cette mémoire est associée à un bus partagé. Beaucoup d'architectures à mémoire partagée sont par conséquent en réalité des mémoires distribuées-partagées où les mémoires sont physiquement distribuées, mais avec un mécanisme, (logiciel, matériel ou hybride) se chargeant de traduire les adresses et de gérer la cohérence des données partagées dans les différentes mémoires locales. Ce mécanisme s'il est implanté intégralement en matériel est très complexe à concevoir, notamment en raison de la gestion des problèmes de cohérence des données entre les caches[62][47]. Une mémoire purement distribuée, en plus de performances accrues présente l'avantage de simplifier considérablement la conception de l'architecture mémoire dans une architecture multiprocesseur. La contrepartie est alors l'existence de plusieurs espaces d'adressages (un par processeur) et la nécessité d'utiliser un autre modèle de communication (par exemple le passage de message) rends la conception logicielle particulièrement complexe.

2.1.4 Systèmes de communication

Le système de communication relie physiquement les composants les uns aux autres, permettant ainsi le transit d'informations de contrôle, ou de données entre ces composants. Les communications entre les blocs fonctionnels d'un système monopuce ont en premier été assurées par des architectures basées sur des bus.

2.1.4.1 Bus unique partagé

Le concept de Bus associe une topologie simple à un standard d'interconnexion permettant de faciliter la modularité. Les fabriquants précurseurs dans la réalisation de SoC ont chacun proposé leur standard de bus. Le principal enjeu était alors de fournir un standard solide, facilitant l'intégration et la réutilisation de composants IP, tous compatibles avec l'interface de bus en question.

Les avantages d'un bus sont un coût faible en terme de nombre de connections et un taux d'utilisation important parce que un seul canal est partagé par les noeuds. Un autre avantage réside dans les gains de productivité apportés par l'emploi d'IPs compatibles avec le standard d'interconnexion.

Son principal inconvénient est qu'un bus n'est pas extensible car sa bande passante est partagée entre les différents noeuds connectés à celui-ci. Un bus simple ne permet pas de multiples communications en concurrence et ses performances se dégradent avec l'ajout de noeuds supplémentaires dans le réseau (on dit qu'il n'est pas extensible). De plus le nombre d'éléments connectés entraine l'augmen-

tation de la longueur des lignes de bus et pose un problème de capacités parasites, réduisant la fréquence de fonctionnement et augmentant la consommation électrique.

Par conséquent cette topologie est de moins en moins adaptée aux problèmes de communications dans les SoCs, et particulièrement dans le cas d'architectures multiprocesseurs exécutant des algorithmes parallèles.

2.1.4.2 Bus hiérarchique

Les architectures que l'on rencontre désormais couramment sont de type Multi-bus (aussi appelé bus hiérarchique). Un Multi-bus consiste en un certain nombre de bus locaux indépendants reliés par des ponts. Ce type d'interconnexion permet des communications concurrentes multiples. Il est nécessaire de bien repartir les IP sur les différents bus pour exploiter la localité des communications, et limiter ainsi les latences dues à des transferts entre deux bus par le pont. Les différents segments d'un multi-bus ont des liaisons plus courtes et moins de composants connectés qu'un bus unique, ce qui en améliore les caractéristiques en terme de consommation et de fréquence. L'architecture Multi-bus réduit ainsi certains inconvénients du bus mais n'est cependant toujours pas extensible.

L'architecture IBM coreconnect[4] et les premières versions du standard AMBA d'ARM[2] sont deux exemples d'architecture Multibus. Le standard AMBA est par exemple constitué de deux types de bus. Le premier sert à connecter des ressources nécessitant un débit élevé (le bus AHB : Advanced Highperformance Bus), le second à connecter les périphériques utilisant un débit moindre (le bus APB : Advanced Peripherial Bus). Suivant le même principe, le standard IBM Coreconnect sépare un bus PLB, de haute performance et destiné aux processeurs et les bus OPB et DCR destinés à piloter des périphériques plus lents.

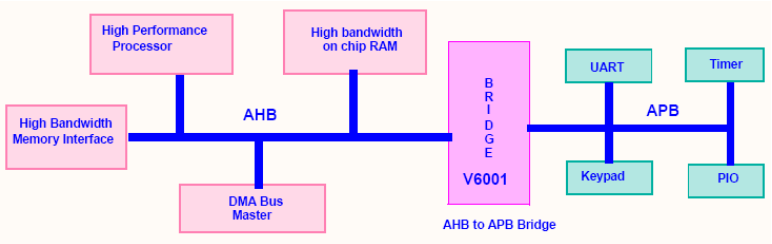


FIGURE 2.1 – Le standard de bus AMBA.

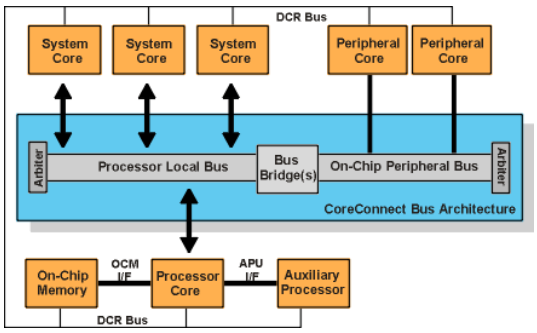


FIGURE 2.2 – Le standard de bus coreconnect.

2.1.4.3 Bus avancés et architectures hybrides

Les acteurs industriels font progressivement évoluer leurs standards de communication dans le but de repousser les limites des architecture de bus classique. Des bus avancés comme le bus *Silicon Backplane*[5] de Sonics et AMBA 3 AXI[11] disposent disposent de canaux multiples et/ou sont pipelines. Avec le bus Avalon[1] d'Altera, il est possible de faire varier le nombre de canaux de telle manière que la solution extrême devient équivalente à un crossbar.

Le cas le plus emblématique est le STBUS[6] de STmicroelectonics, qui a évolué en intégrant au fur et à mesure des dispositifs de communications de plus en plus élaborés. Dans son implantation la plus simple le STBUS est un bus partagé mais qui peut tout aussi bien être une matrice d'interconnection de type crossbar, ou encore intégrer dans son mode le plus avancé un véritable NoC avec le Spidergon[23], successeur de l'Octagon[54]. Il s'agit en réalité d'un environnement de synthèse permettant de générer des bus d'interconnexions basés sur un composant central appelé noeud. Un noeud permet à plusieurs initiateurs d'adresser plusieurs cibles. Les différents noeuds générés sont cascadables et l'environnement prend en charge la génération des passerelles pour assurer les communications entre noeuds de tailles différentes. Selon le cas de figure, l'une des formes cités précédemment sera sélectionnées.

2.1.4.4 Les réseaux embarqués

Avec la complexification des SoC, les besoins en terme de ressources de communication (bande passante, latence,extensibilité, ...) et en qualité de service (QoS) sont de plus en plus sévères. Ceci pousse les acteurs industriels à faire évoluer leurs standards vers des solutions de plus en plus évoluées. De nombreux travaux de recherche sont de plus effectués afin d'offrir de nouvelles solutions dans le but d'améliorer les interconnexions entre modules ou IP.

Ces travaux[26][10][44] montrent alors que l'utilisation d'un réseau embarqué (ou NoC de l'anglais *Network on a Chip*) devient une approche de plus en plus appropriée au fur et à mesure qu'augmente le nombre de processeurs dans les systèmes embarqués.

Le premier intérêt d'utiliser un NoC est de structurer les communications. L'utilisation d'interfaces standard augmente la modularité des IPs.

Les réseaux distribués existent déjà depuis longtemps dans les réseaux d'ordinateurs. La plupart des notions et modèles utilisés dans les réseaux embarqués sont ainsi une adaptation au contexte des SoC des solutions normalisées issues du monde des Technologies de l'Information et de la Communication.

Eléments de base d'un NoC :

Un Noc comprends les élements de base suivants :

- les routeurs qui gèrent l'acheminement des données dans le réseau en accord avec le protocole de routage choisi.
- les liens qui connectent deux à deux les différents routeurs. Ils peuvent être mono ou bidirectionnel, ce sont eux qui déterminent la bande passante entre les ressources.
- les adaptateurs réseau (NA) qui servent d'interface entre le noeud de traitement (bloc IP ou processeur) et le routeur.

Les couches réseau :
Les NoCs utilisent un modèle de couches inspiré du modèle de couche réseau OSI (Open System Interconnection), qui est une norme ISO. Cette norme mis en place depuis 1984 est basée sur la représentation de 7 couches distinctes et fait appel aux notions de service, de protocole et d'interface ainsi qu'au principe d'encapsulation. On peut voir sur la figure 2.3 le transit d'une donnée entre deux IP et les différents éléments de NoC traversés, ainsi que les couches OSI correspondantes. Une adaptation du modèle OSI réduite en trois couches a été proposée par Luca Benini et Giovanni De Micheli dans [29].

Topologie :
Dans un réseau d'interconnection, la topologie définit l'organisation spatiale et les relations entre les différents noeuds du réseau sous forme d'un graphe. Une topologie est associées à un certain nombre de notions ou propriétés :

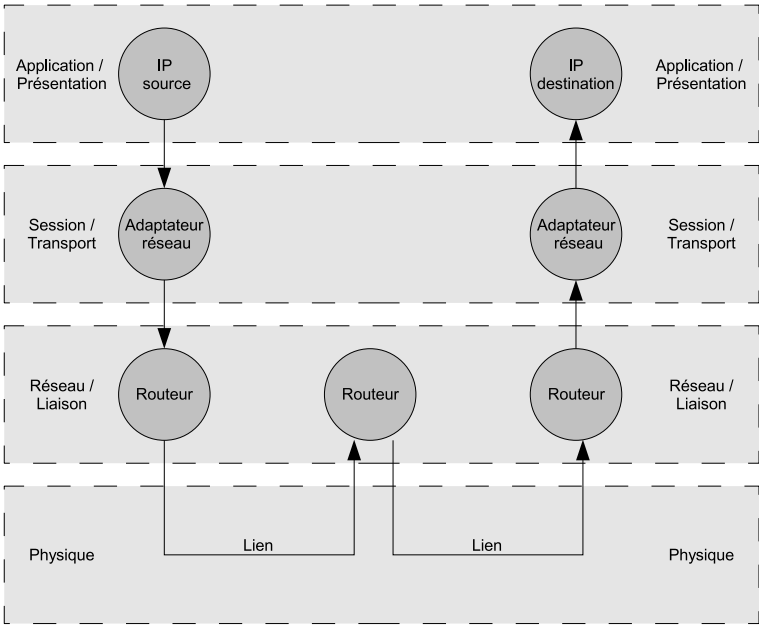
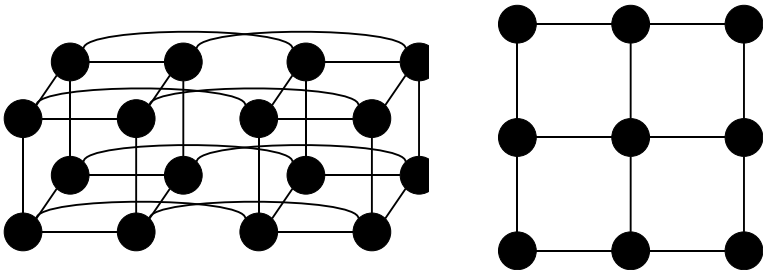


FIGURE 2.3 – Couches réseau et élements d'un NoC correspondant.

- **Le degré** d'un réseau est le nombre de canaux qui relie chaque noeuds a ses voisins. Si le degré est constant on dit que le réseau est de degré régulier.
- **Un chemin** dans le graphe est un ensemble de noeuds tous reliés consécutivement entre eux.
- **La distance** entre deux noeuds est le plus petit nombre de canaux emprunté consécutivement entre deux noeuds.
- **Le diamètre** d'un réseau est la distance maximale entre deux noeuds du réseau.
- **La bissection** d'un réseau est le nombre minimum de canaux qu'il faut enlever pour partitionner ce réseau en deux sous-réseaux contenant chacun la moitié des noeuds.

Ces propriétés ont une incidence directe sur les coûts de réalisation du réseau et de ses performances. De manière générale, utiliser un réseau fortement connecté



(a) hypercube dimension 4 (b) grille 2D (3,3)

FIGURE 2.4 – Exemples de topologies de réseaux directs.

(de degré élevé) nécessitera plus de ressources, et pourra poser des problèmes de routage sur silicium. Mais en même temps cela permet de diminuer le diamètre du réseau et donc d'augmenter les performances de communication. La technologie 2D des circuits intégrés favorise les topologies qui peuvent se réaliser dans un plan (ce qui implique un degré faible).

Le contenu d'un noeud dépend du type de réseau. On distingue classiquement les réseaux directs où chaque routeur est appareillé à une ressource, des réseaux indirects où certains routeurs ne sont pas directement connecté à une ressource. Dans un réseau direct, le routeur et la ressource appartiennent à un même noeud, tandis que dans un réseau indirect, chaque routeur et chaque ressource est considéré comme un noeud différent. Les topologies rencontrées dans la littérature sont souvent régulières, les plus populaires sont la grille 2D, la topologie en anneau, en arbre et l'hypercube illustrées dans la figure 2.4.

L'intérêt d'utiliser une topologie régulière est de simplifier la modélisation, ce qui permet de définir des règles de numérotation simples, et de simplifier les stratégies de routage, mais aussi de disposer de modèles de performances bien établis. Il est possible, en choisissant une topologie irrégulière, de tailler le réseau de communication complètement sur mesure. Cela permet d'étendre les possibilités d'optimisation, mais cela complexifie le problème du routage et rend plus difficile la modélisation des performances.

Problèmes de communication

Dans un réseau, le message dans son format de transmission doit traverser plusieurs routeurs avant d'atteindre sa destination. Malgré l'existence de plusieurs chemins de communication, plusieurs situations peuvent l'empêcher d'atteindre

sa destination finale. Ces situations proviennent du fait que les ressources du réseau sont finies et que les différentes communications sur le réseaux peuvent requérir l'utilisation d'une même ressource.

Quand le paquet est bloqué indéfiniment au niveau d'une ressource intermédiaire on parle d'une situation de *blocage mortel* (*deadlock*). Quand le paquet circule indéfiniment dans un chemin cyclique ne menant pas à destination arrive une situation de (*livelock*). Cela arrive quand les ressources menant à la destination sont occupées et obligent la communication à prendre un autre chemin. Pour prévenir toute situation de *livelock*, il est suffisant d'utiliser une stratégie de routage garantissant un chemin minimal de communication. Le seul inconvénient de l'utilisation de chemin de routages minimaux est la tolérances aux fautes. Dans le cas d'un routage non-minimal les *livelocks* peuvent aussi être prévenus en limitant le nombre de changements de route.

Il y a une situation de *famine* quand l'accès à une ressource est indéfiniment refusé au paquet (*starvation*), ce qui est souvent du à une mauvaise politique d'arbitrage. Le problème peut être résolu avec un arbitrage accordant de temps en temps les ressources aux communications les moins prioritaires.

Les impasses sont les situations les plus compliquées à résoudre. Il existe trois catégories de stratégies permettant de garantir que les communications ne se trouveront pas en impasse : la prévention, l'évitement et la récupération. Par une stratégie de *prévention*, les ressources du réseau sont définies avant le début de la communication, de manière à garantir l'absence de *deadlock*. Cela peut se faire par exemple en réservant toutes les ressources avant le début de la transmission. La stratégie d'*évitement* des impasses de communication, utilise une stratégie de contrôle de flux de communication prenant en compte l'état global du réseau. Les ressources de communication sont allouées au fur et à mesure de l'avancement de la communication uniquement quand une allocation garantie la conservation d'un état global sûr. Avec une stratégie de récupération, les ressources de communication sont allouées sans aucune vérification, jusqu'à l'arrivée d'une situation d'impasse. Après la détection d'une impasse, certaines ressource sont désallouées afin de permettre la reprise des communications impliquées dans la situation d'impasse. La technique de récupération, employée par les protocoles du réseau internet, nécessite de procéder à une nouvelle émission des données perdues lors de la désallocation d'une ressource de communication.

Ces stratégies pour garantir des communications sûre sont décrites avec plus de détails dans [28],[27],[34] et [35].

La deuxième partie de ce chapitre est consacrée à la description des approches de conceptions envisageables et de leurs caractéristiques.

2.2 Approches de conception des systèmes monopuces

Il existe différentes approches de conception de SoC qui ont chacune leur spécificités, leurs avantages et leurs inconvénients.

2.2.1 Conception logicielle/matérielle par une approche codesign

La conception conjointe matérielle/logicielle est apparue dès le début des années 90. Dans un souci d'abaisser les coûts et d'augmenter la flexibilité d'un système spécifique, on peut faire le choix d'implanter certaines fonctions en logiciel et les fonctions les plus contraignantes au niveau "performances" en matériel. Les méthodologies dites de *hardware/software codesign* proposent dans la phase de spécification de *partitionner* le système, c'est à dire de décider quelles parties seront conçues en logiciel et quelles parties seront conçues en matériel. Une fois le système spécifié et partitionné, il est alors possible pendant la phase de conception de développer conjointement matériel et logiciel. Cette méthodologie nécessite la coopération et la collaboration de deux équipes de développement (matériel et logiciel) afin de réduire les cycles de développements.

Les méthodologies de codesign présentent l'avantage de pouvoir effectuer des évaluations du système très tôt dans le flot de conception (dès la fin de la phase de spécification qui se matérialise par l'élaboration d'un prototype virtuel) Des étapes intermédiaires de vérification peuvent être effectuées au fur et à mesure de de la progression de la conception par le biais de la *co simulation*.

2.2.2 Réutilisation des IPs

Pour gagner en productivité, la réutilisation massive de composants matériels et logiciels précédemment conçu est une pratique qui s'est généralisée avec les System on a Chip. Cette approche de conception pour la réutilisation (en anglais *Design for Reuse* repose sur le principe d'assemblage de composants existants (IP) et la mise en oeuvre de techniques pour étendre les possibilités de réutilisation.

Un des obstacles à la réutilisation des IP est l'existence de nombreux standards d'interfaces.

En effet à l'heure actuelle, chaque acteur industriel a en général défini son propre protocole, et a développé un ensemble d'IP conformément à ce protocole, mais incompatible avec le protocole d'un autre constructeur.

L'assemblage de composants qui ne sont pas directement compatibles est alors impossible sans l'ajout de pont d'interfaces de protocole (*wrappers*). La réalisa-

tion manuelle de telles interfaces nécessite un effort de conception qui réduit les bénéfices de la réutilisation du composant.

Il est alors nécessaire d'utiliser des méthodes d'assemblage automatisé de composants aux interfaces hétérogènes.

Ainsi pour la phase d'implantation l'*approche composant* se base sur la connexion de blocs préconçus (IPs) avec des interfaces "universelles" telles que la norme VCI[77] ou OCP[64]. L'utilisation de composants et d'interfaces "virtuelles" encapsulant les éléments réels permettent de générer automatiquement ces "wrappers" et permettent d'étendre les possibilités de réutilisation des IPs.

Cesario et Al.[19] proposent une approche de codesign orienté "composants" qui en partant d'un modèle architectural virtuel composé de fonctions matérielles et logicielles, automatise aussi bien la génération d'interfaces (ou *wrappers*), que la génération de couches de logiciel intermédiaires tels que des drivers, le système d'exploitation et des APIs.

2.2.3 Réutilisation d'une plateforme de référence

Cette approche de conception appelée en anglais *Reference Platform-based Design* étend le concept de réutilisation au delà de la réutilisation des IP puisque il s'agit là de réutiliser des architectures voire des systèmes pour plusieurs applications.

Dans [56] et [22], une plateforme est définit par une ossature plus ou moins extensible et des sous ensembles (IP logiciels et matériels) conçus pour être réutilisés tels quels ou pour faciliter les modifications. L'environnement *Platform studio* intégré dans la suite EDK de Xilinx[79], permettant de développer des systèmes sur circuit reconfigurable (SoPC) est aussi conçu pour faciliter la réutilisation de plateformes de référence.

Le principal avantage de ce type de méthodologie est de permettre de dériver rapidement de nouvelles applications à partir d'une plateforme unique, en procédant à des modification marginales dans le système. Par exemple en ajoutant au système uniquement un accélérateur matériel spécifique à la nouvelle application visée. L'inconvénient de ces approches est que le modèle architectural est prédéterminé, c'est à dire que l'architecture du système est obtenue indépendamment de l'application visée.

2.2.4 Conception matérielle par la synthèse de haut niveau

Les méthodes de synthèse de haut niveau sont dédiées à une conception exclusivement matérielle.

Les techniques de synthèse d'architecture (appelées aussi compilation de silicium), sont généralement basées sur un modèle comportemental ou fonctionnel, un langage de programmation de haut niveau (C, fortran)[13] ou la représentation un modèle flot de données avec l'outil Syndex-IC[63].

Ces techniques sont en général aussi employées pour la conception d'un composant particulier au sein d'un SoC, par exemple la conception d'un ASIP[4], ou d'un accélérateur matériel.

2.2.5 Conception d'un ASIP

Il est possible d'opter pour la mise en oeuvre d'un processeur conçu spécifiquement pour l'application. L'intérêt d'une telle approche est que ce processeur sera parfaitement dimensionné par rapport aux besoins, alors qu'un processeur choisi sur le marché sera en général surdimensionné. L'inconvénient de cette approche est qu'elle est à priori particulièrement couteuse en terme d'effort de conception. En effet, il faut bien entendu concevoir le processeur mais sa programmation ultérieure suppose de disposer d'un certain nombre d'outils de développement logiciel (compilateur, débogueur, profileur, ...)

Certains industriels proposent des coeurs de processeurs particulièrement flexibles, comme par exemple le processeur Xtensa[73] de Tensilica, dont le jeu d'instructions et de nombreuses options architecturales peuvent être "taillée" en fonction des besoins.

L'approche extrême consiste à définir intégralement l'architecture du processeur. De nombreux travaux de recherches ont été effectués afin de faciliter la spécification d'un ASIP. Ils ont aboutis à l'émergence des langages ADL[5][46][70][80][78], dédiés à la description d'architectures de processeur. Avec un tel outil, il est possible de spécifier le processeur (son architecture, son jeu d'instructions, etc) avec une description de haut niveau et d'en évaluer les caractéristiques par simulation. Les ADLs ont ainsi contribué à l'accélération de *l'exploration d'architecture* qui consiste à rechercher dans l'ensemble des variations architecturales possibles celles qui répondent le mieux aux critères de l'application.

La conception d'un ASIP à partir de l'utilisation d'un ADL ouvre ainsi des perspectives particulièrement riches. Cependant la plupart des outils disponibles à l'heure actuelle ne permettent pas encore d'obtenir automatiquement une description synthétisable et optimisée du processeur (en langage RTL). Une étape de réécriture ou de raffinage manuel des sources au niveau RTL est ainsi nécessaire

[4] Application Spécific Instruction set Processor

[5] Architecture Design Languages

afin d'obtenir une version optimisée du processeur. Certains langages ont rapidement été intégrés dans des environnements commerciaux comme par exemple *Processor designer*[24] de Coware, issu de l'ADL LISA[70].

2.2.6 Conception de MP-SOC

Si l'idée d'intégrer un nombre élevé de processeurs au sein d'un même composant n'est pas complètement nouvelle[62], les capacités d'intégrations ont longtemps imposé une architecture bus[47] ou crossbar[16] et mémoire partagée. L'évolution des taux d'intégration permet depuis peu de faire communiquer plusieurs processeurs par un réseau d'interconnexion câblé[52] mais permet aussi des architectures mémoires plus faiblement couplées. Il est alors possible d'envisager la conception d'une véritable "architecture parallèle sur silicium".

Chapitre 3

Une méthode de conception de MP-SoC homogènes et réguliers pour des applications parallèles

Les architectures multiprocesseurs font partie des solutions capables d'exécuter efficacement les applications de traitement d'image, qui contiennent une part élevée de parallélisme. La méthodologie de conception de SoC développée dans ce mémoire permet de définir et de mettre en oeuvre un système monopuce multiprocesseur optimisé pour une application de traitement vidéo en mettant à profit le parallélisme de l'application. Elle ne nécessite pas une étape de partitionnement logiciel/matériel car tous les traitements sont placé sur des processeurs, elle remplace cette étape par une étape de parallélisation. Cette méthode peut donc être vue comme une alternative à une méthodologie de CoDesign classique. Elle cible une technologie de type FPGA ou SOPC, mais peut être déclinée de manière à cibler d'autres technologies ASIC. Les architectures réalisées sont de type Réseau Homogène de Processeurs Communicants (RHPC) et l'implantation des algorithmes utilise les principes de la programmation parallèle.

En première partie de ce chapitre, nous présentons la méthodologie. Celle ci est structurée en deux grandes étapes, la première étant la parallélisation de l'algorithme et la seconde étant la recherche et la réalisation d'une architecture matérielle optimisée pour l'application basée sur un RHPC. Nous donnerons un rapide aperçu des travaux sur les outils d'aide à la parallélisation sur lesquels se basent la première étape de la méthodologie. Les travaux présentés dans ce mémoires étant concentrés sur l'étape de conception matérielle, nous présenterons cette dernière étape plus en profondeur.

3.1 Objectifs visés et principe

Cette méthodologie s'inscrit dans le cadre du prototypage rapide et celui de l'Adéquation-Algorithme-Architecture.

Les objectifs sont donc les suivants :

- Obtenir une architecture optimisée pour l'application.
- Obtenir une implantation efficace de l'algorithme sur l'architecture.
- Permettre le test dans un délai très court en réduisant le nombre d'étapes intermédiaires dans le processus de réalisation.

3.2 Le concept de Réseau Homogène de Processeurs Communiquants

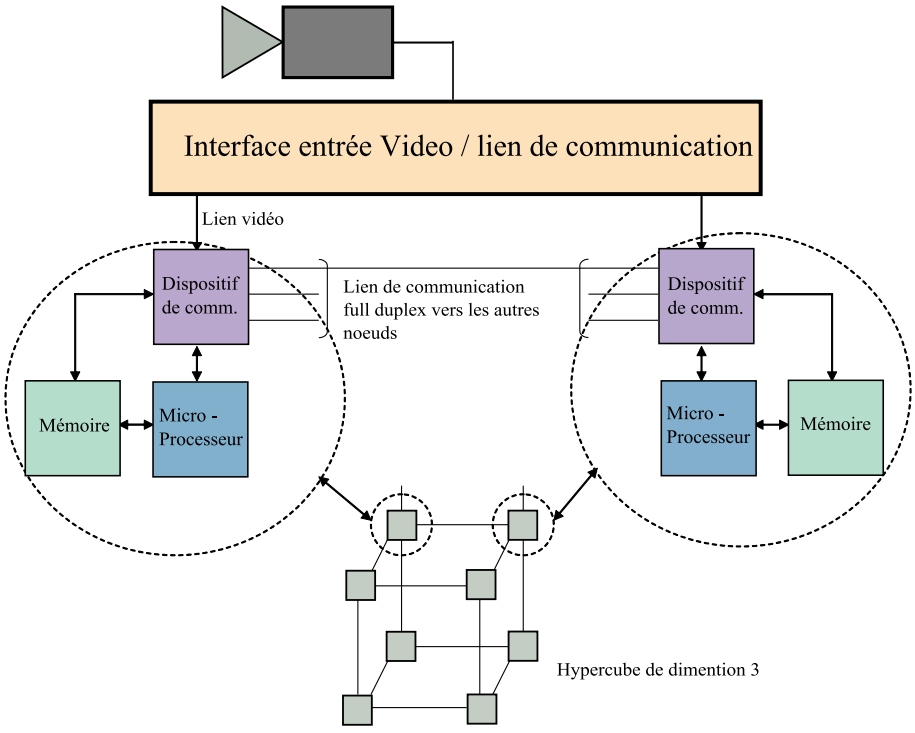


FIGURE 3.1 – Réseau Homogène de Processeurs Communiquants : exemple de topologie Hypercube 8 noeuds avec dispositif d'entrée vidéo dédié.

Les architectures mises en oeuvre avec la méthodologie proposée sont de type Réseau Homogène de Processeurs Communiquants. Elles ont toutes en commun les caractéristiques suivantes :

- Tous les noeuds de traitements sont identiques entre eux.
- Les noeuds sont interconnectés par des liens bidirectionnels.
- Le réseau formé est direct et à topologie régulière.
- Tous les noeuds possèdent un mode d'accès uniforme aux données vidéo.

Chaque noeud est composé d'un unique processeur d'une mémoire locale et d'un dispositif de communication, chargé de faire l'interface entre le processeur et le lien de communication et éventuellement la mémoire si ce dispositif de communication est doté de capacité d'accès direct à la mémoire. Une architecture RHPC devra être dotée d'un dispositif d'entrée vidéo permettant de permettre l'accès au données vidéo directement à chaque noeud. La figure 3.1 illustre le concept de RHPC avec une topologie Hypecube de 8 noeuds.

Les machines parallèles de types RHPC se classent donc parmi les machines MIMD avec mémoire distribuée. Le modèle de communication est le passage de message, qui peut être mis en oeuvre en utilisant une bibliothèque de communications par le passage de message comme MPI [7].

3.3 Description globale du flot de conception

Les différentes étapes de notre méthode sont illustrées dans la figure 3.2. Nous présentons dans cette section notre méthode de conception telle qu'elle serait dans l'idéal, si l'intégralité des éléments nécessaires à sa mise en oeuvre étaient dors et déja disponibles.

Le point d'entrée de la méthode est un algorithme séquentiel, classiquement programmé en langage C/C++. Le point de sortie est l'implantation et le test de cet algorithme sur RHPC.

La première étape consiste en une parallélisation de l'algorithme séquentiel.

A ce niveau du flot de conception l'architecture matérielle finale n'est pas encore définie. La parallélisation est effectuée en ciblant d'abord une architecture de type cluster de station de travail.

Le but de cette étape est double, il s'agit de reformuler l'algorithme de manière parallèle, mais il s'agit aussi à travers cette étape de parallélisation, d'extraire les

caractéristiques de l'algorithme parallèle afin de s'en servir pour définir une architecture parallèle optimisée pour l'application :

- Analyser la nature des opérations effectuées, permet par exemple d'extraire des informations sur le format et la précision des calculs (opérations logiques, calculs en nombres entiers, ou en nombres à virgule flottante) ce qui donne des critères pertinents pour choisir une architecture de processeur.
- L'analyse du code et le profilage de l'application permettent d'évaluer les effets du conditionnement et de la localité des données ainsi que la consommation en ressource mémoire pour chaque processeur.
- De la même manière, l'évaluation des besoins en communications peut servir de critère pour le choix des dispositifs de communications, aussi bien sur la nature matérielle des liens que sur la topologie du réseau (hypercube, anneau...).

A la sortie de cette étape le programme parallélisé est un code C qui doit être compilé pour chaque processeur. Après compilation, l'exécutable binaire devra être placé dans la mémoire locale de chaque processeur.

La seconde étape est une étape d'exploration de l'espace des solutions architecturales. Le concepteur a à sa disposition un ensemble de composants matériels sous forme d'une librairie d'IP synthétisables (en langage VHDL) comprenant les processeurs, les composants de communications, les composants d'entrée/sortie, etc.

Les éléments de choix pour définir l'architecture matérielle sont nombreux :

- Le nombre de noeuds dans le réseau.
- Le choix du mode de communication (type de liaison, avec/sans routage hardware, avec/sans accès DMA).
- L'architecture du processeur, que ce soit pour changer d'IP de processeur ou d'inclure dans celui ci des unités de traitements optionnelles.
- Les entrées/sorties vidéo du système, qui dépendent directement de l'application.
- La capacité mémoire locale du processeur pour contenir le programme et les éléments de données à traiter.

Une fois la structure du réseau définie, les composants choisis et paramétrés, le système est complètement défini et peux être généré automatiquement et prendre la forme d'une description unifiée (description en VHDL).

Les dernières étapes correspondent à des étapes traditionnelles d'un cycle de développement de SoPC. L'étape de synthèse matérielle (la synthèse RTL, l'assignation technologique et le placement routage) donne les fichiers de configuration de la cible FPGA pour l'implantation de l'architecture (le *bitstream*).

Le prototype peut ensuite être testé rapidement et validé sur carte. Le concepteur qui est guidé initialement par les résultats de l'étape de parallélisation, peut donc procéder à l'évaluation (en terme de ressources, de temps de traitement ...) en testant différentes alternatives, et affiner les résultats progressivement de manière à retenir l'architecture plus adéquate aux contraintes de l'application.

Bien que faisant partie intégrante de la méthode présentée, ne sont pas mis en avant dans cette partie les aspects itératifs des différentes étapes. Les outils de vérification et d'estimation des performances permettant de raffiner les choix de conceptions seront notamment présentés plus avant dans le chapitre.

Les travaux présentés dans ce mémoire se sont concentrés sur la mise en place d'un environnement de conception de RPHC ciblant des technologies FPGA/SoPC.

Nous présentons donc brièvement les notions relatives à l'étape de parallélisation et les solutions envisagées pour automatiser cette étape pour ensuite mieux mettre l'accent sur les aspects de conception matérielle et notamment d'exploration de l'espace des solutions architecturales.

3.3.1 Etape de parallélisation

3.3.1.1 Problématique

La programmation parallèle est connue pour les difficultés rencontrées lors de sa mise en oeuvre, en comparaison avec la programmation séquentielle. En effet les modèles classiques, maintenant éprouvés, de programmation des architectures séquentielles ne prennent pas en compte de manière adaptée la coordination de tâches réparties, le traitement de plusieurs données simultanément, ou la gestion des transfert de données entre les tâches. On constate aussi la très grande difficulté à penser directement un problème, et donc son algorithme de résolution, sous forme parallèle, c'est-à-dire en identifiant les différentes parties de l'algorithme qui pourraient être effectuées en concurrence. Classiquement, le problème de parallélisation est donc vu comme celui du passage d'une version séquentielle d'un algorithme à une version parallèle.

La parallélisation s'appuie sur l'exploitation des sources de parallélisme de la

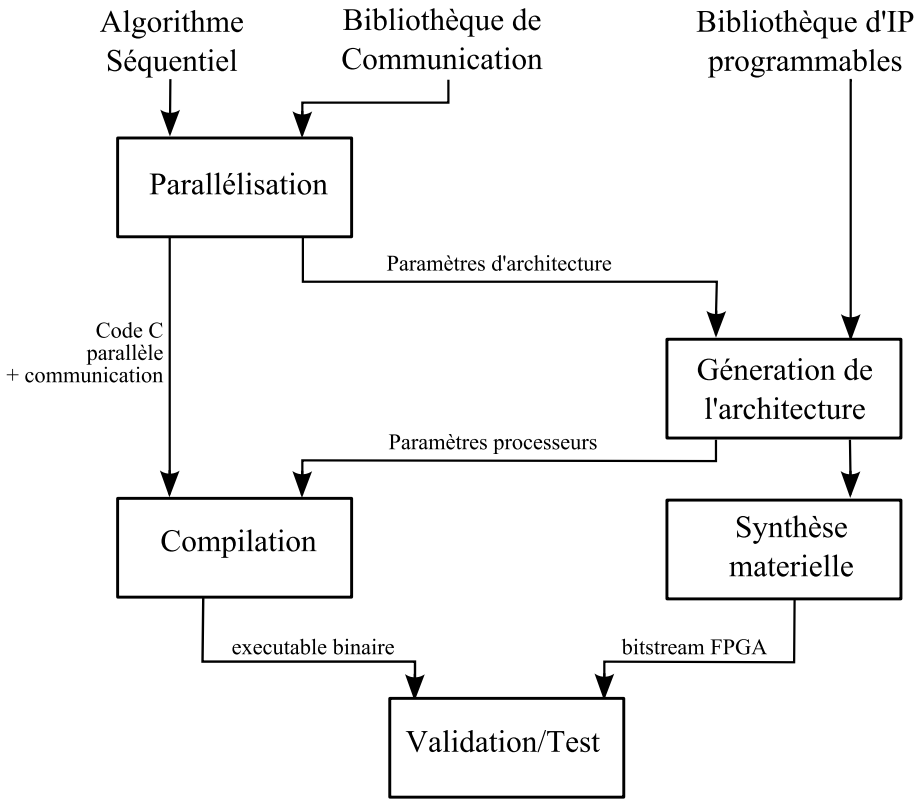


FIGURE 3.2 – Flot de conception simplifié

version séquentielle. Cette mise en oeuvre utilise les formes récurrentes suivantes :

- de parallélisme de données : diviser les données, exécution d'un même programme par N processeurs, fusionner les sous résultats. On distingue le parallélisme de données statique (découpage fixe) du parallélisme de données dynamique, où chaque processeur se voit attribuer une ou plusieurs région(s) d'intérêt dynamique(s).
- de parallélisme de tâches : attribuer à N processeurs des traitements différents,
- de parallélisme de flux : mettre à la chaine un ensemble de traitements sous forme de pipeline.

Une imbrication de ces différentes formes de parallélisme peut être réalisée.

Le passage "à la main" d'une version séquentielle à une version parallèle ne peut être réalisé que par des spécialistes et l'exploration de solutions est fastidieuse.

3.3.1.2 Mise en oeuvre

L'utilisation d'outils d'aide à la parallélisation permet d'une part de fiabiliser et de réduire les temps de conception, mais aussi de rendre la méthode accessible à des personnes spécialisées dans d'autres domaines que la programmation parallèle.

Les outils d'aide à la programmation parallèles se basent principalement sur l'utilisation de langages de programmation parallèles, de bibliothèques de communications, ou de bibliothèques de traitement parallèle.

Différents outils d'aide à la parallélisation ont été développés au sein de notre laboratoire, à partir de la notion de squelettes algorithmiques, permettent le prototypage rapide de différentes solutions parallèles.

L'outil Skipper[72] et la bibliothèque d'aide à la parallélisation QUAFF[39], permettent le développement d'applications parallèles en langage C/C++.

Le concept de squelette algorithmique repose sur le fait que l'on retrouve fréquemment les formes récurrentes que sont le parallélisme de données, de tâches et de flux dans les applications que l'on veux paralléliser.

En pratique un squelette algorithmique est défini comme un schéma parallèle générique, paramétré par une liste de fonctions qu'il est possible d'instancier et de composer. Les fonctions sont spécifiées par l'utilisateur et chaque squelette encapsule les communications qui lui sont propres.

Le concepteur n'a donc plus à gérer explicitement des tâches de communication ou de synchronisation "bas-niveau" et peut donc se concentrer sur des aspects plus algorithmiques de l'implantation.

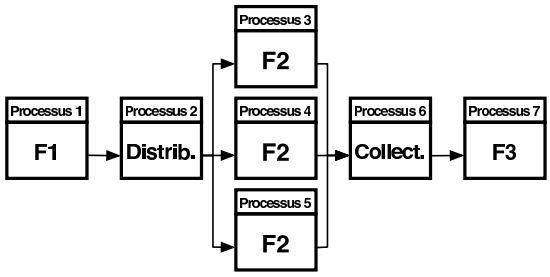


FIGURE 3.3 – Graphe de processus communicants dans une application parallèle : PIPE(SCM)

Chaque type de squelette permet de répondre à une catégorie de problème parallèle, ainsi pour les formes de parallèlismes présentées précédemment il existe les squelettes suivants (la liste n'est pas exhaustive) :

- Le SCM (Split Compute and Merge) qui englobe un schéma de répartition des données, traitement et fusion des résultats correspond à un algorithme régulier comportant un parallélisme de données.

- Le PIPE (pipeline), correspond à un parallélisme de flux et met a profit l'ordonnancement des tâches pour les placer dans un enchainement de processeurs.

- Le FARM (la ferme de processeurs) où un processeur maître distribue les données ou les traitements à des processeurs esclaves et collecte les résultats) permet de traiter des algorithmes irréguliers, ou de mettre en oeuvre un parallélisme de tâches où un parallélisme de données dynamique.

On peut donc, grace à ce formalisme représenter le schéma parallèle d'une application. La figure 3.3 montre par exemple une application parallèle qui peut être exprimée sous la forme de la composition d'un squelette PIPE et d'un squelette SCM.

Les outils Skipper et Quaff génèrent un code parallèle en incluant les fonctions de communications induites par la parallélisation du traitement, suivant le type de squelette spécifié par l'utilisateur. Les fonctions de communications que ces outils intègrent dans le code C sont basées sur la bibliothèque MPI, qui pourront être remplacée par une bibliothèque contenant les fonctions de communications propres à l'architecture.

Il faut noter d'une part que les outils d'aide à la parallélisation utilisés permettent seulement de générer un programme parallèle exécutable sur une machine parallèle de type cluster, comme la machine Babylon[40]. Idéalement, ces outils devraient permettre de générer aussi directement un code parallèle ciblant les architectures embarquées SoPC, ce qui n'est pas le cas actuellement. Une étape supplémentaire de portage de l'application entre le code C parallèle ciblant l'architecture cluster et le code C parallèle ciblant l'architecture embarquée finale est donc nécessaire.

D'autre part le passage de l'étape de parallélisation à l'étape de conception matérielle, qui consiste en l'utilisation des informations obtenues avec le profil de l'application pour guider les différents choix architecturaux, n'est pas facile dans l'état actuel de la situation.

Les outils d'aide à la parallélisation utilisé "tels quels" dans le cadre des travaux présenté dans ce mémoire, sont actuellement repris pour s'adapter l'approche proposée. L'étape de parallélisation sur cluster et les performances obtenue sur se type de plateforme permettent actuellement de déterminer grossièrement un

premier choix de paramètres, qui est ensuite raffiné grâce aux données plus précises obtenues pendant les tests et simulations de l'étape de conception matérielle. La finalité étant, une fois les outils de parallélisation adaptés, de mieux déterminer le premier choix de paramètres ce qui réduira le nombre de raffinements nécessaires, et par un ciblage direct de l'architecture embarquée, éliminera l'étape de portage de code parallèle.

3.3.2 Etape de conception matérielle

3.3.2.1 Problématique générale de l'exploration d'architecture

L'objectif de l'étape d'exploration d'architecture est la recherche d'une architecture de RHPC optimisée pour l'algorithme en vue de son implantation.

Le concepteur est aidé par l'utilisation d'un ensemble d'outils automatisant le flot complet de conception matérielle. L'intérêt est de d'accélérer et de rendre plus fiable l'obtention d'une solution architecturale. Le concepteur pouvant se concentrer sur l'évaluation des performances de son architecture, plutôt que sur la mise en oeuvre de l'architecture et la validation de son fonctionnement.

Les composants IPs (modèles VHDL des processeurs, de leur périphériques, des dispositifs de communications et d'entrée/sortie, etc.) sont préalablement disponibles dans des bibliothèques. Grâce à un outil de génération automatique d'architecture qui assemble ces composants IPs, pour former le système complet, l'étape de conception matérielle consiste essentiellement en une exploration des différents architectures possibles et en leur évaluation.

Un des principaux problèmes de l'exploration d'architecture est celui de l'explosion combinatoire du nombre de solution à évaluer. Ce problème est dû à la quantité d'éléments de choix de conception possibles (ne serait ce qu'en considérant uniquement les paramètres des IPs). Une exploration exhaustive est donc rendue prohibitive par l'étendue de l'espace des solutions architecturales.

Les tâches les plus importantes concernent l'optimisation du coeur de processeur afin de l'adapter aux traitements effectués , et l'optimisation des communications. En fonction du cas de figure, les contraintes temporelles de l'application peuvent nécessiter une plus ou moins large bande passante de communication ou une plus ou moins forte puissance de traitement.

Un autre problème supplémentaire est que chacun de ces choix à une incidence sur l'étendue des autres choix relativement à la cible technologique retenue. Par exemple le choix d'un réseau de communication complexe impliquera un nombre de processeur plus restreint qu'avec une solution de communication basique. Un autre exemple est qu'un nombre important de processeurs réduira la quantité de

mémoire disponible.

Un Réseau de Processeurs Homogène Communicant est une architecture où tous les noeuds sont identiques entre eux et de topologie régulière, ce qui comporte deux avantages par rapport à une architecture hèterogene :

- Il est possible de décrire le réseau complètement, et de générer automatiquement sa structure à partir d'un ensemble réduit de paramètres.
- Cela permet aussi de faciliter l'exploration d'architecture par une réduction considérable du nombre de solutions à évaluer.

3.3.2.2 Mise en oeuvre de l'exploration d'architecture avec notre méthode

3.3.2.2.1 Choix au niveau du processeur

Les choix de conception relatifs à l'architecture du processeur sont nombreux, et ont tous un impact notable sur les performances à l'exécution de l'algorithme et les coûts en ressources. On peux citer par exemple la taille du pipeline, le *datapath*, le type et le nombre d'ALU[1] et de FPU[2]. L'intention est à terme, de pouvoir choisir une architecture de processeur parmi plusieurs IP de type différent : généraliste ou DSP, de famille RISC, CISC ou VLIW.

Dans le cadre des travaux présentés dans ce document, les possibilités de configuration au niveau du processeurs sont limitées à la sélection de paramètres du processeur soft MicroBlaze.

Le MicroBlaze dispose d'unités arithmétiques et logiques optionnelles telles qu'un Barrel Shifter [3], un diviseur entier ou une unité de calcul à virgule flottante. Il est possible aussi d'intégrer une unité de segmentation mémoire (MMU) ou de choisir la profondeur de son pipeline (3 ou 5).

Une description plus détaillée de l'architecture du processeur et de l'impact de ces différents choix sera faite dans le chapitre 4.

Dans la perspective d'étendre les choix au niveau processeur, nous avons aussi testé le processeur OpenFire[25]. Ce processeur est à la base un clone du MicroBlaze développé à l'université Virginia Tech.

Il reprend dans son ensemble l'architecture du MicroBlaze (pipeline 3 étages et architecture mémoire Harvard) et est compatible avec celui-ci au niveau jeu d'instruction. Il constitue une solution de remplacement libre à l'IP propriétaire de

[1] Unité Arithmetique et Logique

[2] Unité de calcul à virgule flottante

[3] Unité logique permettant en une seule fois le décalage ou la rotation binaire dans un registre d'un nombre quelconque de cran.

Xilinx avec des caractéristiques très intéressantes au niveau implantation (équivalentes au Processeur MicroBlaze). Cependant l'absence de compilateur spécifique au processeur OpenFire en contraint l'utilisation comme processeur de "substitution" et ne permet pas d'exploiter des éventuelles modifications architecturales provoquant une variation dans le jeu d'instruction.

L'utilisation de langage de description d'architecture (ADL) décrits dans le chapitre 2, permettant aussi la génération du compilateur C/C++ associé au processeur est une solution en cours de développements en complément aux travaux présentés dans ce mémoire. Ces travaux [21] permettrons à l'avenir d'étendre les possibilités de choix d'IP processeur prévues par cette méthodologie.

3.3.2.2.2 Choix au niveau réseau de communications

La construction du réseau de processeurs complet se résume à l'instanciation du nombre de processeurs souhaité avec leur mémoire associée, et à effectuer les liaisons via l'IP de communication retenue. A chaque noeud est associé une adresse distincte et une stratégie de routage est choisie relativement à cette topologie et cet adressage.

Bien qu'il soit prévu a terme de pouvoir choisir entre différentes topologies, une unique topologie est utilisée dans un premier temps pour l'implémentation. La topologie "hypercube" a été choisie pour ses propriétés en matière d'extensibilité et de routage (cf chapitre 2). Le nombre de liens par noeud ($\log(n)$) et le diamètre D du réseau augmentent raisonnablement avec le nombre de noeuds $n = 2^D$, garantissant une certaine extensibilité en terme de coûts et de performances. En effet, si l'on double le nombre de processeurs, le nombre de liens par noeud et le diamètre n'augmentent que d'un.

Afin de répondre à la problématique de l'exploration d'architecture, nous devons permettre différentes spécifications du réseau de processeurs et assurer que leur implantation soit rapide. Nous avons donc développé plusieurs blocs IP de communication pour que les ressources - en terme de surface occupée par le réseau de communication - puissent être adaptées aux performances exigées par l'application visée.

Nous avons développé différents IPs de manière à permettre le choix entre plusieurs types de liens communications. Les noeuds pourront communiquer dans un réseau direct, avec au choix, des connexions de type point à point ou en utilisant un routeur.

Les connexions point à point peuvent être de deux types. Le plus simple est un lien de type FIFO[4]. Pour diminuer la charge du processeur dans les commu-

[4]De l'anglais, First In First Out, canal de communication organisé en file, ou les données sont

nications, un canal DMA[5] dédié entre les mémoires de noeuds voisins est aussi proposé.

Une solution plus complexe de communication, basée autour d'un routeur de paquets, a aussi été ajoutée à la bibliothèque d'IP de communication. De la même manière, le routeur est proposé avec (ou sans) dispositif d'accès à la mémoire du processeur de manière à permettre au concepteur de faire des compromis relativement au besoin en bande passante, ou à la diversité des profils de communication rencontrés. Ces différents liens et l'architecture du routeur, seront décrits plus amplement avec leur coûts en ressources ainsi qu'une évaluation de leurs performances, dans le chapitre 4.

Chaque type de communication dispose, de plus, d'un ensemble de paramètres qui lui est propre, par exemple la capacité des tampons de communications est une profondeur de FIFO dans le cas d'un lien de communication par FIFO alors que pour les communications DMA, elle s'exprime en terme de capacité de mémoire tampon (en Ko).

3.3.2.3 Environnement de conception

Notre environnement utilise l'environnement commercial Xilinx EDK/ISE et le complète de manière à supporter notre flot de conception. Un flot de conception sur lequel apparaissent les outils de conceptions utilisés avec notre méthode est montré sur la figure 3.4.

3.3.2.3.1 Outils d'implantation

La suite d'outil Xilinx EDK/ISE est dédiée à la conception de SoPC. L'environnement EDK permet la conception matérielle, la programmation, et le test d'un SoPC dans un environnement intégré. Le flot de conception matériel de l'EDK est basé sur l'assemblage de composants IP paramétrables choisis dans une bibliothèque de composants fournie avec l'environnement EDK. Cette base de composants disponibles sous forme de sources VHDL peut être élargie par l'utilisateur avec ses propres composants IP. Les outils de l'environnement ISE permettent les étapes classiques d'implantation FPGA : la synthèse RTL, le placement technologique, le placement routage, et les outils de configuration du circuit. Le flot de conception logiciel est basé autour des très classiques outils GNU (Gcc, Gdb, etc ...) adaptés pour fonctionner avec le processeur Microblaze.

accessibles uniquement dans l'ordre ou elles ont été émises.

[5] De l'anglais Direct Memory Access, dispositif permettant de piloter des accès en mémoire à la place du processeur.

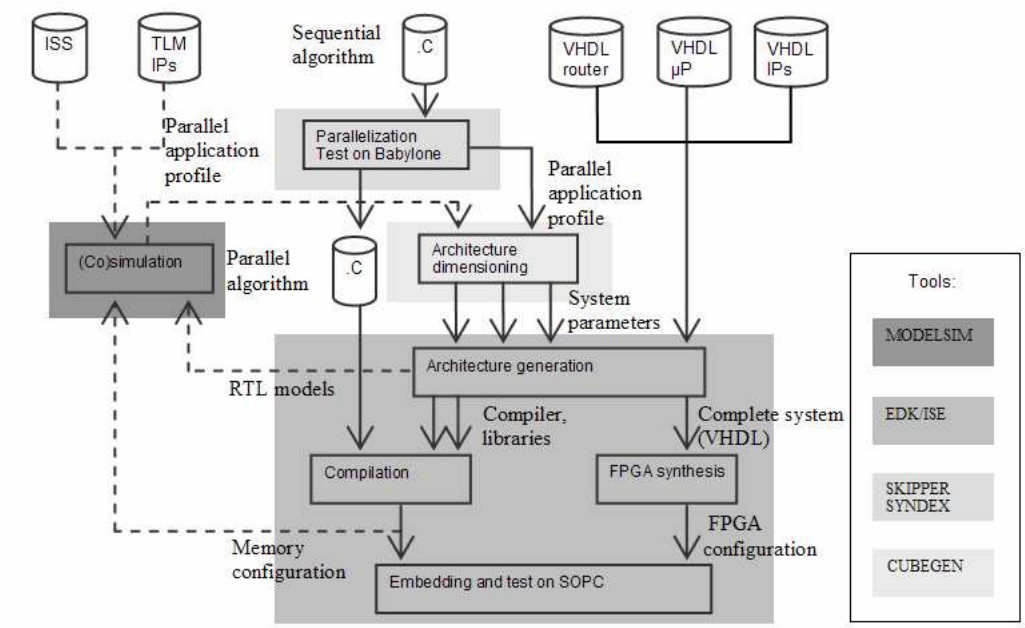


FIGURE 3.4 – Flot de conception automatisé d'un réseau homogène et régulier de processeurs sur SoPC

L'environnement EDK/ISE supporte la mise en oeuvre de système multiprocesseurs. Dans le principal fichier d'entrée (fichier texte ".mhs") se trouvent les appels aux différentes IP retenues, et les informations liées aux connections des composants. L'édition de ce fichier ".mhs" en utilisant directement l'environnement, permet de parvenir relativement facilement à créer un système contenant un petit nombre de processeurs. Cependant l'environnement est inadapté à l'édition d'un système intégrant un nombre élevé de processeurs. Pour les systèmes dépassant 4 processeurs, la description du système devient fastidieuse et source d'erreurs dues (en autres) à l'augmentation du nombre de connexions dans le système.

Nous avons donc crée un outil (CubeGen) automatisant cette étape d'écriture (cf figure 3.5. Il s'agit essentiellement d'un script qui permet d'effectuer la spécification du système à 3 niveaux. Le premier au niveau de l'architecture réseau où sont définis la topologie et les règles d'adressage de chaque noeud. Un second niveau composant où l'on choisi les différentes IP. Puis le niveau micro-architecture, où les paramètres internes des composants sont spécifiés. CubeGen intègre ces 3 niveaux de spécification et génère alors les fichiers d'entrée de l'environnement de développement EDK.

En utilisant l'outils CubeGen, on évite l'écriture manuelle de ce fichier ".mhs".

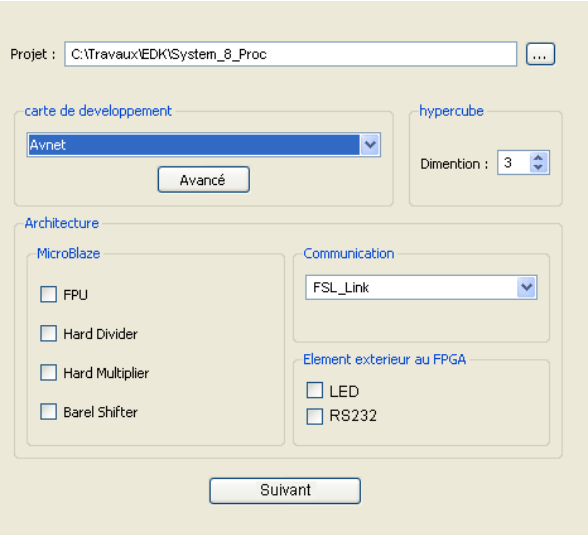


FIGURE 3.5 – Interface graphique de l'outil CubeGen

La conception se résume alors à la spécification de quelques paramètres. Comme aucune étape d'implantation matérielle manuelle n'est nécessaire la fiabilité de la méthode est augmentée.

3.3.2.3.2 Outils de vérification

En plus de la possibilité de test sur carte un cycle de vérification et d'estimation des performances par co-simulation peut être intégrée dans le flot de conception. Nous utilisons l'outil Modelsim, qui opère facilement avec l'environnement EDK. L'utilisation de modèles de simulation RTL, générés par l'environnement EDK, permet d'obtenir des résultats de simulation très précis, mais au prix de temps de simulation particulièrement longs. La simulation sur 100ms d'un système comprenant uniquement 4 processeurs requiers environ 5h sur un PC fonctionnant à 2 GHz. L'utilisation de modèles écrits avec un plus haut niveau d'abstraction est donc nécessaire. Comme les systèmes conçus avec notre approche peuvent comprendre un nombre élevé de processeurs, il est particulièrement important d'utiliser un modèle de processeur performant en terme de temps de simulation. Nous employons un environnement de cosimulation optimisé[57][58], mis en oeuvre par l'université de Alcala de Henares (Espagne). Il est basé sur un simulateur de jeu d'instruction (ISS) modélisant le jeu d'instruction du Microblaze et un modèle fonctionnel de bus (BFM), permettant de coupler l'ISS avec des modèles matériels. Les performances de communications entre l'ISS et le BFM influençant notablement les performances, celles ci ont été optimisées par la mise en oeuvre

de mémoires locales du processeur directement visible par l'ISS. Cet environnement permet la réduction des temps de simulation allant d'un facteur 10 jusqu'à un facteur 1000[58], en fonction de la proportion entre la quantité de modèles TLM et la quantité de modèles RTL utilisés.

3.4 Conclusion

Nous proposons une approche pour l'implantation de traitement d'images sur des systèmes électroniques embarqués. La méthode se base sur une approche en deux étapes basées sur des outils d'aide à la parallélisation, et de génération automatique d'une architecture matérielle, l'implémentation se faisant à partir d'un ensemble de blocs IP paramétrables. Les outils utilisés pour mettre en oeuvre notre approche permettent l'implantation rapide de prototypes d'architectures comportant un nombre élevé de processeurs, l'évaluation par (co)simulation et le test sur cible SoPC Xilinx. L'approche décrite n'est pas un codesign classique, le partitionnement est remplacé par une étape de parallélisation. Ensuite, au prix de restrictions dans les choix architecturaux, elle offre à terme d'être une approche fortement automatisée, ce qui permet des temps de conception réduits et une certaine fiabilité des processus de conception. De plus elle ne nécessite qu'une équipe très réduite et non un ensemble d'experts dans tous les domaines touchés par l'application (traitement d'image, programmation parallèle, conception électronique).

Chapitre 4

Contributions pour l'implantation matérielle de RHPC sur SoPC Xilinx

La section suivante résume l'ensemble des travaux et résultats destinés à prouver la faisabilité de notre approche.

Nous présentons, dans l'optique de l'implantation d'architectures RHPC, des éléments existants, que nous avons adaptés à notre approche.

Nous présentons aussi l'ensemble des contributions dans cette optique, elles concernent essentiellement les aspects de communication. Des résultats d'implantation avec les composants et outils présentés sont donnés.

4.1 Etude préliminaire

4.1.1 Ressources disponibles dans un SOPC

La table 4.1 donne la quantité de ressources disponibles dans des FPGAs Xilinx de grande taille. L'unité utilisée par Xilinx pour les taux d'occupations en terme de surface est le *slice*[1]. Avec les ressources FPGA traditionnelles (registres et LUTs) sont aussi indiquées dans la table, les ressources spécifiques qui peuvent contraindre l'implantation (Blocs DSP[2], et blocs BRAM constituant des mémoires embarquées de 18Kbits).

[1] Dans un virtex 4, un slice contient 2 registres et 2 LUTs. Dans un Virtex 5 un slice contient 1 registre et 1 LUTs

[2] Cellules DSP : Cellules composées de multiplieurs et d'additionneurs, permettant d'intégrer plus efficacement ces traitements qu'avec les classiques LUTs et registres.

TABLE 4.1 – Espace disponible et ressources dans des composants Virtex-4 et Virtex-5 de grande taille.

Cible	Espace SLICE	ressources registres	LUTs	DSP48	BRAM
XC4VLX200	89088	178176	178176	96	336
XC5VLX330	207360	207360	207360	192	288

4.1.2 Estimation des coûts du processeur

4.1.2.1 Architecture du processeur MicroBlaze

Le processeur MicroBlaze dispose d'un certain nombre d'unités arithmétiques et logiques optionnelles qui peuvent avoir un impact significatif sur ses coûts d'implémentation. Il dispose notamment d'une unité de décalage logique (*Barrel shifter*) qui permet d'effectuer des rotations ou des décalages logiques de plusieurs bits sur les registres en une seule opération. Une autre option est celle d'inclure un multiplieur entier plus performant en utilisant les cellules DSP Xilinx pour l'implantation. Un diviseur entier hardware et une unité de calcul à virgule flottante sont aussi optionnels. En l'absence de ces unités, mais si une division ou un traitement à virgule flottante est présent dans le code source C, le compilateur doit insérer un appel de fonction à la place d'une seule instruction.

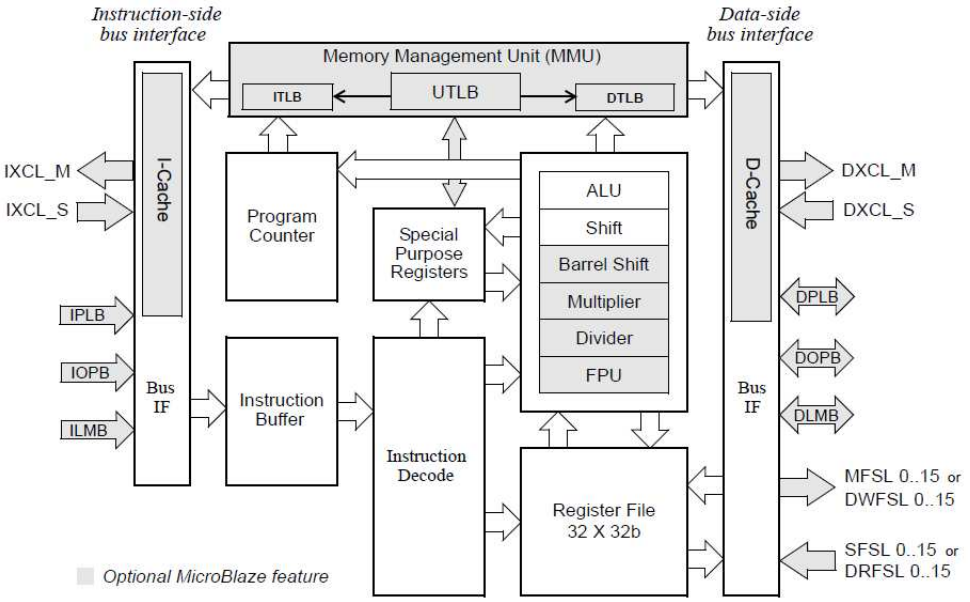


FIGURE 4.1 – Architecture du processeur MicroBlaze

4.1.2.2 Coûts d'implantation du processeur MicroBlaze

Les choix énoncés précédemment ont un impact déterminant sur la quantité de ressources du FPGA utilisées par chaque processeur, et sur l'utilisation de ressources spécifiques (notamment les cellules DSP). L'utilisation de ressources spécifiques amène des contraintes supplémentaires. Premièrement parce que ces ressources sont disponibles en quantité relativement plus limitée que les ressources "classiques" LUT et registres. Deuxièmement parce que ces ressources spécifiques ne sont pas disposées de manière homogène sur la surface du FPGA. Ce qui pourra provoquer des problèmes de placement/routage lors de l'implantation.

TABLE 4.2 – Impact de quelques unités de traitement optionnelles sur la surface occupée et l'utilisation de ressources FPGA pour l'implantation d'un processeur MicroBlaze.

Unités de traitement				Ressources requises			
Décalage logique	Diviseur entier	Multiplieur	Virgule Flottante	slices	registres	LUTs	DSP
0	0	0	0	936	559	1146	0
1	0	0	0	1071	593	1404	0
0	1	0	0	998	669	1269	0
1	1	0	0	1134	706	1536	0
0	0	1	0	927	578	1115	3
0	0	0	1	1578	1144	2220	4
0	1	1	0	1008	687	1266	3
1	1	1	0	1145	721	1527	3
1	1	1	1	1777	1297	2582	7

Nous avons synthétisé différentes configurations de processeurs afin d'évaluer l'impact des choix d'implantation du processeur MicroBlaze sur son coût en ressource et sa surface occupée. Les résultats présentés dans le tableau 4.2.

Sur ce tableau on constate aussi que l'implantation de l'unité de multiplication entière en utilisant des cellules DSP permet de réduire le nombre de LUT consommées par le processeur. Les configurations de processeur avec une unité de traitement à virgule flottantes sont les plus volumineuses, nécessitant plus de 1578 slices pour chaque processeurs. Une FPU nécessite de plus 4 cellules DSP.

Pour un XC4LX200, si des blocs DSP48 sont utilisés pour l'implantation, nous avons constaté que spécifier plus de 8 processeurs pouvait amener à un échec des outils du placement/routage.

Les processeurs dotés d'une unité de décalage optionnelle ou d'une unité de division entière sont plus volumineuses (+14,4% pour l'unité de décalage logique, +6,6% le diviseur, +21% pour les deux) mais ne consomment pas de blocs DSP.

En comparant les tables 4.1 et 4.2, on peut constater que la plupart des configurations occupent moins de 1% d'un FPGA de grande taille.

Il est ainsi envisageable d'implanter une centaine de processeur si les choix d'implantation utilisent uniquement les ressources les plus abondantes (registres et LUTS).

4.1.3 Estimation des ressources mémoires

Implanter une application dont le coût mémoire est inférieur à la quantité de mémoire disponible à l'intérieur du FPGA fait partie des contraintes qui peuvent aussi limiter le nombre maximum de processeur que l'on pourra implanter sur cible donnée.

Nous nous plaçons dans le cas où l'intégralité de l'application est placée dans la mémoire embarquée (pas d'accés à la mémoire externe au FPGA). Cette mémoire est repartie équitablement entre les différents noeuds du RHPC. L'estimation de l'évolution des ressources mémoire en fonction du nombre de processeur est donc une fonction de la forme suivante :

$$Q_{noeud} = \frac{Q_{Tot}}{Nb_{noeud}} \tag{4.1}$$

Q_{Tot} étant la quantité de mémoire embarquée de la cible (en octet), Q_{noeud} étant la quantité de mémoire disponible pour chaque noeud (en octet) et Nb_{noeud} le nombre de noeud du RHPC.

Dans le cas d'un FPGA Virtex-4 LX200, contenant 336 blocs de 4Ko, implanter 64 noeuds signifie que chaque noeuds ne dispose que d'au maximum 16Ko de mémoire locale (4 blocs).

Le croisement de la fonction de contrainte et celle de coût en mémoire de l'application, illustrées sur la figure 4.2, indique le nombre maximal de processeurs.

Le coût en mémoire de l'application montré sur la figure est donné juste de manière indicative, car celui si dépends complètement du cas particulier de l'application, et de son implantation.

La quantité de mémoire T_{noeud} nécessaire à chaque processeur pour exécuter l'application comprends deux composantes. Une première composante correspond à la place occupée par les instructions que le processeur doit exécuter. Une deuxième composante correspond aux données manipulées par ce programme.

La figure illustre l'exemple d'une application de traitement d'image qui comporte un parallélisme de données. Son implantation parallèle est faite avec un découpage statique des données images et un mode d'exécution SPMD[3].

Avec un mode SPMD, chaque processeur exécute le même programme, la place occupée par celui ci que nous appellerons T_{prg} est donc constante. Dans un cas idéal, on peux repartir la totalité des données à traiter (T_{Data}) entre les noeuds sans avoir à dupliquer les données. La mémoire nécessaire à l'exécution de l'application est de la forme :

$$T_{noeud} = \frac{T_{Data}}{Nb_{noeud}} + T_{prg} \tag{4.2}$$

On peut constater que même dans un cas où l'on peut répartir avec efficacité parfaite les données à traiter entre processeurs, il resterait la composante liée aux programme T_{prg}.

Il est nécessaire d'implanter l'application de manière optimisée en terme de consommation de ressources mémoire. Pour cela, nous effectuons une étape de raffinement de l'application parallèle pour l'optimiser en fonction de ce critère.

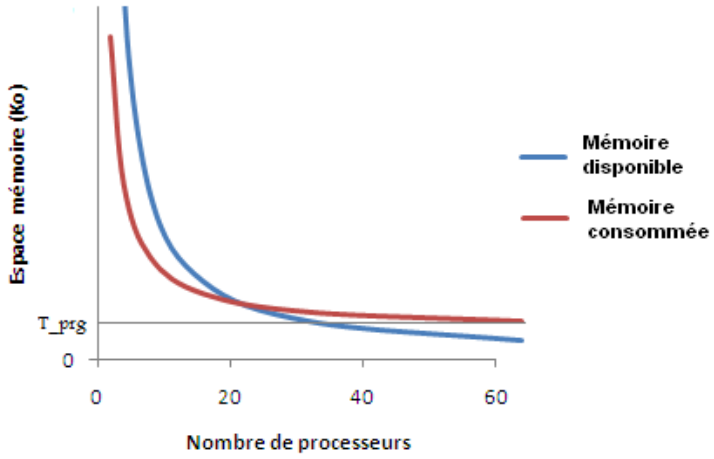


FIGURE 4.2 – Evolution de la mémoire disponible par noeud et de la consommation mémoire par noeud en fonction du nombre de noeuds.

[3]Simple Programme Multiple Données : Mode d'exécution d'une machine MIMD où chaque processeur exécute le même programme.

4.2 Description des composants matériels

Comme il l'a été énoncé dans le chapitre 3, nous avons mis en oeuvre divers modes de communications. En fonction des besoins, il est possible d'utiliser (ou de ne pas utiliser) un routeur. Dans les deux cas, il est possible de doter (ou non) le lien de communication d'un accès direct à la mémoire locale (DMA), ce qui donne quatre liens de communication que nous nommerons de la façon suivante :

- Lien de communication point à point par canal FIFO.
- Lien de communication point à point par canal DMA.
- Lien de communication avec un routeur.
- Lien de communication avec un routeur et DMA.

4.2.1 Communication point à point par canal FIFO

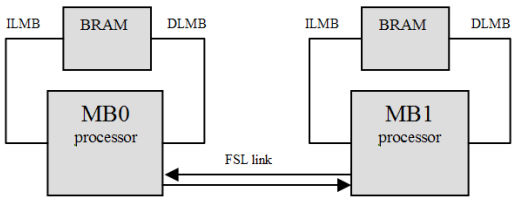


FIGURE 4.3 – Connection point à point FIFO entre 2 processeurs.

4.2.1.1 Description

Premièrement, nous avons utilisé le lien FSL (Fast Simplex Link) du processeur MicroBlaze (cf figure 4.3), qui permet au processeur d'émettre ou de recevoir des données avec un processeur ou périphérique voisin via une connection de type FIFO.

Le MicroBlaze dispose de 16 ports d'entrées et 16 ports de sortie[4], permettant de mettre en oeuvre des liaisons full-duplex entre 16 noeuds voisins.

L'intérêt de disposer d'un tel lien de communication est de pouvoir privilégier une solution économique en ressources matérielle. Cependant l'intégralité de la

[4]Il dispose de 8 canaux maximums dans les versions anterieures à la version 6.

gestion des communications est logicielle. Le processeur est responsable notamment de tous les déplacements de données entre les canaux FIFO et la mémoire. Aucun recouvrement entre temps de traitement et temps de communication n'est donc possible.

4.2.2 Communication point à point par canal DMA

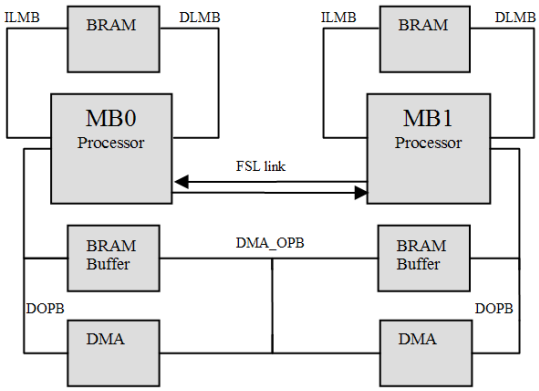


FIGURE 4.4 – Connection point à point avec DMA entre 2 processeurs.

4.2.2.1 Description

Un deuxième type de lien de communication point à point est mis à disposition, avec des canaux DMA cf figure 4.4. Chaque canal est constitué de mémoires tampons d'émission et de réception, et de contrôleurs DMA permettant de piloter les échanges entre ces deux mémoires. Deux canaux DMA distincts ne partagent aucune ressource. Ce type de lien permet de libérer le processeur des échanges des données entre canal de communication et mémoire, cependant restent à sa charge diverses opérations de gestion des communications, avec par exemple le routage, et toutes les opérations de contrôle de communications.

Le lien de communication DMA est constitué de deux liaisons de nature distincte, la première étant destinée aux transferts de données, et la seconde à l'échange d'informations de contrôle. Le transfert en DMA proprement dit est effectué par

un bus OPB connectant 2 mémoires, et 2 contrôleurs de DMA[5]. Pour les échanges d'information de contrôle entre noeuds adjacents, un canal FIFO du même type que celui décrit précédemment à été utilisé.

4.2.3 Communications réseau avec un routeur

L'utilisation d'un NoC permet de libérer les processeurs de la prise en charge des communications pendant toute la traversée du réseau. Plus précisément, le routage, et le contrôle de flux, ne sont plus ici effectués en logiciel, mais sont effectués par le routeur.

Sont proposés là aussi deux modes de communication de type NoC : une solution employant simplement le routeur directement interfacé avec le processeur et une solution employant le routeur et une interface DMA entre le routeur le processeur, et sa mémoire locale. Les deux solutions sont illustrées dans la figure 4.5 pour des réseaux de 4 noeuds.

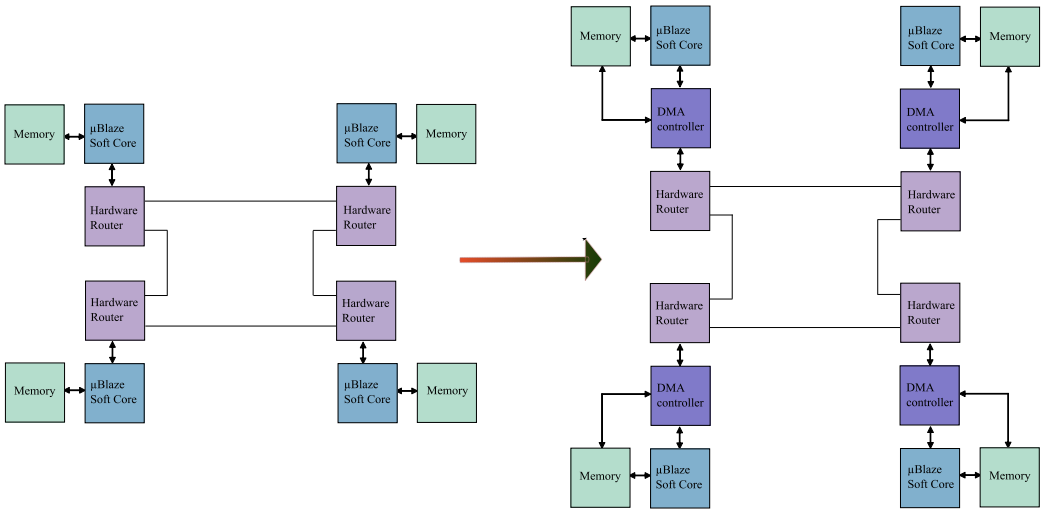


FIGURE 4.5 – Réseau de 4 processeur avec routeur, sans interface DMA (à gauche) et avec interface DMA (à droite).

4.2.3.1 Description du routeur

Le routeur que nous présentons ici a été conçu pour mettre en oeuvre des NoC à topologie Hypercube. Son nombre de canaux est $D+1$, il correspond donc à la

[5]Ces composants sont disponibles dans les librairies D'IP de l'environnement EDK de Xilinx.

dimension D de l'hypercube correspondant, plus un lien en direction du noeud de traitement local.

Chaque noeud est numéroté de manière à ce que la distance de Hamming entre deux noeuds soit égale à la distance qui les sépare et que chaque bit corresponde au canal traversé. Ainsi, une simple fonction logique (OU exclusif) entre la destination du paquet et sa position actuelle dans le réseau permet de déterminer dans quelle direction faire suivre le paquet. Le paquet circule ainsi de proche en proche jusqu'à ce qu'il atteigne sa destination finale. Ce mode de calcul assure un routage déterministe [6] et minimal.

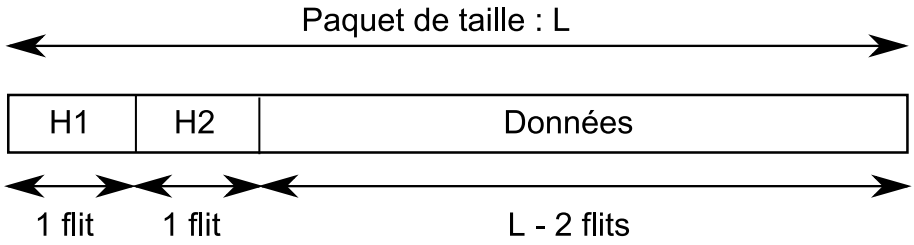


FIGURE 4.6 – Format de paquet manipulé par le routeur.

Pour sélectionner les chemins empruntés par les paquets, les différents canaux sont traversés strictement en ordre croissant. L'algorithme de routage utilisé[28] est exempt de deadlock, les croisements de trajectoires empruntées par les paquets ne formant pas de cycles fermés.

La technique de contrôle de flux est basée sur un protocole *handshake*[7] : un routeur émet une requête dès qu'il peut réémettre un flit[8] et accepte une requête dès qu'il a suffisamment de ressources tampon pour l'accueillir.

Le routeur manipule des paquets au format suivant illustré sur la figure 4.6 : La longueur L du paquet (en nombre de flits) est paramétrable. Le paquet est constitué de deux flits d'entête H1 et H2 et d'un nombre variable (L-2) de flits de données. Le format des entêtes H1 et H2 sont illustrés sur les figure 4.7 et 4.8.

La technique de routage utilisée est de type wormhole, le routeur disposant de registres tampons permettant de conserver une partie des paquets en transit sous la forme de quelques flits, et renvoie les flits reçus dès que possible. Le choix de cette technique de routage permet d'une part de minimiser l'accroissement des temps de latence avec la distance de traversée des données dans le réseau. Cela

[6] Deux paquets suivent un chemin identique pour un couple émetteur/récepteur donné.

[7] Handshake : Protocole requête-acquitement

[8] Un flit est l'unité élémentaire de contrôle de flux. Le terme provient de la contraction de l'anglais *flow control unit.*

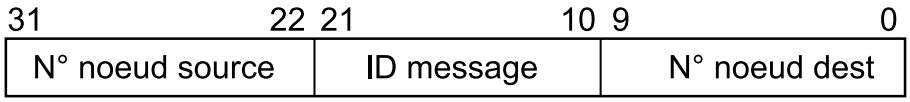


FIGURE 4.7 – Format du premier flit d'entête.

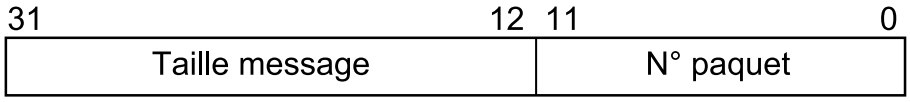


FIGURE 4.8 – Format du second flit d'entête.

permet d'autre part de définir un routeur nécessitant moins de ressources tampon. Comme un même paquet transite dans plusieurs routeurs à la fois, cette technique à l'inconvénient d'être sensible à des congestions en chaîne si les paquets ne sont pas retirés suffisamment rapidement du réseau par le destinataire final. De manière à réduire les effets de congestions résultants de ces blocages en fin de ligne le routeur dispose donc de deux canaux virtuels.

4.2.3.2 Architecture du routeur

Le canal associé au processeur est fixé à 32 bits, tandis que les canaux entre les routeurs sont de 33 bits, 32 bits contenant les données du flit et le bit supplémentaire identifie le canal virtuel emprunté.

L'architecture interne du routeur est visible sur la figure 4.9. Il comprends 5 blocs, un module de contrôle central, une matrice de commutation et un module d'émission et de réception, et un module d'arbitrage/routage :

- Les blocs de transmission et d'émission sont constitués de FIFOs qui permettent aux flits d'être stockés temporairement. Dans ces modules est géré la politique de contrôle de flux en entrée et sortie, relativement à l'état des FIFOs.

- Le bloc d'arbitrage/routage décode les entêtes des nouveaux paquets en réception, et attribue un canal de sortie à un canal d'entrée relativement à la politique de routage.

- Le bloc de contrôle central pilote le crossbar et les flux de données internes au routeur. Une machine à état fini est associée à chaque canal. Son rôle est de contrôler le transit d'un paquet à travers le crossbar, flit après flit, jusqu'à ce que le dernier flit ait été ré-émis.

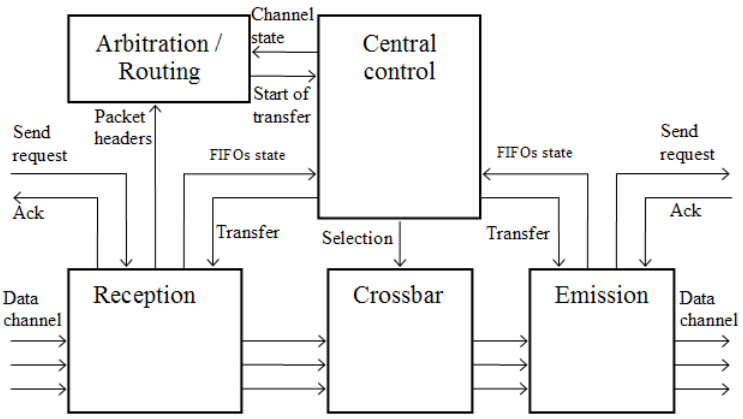


FIGURE 4.9 – Blocs fonctionnels du routeur.

TABLE 4.3 – Resources SoPC utilisées par le routeur en terme de surface, de registres et de fonctions logiques.

cible	SLICE	FF	LUTs
Virtex4L200	1688 (2%)	1964 (1,1%)	1777 (1%)
Virtex5LX330	2613 (1,3%)	2071 (1%)	1073 (0,5%)

4.2.3.3 Coûts d'implantation du routeur

Sur la table 4.3 sont donnés les coûts en ressources pour un routeur avec des canaux capables de stocker 8 flits. Cette configuration de routeur occupe 2% de la surface d'un Virtex4 ou 1,3% d'un Virtex5 de grande taille. Le gain en surface ocupée sur Virtex5 provient d'une meilleure utilisation des LUTs.

4.2.3.4 Performance d'un routeur isolé

Nous voulons ici montrer les performances propres du routeur, indépendamment de l'effet que pourraient avoir d'autres éléments sur les performances de communication du système. Les mesures sont donc effectuées dans cette optique, en enlevant tout facteur externe au routeur, qui pourrait en limiter les performances.

Premièrement, nous nous situons ici dans le cas d'une utilisation optimale de la bande passante des canaux du routeur. C'est le cas quand les données sont présentées en entrée du routeur aussi vite que le routeur peut les recevoir. Deuxièmement

nous nous situons ici en l'absence de congestion, c'est à dire que les données sont acceptées au niveau des sorties du routeur aussitôt que le routeur les présente.

Nous avons simulé, par l'intermédiaire d'un benchmark, dans ce contexte de routeur isolé d'un système, l'envoi direct d'un paquet à travers le routeur pour en mesurer le débit et la latence.

Les paquets envoyés sont de longueur $L = x + 2$ flits (x flits de données et 2 flits d'entête). L'envoi est fait à la cadence de 1 flit par coup d'horloge, directement dans l'un des ports d'entrée du routeur et directement récupéré sur un port de sortie.

Latence :

$$\boldsymbol{r} = 4clk$$

Le routeur a une latence minimum de 4 coup d'horloge, dû à la traversée successive des différents étages synchrones le composant.

Bande Passante :

$$Bw = \frac{4 \times F \times (L-2)}{L+4} Octet.s^{-1} \tag{4.3}$$

En prenant en compte une latence de 4 coups d'horloge, un paquet de (L-2) données sera ré-émis en $L+4$ coups d'horloge.

Par conséquent un paquet de 16 flits sera ré-émis avec une bande passante de $2,8 \times F$ Mo.s^{-1}, c'est à dire 420 Mb.s-1 à 150 MHz. En augmentant la taille des paquets, la bande passante théorique augmenterait à une limite de $4 \times F$ Mo.s^{-1} ; Néanmoins il faut noter qu'augmenter la taille des paquets peut en même temps augmenter la probabilité d'apparition de congestions.

4.2.4 Communications réseau avec routeur et interface réseau DMA

L'interface entre noeud et routeur est une simple interface FIFO 32bits. Via cette interface, le routeur peut être directement connecté avec le processeur sur un de ses port FSL. Une deuxième possibilité que nous décrivons ici est de placer un interface DMA entre le processeur, le routeur et la mémoire.

4.2.4.1 Description de l'interface réseau DMA-routeur

Nous verrons par la suite que si l'on interface directement le processeur au routeur, les performances de communications sont limitées par l'efficacité du processeur.

Les fonctions implémentées en matériel sont :

- coté émission : la segmentation du message à émettre en paquets.
- coté réception : la reconstitution du message à partir des paquets, et bien sur les accès en lecture et en écriture à la mémoire du noeud.

4.2.4.2 Architecture de l'interface réseau DMA-routeur

L'architecture interne de l'interface réseau DMA-routeur est visible sur la figure 4.10. Le DMA est composé de quatre blocs fonctionnels distincts, un interface microprocesseur, les registres de configuration, un blocs d'émission et un bloc de réception :

- L'interface microprocesseur est une machine à état qui décode les commandes provenant du processeur. Il est connecté au processeur via un port FSL. En fonction des commandes reçues, elle lit ou écrit dans les registres de configuration, et répond éventuellement au microprocesseur.
- Les registres de configuration servent d'interface entre les trois autres blocs. Ils contiennent l'intégralité des informations de configuration des communications et l'état des communications en cour. Jusqu'à 4 émissions de message et 4 réceptions peuvent être configurées dans les registres.
- Le bloc émission est essentiellement une machine à état pilotée par le bloc de configuration. Il est connecté d'un coté à un tampon d'émission qui fait partie de la mémoire locale et de l'autre coté à une entrée du routeur. Lors d'une émission, ce bloc calcule l'adresse mémoire des données, les lit et injecte les paquets flit par flit dans le routeur en insérant l'entête.
- Le bloc réception est lui aussi essentiellement une machine à état pilotée par le bloc de configuration. Il est connecté d'un coté à un tampon mémoire de réception et de l'autre coté à une sortie du routeur. Lors d'une réception, ce bloc retire les paquets du réseau, en récupérant les données flit par flit et en enlevant l'entête de message, et les écrit, progressivement en mémoire.

4.2.4.3 Coûts d'implantation

Sur la table 4.4 sont donnés les coûts en ressources de cet interface DMA. L'interface réseau DMA-routeur occupe $1,2\%$ de la surface d'un Virtex4 ou $0,6\%$ d'un Virtex5 de grande taille. Le gain en surface sur Virtex5 montre aussi une moindre utilisation en ressource LUT que sur Virtex4.

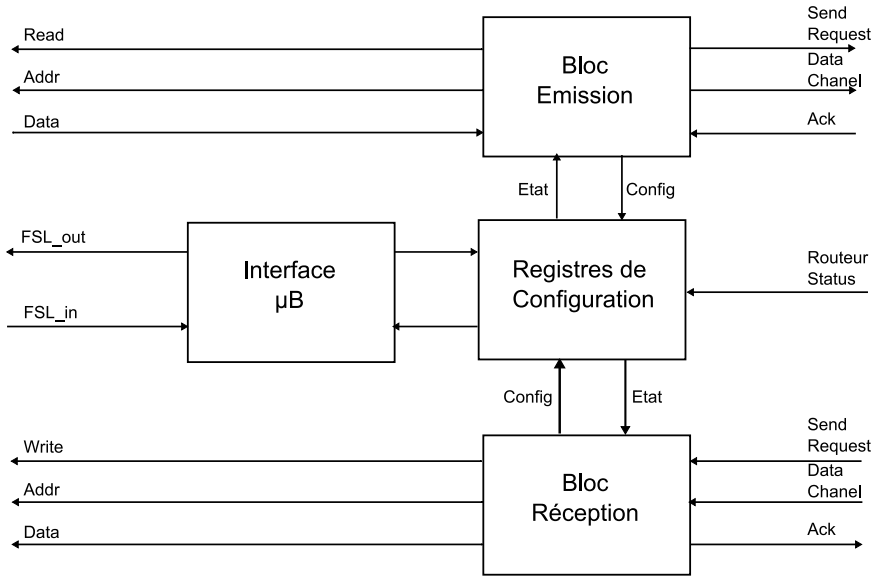


FIGURE 4.10 – Blocs fonctionnels de l'interface réseau DMA.

TABLE 4.4 – Ressources SoPC utilisées par un interface DMA en terme de surface, de registres et de fonctions logiques.

cible	SLICE	FF	LUTs
Virtex4LX200	1088 (1,2%)	771 (0,4%)	1482 (0,8%)
Virtex5LX330	1355 (0,6%)	837 (0,4%)	1147 (0,6%)

4.2.4.4 Performances

Le DMA étant en cours de validation, les performances données ici sont issues de résultats de simulation.

Latence :

$$\boldsymbol{r} = 25clk$$

La latence d'initialisation de la communication, est essentiellement dûe au temps de programmation de l'interface réseau-DMA par le processeur, elle est d'environ 25 coups d'horloge processeur.

A partir du moment où le contrôleur DMA à commencé à effectuer un transfert, les performances sont indépendantes du processeur.

L'interface DMA-routeur met (L+1) cycles d'horloges à injecter ou récupérer dans le réseau un paquet de taille L contenant 2 flits d'entête et (L-2) flits de données. Il faut ajouter encore 4 cycles d'horloges dûs à la latence du routeur. Ce

temps de 25 cycles cycles d'horloges par paquet est ensuite répété jusqu'à ce que le message soit complètement transmis.

Bande Passante :

$$Bw = \frac{4 \times F \times (L-2)}{L+5} Octet.s^{-1} \tag{4.4}$$

En considérant l'équation 4.4, si on prends un paquet de longueur 16, la bande passante sera de $2.6 \times F$ O.s^{-1} c'est à dire 400 MO.s^{-1} à une cadence de 150 MHz.

On voit ici que l'interface DMA-routeur permettra une utilisation efficace des canaux du routeur. Cependant son insertion augmentera sensiblement le coût en ressources du réseau de communication.

4.2.5 Composants d'entrée/sortie vidéo dédiés

Comme il a été défini dans le chapitre 3, une architecture RHPC dispose d'un mode d'accès uniforme aux données vidéo.

Il est important de permettre à chaque noeud d'accéder directement aux données images, par l'intermédiaire d'un dispositif d'entrée vidéo dédié. Une deuxième fonctionnalité importante de ce dispositif est de pouvoir sélectionner uniquement les données vidéo nécessaires aux traitements en s'adaptant aux modes de répartition de données relatifs à l'application parallèle de TI (découpage en bandes horizontales ou verticales, technique de fenêtrage en régions d'intérêt, etc...).

Enfin, il a été montré dans les travaux de Yoshimoto et Arita[81] et ceux de J.Falcou[40] que disposer d'un mécanisme de distribution des données vidéo matériel, *en plus* d'un réseau classique, est bien plus efficace en terme de latence d'accès aux données images.

4.2.5.1 Description

Nous proposons deux approches architecturales pour implanter un interface entre le flux vidéo et le réseau de processeur illustrées dans la figure 4.11.

Dans les deux cas le flux vidéo entrant est un flux de données vidéo numérique standard[9] produit par la chaine d'acquisition.

Les deux approches sont différentes au niveau du placement de la mémoire tampon vidéo. La première approche (coté gauche de la figure) utilise une mémoire tampon vidéo externe au SoPC. La deuxième approche (coté droit de la figure) utilise les mémoires internes aux SoPC comme ressources tampon video.

[9]Le standard de vidéo numérique entrant choisi est le YCbCr 4 :2 :2, décrit dans les recommandations ITU-R CCIR 601/656.

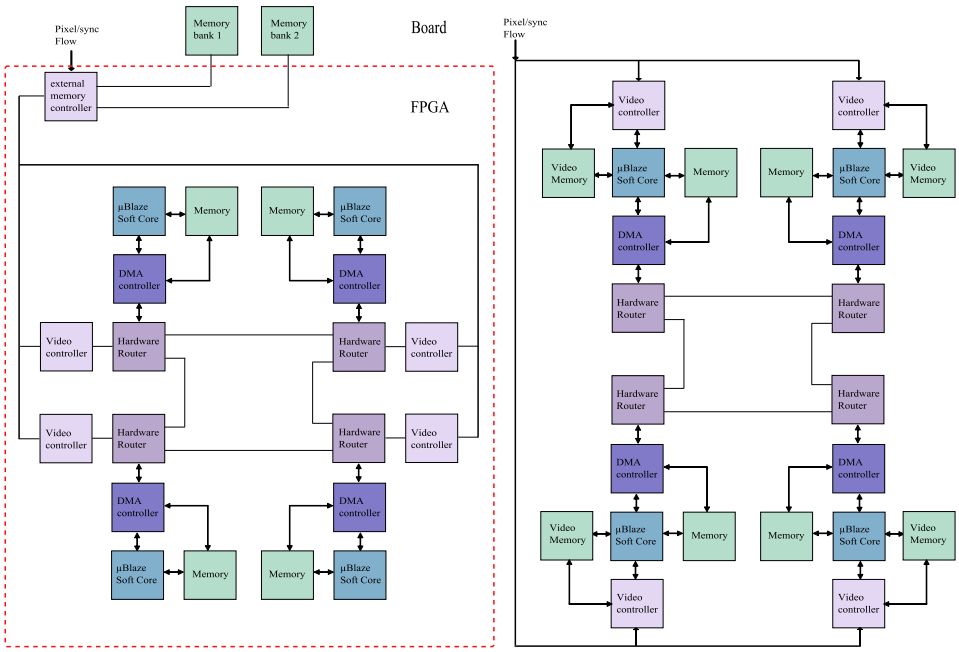


FIGURE 4.11 – Deux approches d'interface d'Entrée/Sortie vidéo dédiée, illustrées avec des RHPC de 4 noeuds.

La première approche consiste à réaliser l'architecture d'entrée vidéo sous forme de deux blocs : un contrôleur de mémoire externe qui stockera temporairement une image dans un premier banc d'une mémoire externe dédiée, puis à l'arrivée d'une nouvelle image, ce contrôleur vidéo stockera la nouvelle image dans un deuxième banc mémoire. L'autre bloc est une interface vidéo-routeur qui transforme les données vidéo brutes provenant du contrôleur de mémoire externe en une succession de paquets au format réseau que nous avons décrit en section 4.2.3.1 de ce chapitre. Cette approche présente l'avantage de mémoriser l'intégralité de l'image en mémoire externe. Il n'est donc pas nécessaire de déterminer à l'avance les données images nécessaires au traitement. Cette architecture pourra par contre être limitée en terme de performance, car les accès à cette mémoire externe peuvent constituer un goulot d'étranglement. Un autre inconvénient de cette approche est qu'elle implique obligatoirement l'utilisation de routeur.

La deuxième approche (coté droit de la figure) est de coupler un contrôleur vidéo à une mémoire tampon vidéo interne au SoPC. Le processeur programme dans le contrôleur vidéo les zones de l'image qui l'intéresse, qui seront directement écrites dans la mémoire tampon vidéo. Cette mémoire tampon étant double port, chaque processeur peut lire les données image de manière transparente sur son port. Cette deuxième approche présente l'avantage d'être indépendante de la

nature du lien de communication choisi. En contrepartie elle présente l'inconvénient d'utiliser des blocs de mémoire internes au FPGA qui sont une ressource très limitée. Il ne sera donc envisageable que de stocker des zones restreintes de l'image. Cette deuxième approche à été implantée et est en cours de validation.

4.3 Programmation des communications

4.3.1 Communications point-à-points FIFO

Les opérations de lecture ou d'écriture sur les ports FSL du MicroBlaze sont directement intégrés avec son jeu d'instruction. Pour programmer ces opérations en langage C, Xilinx a mis à disposition des macros C correspondant à ces opérations.

On peut voir dans la figure 4.12 les macros C et les instructions assembleur correspondantes. Les accès aux canaux FIFO sont déclinés en version bloquante et non bloquante[10] ainsi que d'un mode contrôle/donnée en 8 opérations de communication.

```
// Lecture et Ecriture blocantes de données
#define microblaze_bread_datafsl(val, id)     asm volatile ("get %0, rfsl" #id : "=d" (val))

#define microblaze_bwrite_datafsl(val, id)    asm volatile ("put %0, rfsl" #id :: "d" (val))

// Lecture et Ecriture non-blocantes de données
#define microblaze_nbread_datafsl(val, id)    asm volatile ("nget %0, rfsl" #id : "=d" (val))

#define microblaze_nbwrite_datafsl(val, id)   asm volatile ("nput %0, rfsl" #id :: "d" (val))

// Lecture et Ecriture blocantes d'un mot de contrôle
#define microblaze_bread_cntlfsl(val, id)     asm volatile ("cget %0, rfsl" #id : "=d" (val))

#define microblaze_bwrite_cntlfsl(val, id)    asm volatile ("cput %0, rfsl" #id :: "d" (val))

// Lecture et Ecriture non-blocantes d'un mot de contrôle
#define microblaze_nbread_cntlfsl(val, id)    asm volatile ("ncget %0, rfsl" #id : "=d" (val))

#define microblaze_nbwrite_cntlfsl(val, id)   asm volatile ("ncput %0, rfsl" #id :: "d" (val))
```

FIGURE 4.12 – Fonctions élémentaires d'accès au ports d'entrée/sortie FSL.

Le résultat d'une de ces opérations est l'envoi (la réception) d'un mot de 32 bits sur le canal désigné à partir de (ou vers) un registre du processeur. Comme on le voit ici, la gestion des communication est intégralement à la charge du processeur, qui doit de plus aller chercher ou placer chaque donnée en mémoire. Avec ce

[10] Le processeur passera, ou ne passera pas à la prochaine instruction en cas d'écriture dans une FIFO pleine ou de lecture d'une FIFO vide.

mode de communication, les performances du processeur, et le degré d'optimisation des fonctions de gestions de communication auront un impact majeur sur les performances (cf section 4.4.1.2).

4.3.2 Communications point-à-points DMA

4.3.2.1 Gestion de la communication

La première étape est une étape d'initialisation de la communication. Le noeud récepteur et le noeud émetteur se synchronisent, et échangent les données de contrôle pour établir le communication, en utilisant le canal dédié à cet effet.

Outre les notifications de début et de fin de transfert et les synchronisations, les informations de contrôle comprennent entre autre l'identification de la communication et la quantité de données totale de la communication (qui peut impliquer plusieurs transferts DMA en fonction des tailles de tampon disponibles).

Ces opérations de contrôles étant effectuées en logiciel, elles introduiront plus ou moins de latence entre les étapes de transfert brut en DMA, cette latence étant fonction des performances du processeur et du degré d'optimisation des fonctions de gestion de communication.

4.3.2.2 Transfert de données brut entre deux mémoires distantes

Dans un premier temps, le processeur programme le transfert en écrivant dans les registres de configuration du contrôleur DMA (adresse de la source, adresse de la destination, quantité de données). Dans un second temps, le transfert DMA proprement dit est effectué sur le canal de communication entre la mémoire appartenant au noeud source et la mémoire appartenant au noeud destinataire, sans nécessiter d'intervention du processeur. Selon la configuration, le contrôleur génère (ou non) une interruption pour signifier la fin du transfert. Le processeur qui a initié le transfert peut à tout moment vérifier l'état du contrôleur en accédant directement aux registres du contrôleur.

4.3.3 Communication réseau avec le routeur sans DMA

L'utilisation d'un routeur permet d'éviter au processeur toute la gestion de la circulation des données tant que celles-ci sont dans le réseau. Cependant, en l'absence d'interface DMA, l'injection et le retrait de paquets du réseau ainsi que tous les accès mémoire sont complètement à sa charge.

L'injection ou le retrait de paquet sont faits flit par flit par des écritures ou des lectures sur le port FSL qui connecte le processeur au routeur. Pour une injection

de paquet, le processeur est également chargé d'aller chercher les données à communiquer à partir de sa mémoire locale. Egalement, pendant qu'il retire un paquet du réseau, le processeur doit écrire les données en mémoire.

L'envoi d'un message se fait en répétant l'injection de paquets jusqu'à ce que l'intégralité des données aient été envoyées. Réciproquement, la réception d'un message se fait par la répétition du retrait de paquet jusqu'à ce que l'intégralité des données du message ait été écrite en mémoire dans le noeud récepteur.

4.3.4 Communication réseau avec le routeur avec DMA

La segmentation du message en paquet et les accès à la mémoire du noeud sont cette fois effectués en matériel. Le processeur initialise alors la communication réseau en accédant aux registres de l'interface DMA du routeur. Le processeur est là aussi relié avec l'interface DMA par un port FSL.

Ces accès au port FSL sont effectués avec les fonctions d'entrée/sortie bas-niveau fournies par Xilinx que nous avons aussi employées pour le mode de communication point à point FIFO.

4.3.5 Vers une programmation homogène des communications

Chacun des liens de communication décrit dans la section 4.2 se programme, à bas niveau, de manière distincte. Les différences notables au niveau de l'écriture des communications, obligent le concepteur à adapter le programme en fonction du mode de communication.

Cette adaptation au mode de communication peut se révéler plus ou moins complexe, en fonction du modèle de communication employé.

Les deux liens de communications point-à-point FIFOs et point-à-point DMA ne permettent, à bas niveau, que des communications entre noeuds adjacents.

Quand les communications ne sont considérées que d'un point de vue local (d'un noeud à un autre noeud adjacent), ce type de réseau n'a pas de problèmes d'arbitrage ni de congestion.

Avec les deux liens point-à-point, l'échange de données entre noeuds distants mobilise tous les noeuds intermédiaires, et implique d'écrire explicitement la communication sur chacun d'entre eux. Si c'est une approche valable pour des solutions simples, quand le nombre de noeuds devient très élevé, la localité des communications rends la gestion des communications d'autant plus complexe.

Le portage d'une application avec un mode de communications réseau vers un mode de communication point-à-points est donc complexe. Une application qui a été modélisée avec des communications locales peut par contre être portée sur

une architecture avec routeur sans avoir à changer nécessairement le modèle de communication.

Il est nécessaire à terme de disposer d'un niveau d'abstraction des communications suffisant, car cette étape de portage peut fortement ralentir l'exploration d'architecture.

Deux approches sont envisageables afin d'apporter ce niveau d'abstraction. Le premier est l'écriture de librairies de communication, en implantant une librairie pour chaque mode de communication, chaque librairie aboutissant, à haut niveau, à un mode de programmation unique. La seconde approche est d'utiliser un outil d'aide à la parallélisation qui génèrera directement le code de communication bas niveau.

4.4 Résultats d'implantation de RHPC

4.4.1 Système avec lien de communication point à point FIFO

4.4.1.1 Couts d'implantation

Nous avons généré des architectures utilisant un réseau de communication point à point avec des liaisons FSL, de topologie hypercube, en variant le nombre de noeuds. La table 4.5 montre les coûts en ressources et la surface occupée par un système complet. Le système étant paramétré avec une configuration de processeur minimisant les coûts en ressources, une FIFO de profondeur 64 et 16Ko de mémoire dans chaque noeuds.

On peut voir sur cette table qu'un tel système composé de 32 noeuds, peut être placé dans un Virtex-4 LX200, en occupant moins des deux tiers de la surface. On peut aussi noter que ce type de lien exploite avantageusement les optimisations de la technologie Virtex-5 en matière d'implantation de FIFOs, et se place dans un Virtex-5 de taille équivalente en occupant cette fois ci moins de 25% de la surface.

Le coût total de ce même système de 32 noeuds comprend aussi celui des 32 processeurs et de la mémoire locale embarquée.

4.4.1.2 Performances

Si on considère uniquement des accès élémentaires entre les registres du processeur et le canal FSL, et une mise en oeuvre des des communications avec une programmation à très bas niveau, ce mode de communication a une latence par-

TABLE 4.5 – Ressources SoPC utilisées pour un système de 4, 8, 16, et 32 processeurs, topologie hypercube avec des liens point-à-point FSL.

Cible	Nb μP	surface	registres	LUTs	BRAM
XC4VLX200	4	9%	2%	6%	9%
XC4VLX200	8	11%	2%	10%	19%
XC4VLX200	16	26%	4%	24%	38%
XC4VLX200	32	60%	7%	55%	76%
XC5VLX330	4	3%	2%	3%	6%
XC5VLX330	8	6%	5%	6%	11%
XC5VLX330	16	12%	10%	14%	22%
XC5VLX330	32	23%	17%	28%	44%

ticulièrement réduite : un seul coup d'horloge[11]. On ne peut cependant se placer dans ce cas de figure que si une quantité extrêmement réduite de données est transférée (moins de 10 mots 32 bits).

Dans le cas moins spécifique ou le processeur accède à sa mémoire locale, il faut prendre en compte en supplément les temps de latence des accès en mémoire. Celle ci est de 2 coups d'horloge pour un accès au bus LMB.

Enfin, si une quantité élevée de données est échangée, les déplacements de données entre mémoire et canal FSL, sont nécessairement programmés dans une boucle. Le temps passé à exécuter les instructions de contrôle constituant la boucle peuvent avoir une influence déterminante sur les performances, étant donné les latences très réduites données précédemment.

Typiquement, la bande passante obtenue avec une telle boucle de transfert[12] sera de l'ordre $0.5 \times F$ O.s^{-1} (75 MO.s^{-1} avec un fréquence de 150 MHz). L'utilisation de techniques d'optimisation logicielles permettent d'améliorer légèrement ces performances.

Ce lien de communication permet donc d'atteindre des temps de latence très réduits, mais dont la bande passante est limitée par les performances du processeur et le degré d'optimisation des fonctions de communications.

[11] Les performances sont données dans cette section pour un MicroBlaze en version 5, les versions antérieures nécessitaient 2 coups d'horloge pour accéder au port FSL

[12] Ces chiffres ont été obtenus pour une boucle dépliée prenant 4 octets en mémoire LMB avec un MicroBlaze 5 et les envoyant sur son canal FSL.

4.4.2 Système avec lien de communication point à point DMA

4.4.2.1 Couts d'implantation

De la même manière que précédemment, avec le type de communication FSL, nous avons généré des architectures utilisant un réseau de communication point à point avec des liaisons DMA. La table 4.6 montre les coûts en ressources et la surface occupée par un système complet. Le système étant paramétré avec une configuration de processeur minimisant les coûts en ressources, 4Ko de mémoire tampon associée au DMA et 4ko de mémoire locale dans chaque noeud.

TABLE 4.6 – Ressources SoPC utilisées pour un système de 4, 8, 16, et 32 processeurs, topologie hypercube avec des liens point-à-point DMA.

Cible	Nb μP	surface	registres	LUTs	BRAM
XC4VLX200	4	6%	2%	5%	22%
XC4VLX200	8	11%	17%	19%	47%
XC4VLX200	16	38%	16%	30%	76%
XC4VLX200	32	93%	40%	71%	170%
XC5VLX330	4	4%	4%	5%	11%
XC5VLX330	8	13%	8%	10%	28%

4.4.2.2 Performances du canal de transfert DMA

Le temps d'initialisation du transfert DMA, qui correspond à programmation du contrôleur DMA par le processeur est d'environ 60 coups d'horloge processeur. A partir du moment où le contrôleur DMA prends en charge le transfert, celui ci deviens indépendant du processeur.

Le contrôleur de DMA transfère les données par séries de 16 mots de 32 bits, et met 40 coups d'horloge par série. Par conséquent, la bande passante mesurée est de $1.6 \times F$ O.s^{-1} c'est à dire 240 MO.s^{-1} à une cadencé de 150 MHz.

Si on compare les performances de transfert de données entre ce canal DMA et les performances de transfert de données en utilisant le canal FSL, Le canal DMA a une latence initiale élevée, mais une bande passante plus large.

4.4.3 Système avec routeur sans DMA

Sur la table 4.7 on peut voir les taux d'occupation de système composés de 8 noeuds, eux-mêmes étant composés d'un routeur et de processeurs Microblaze en configuration basique[13] et 16ko de mémoire locale.

Un réseau en hypercube de 8 processeurs avec 8 routeurs a de plus été implanté et testé sur SoPC (XC4VLX60), avec 88% de la surface disponible occupée. Les outils de synthèse indiquent que l'ont peut le faire fonctionner jusqu'à une frequence de 250MHz.

La version du routeur actuellement disponible (3 ports) ne permet pas encore de choisir le nombre de noeuds. Une version générique (avec un nombre variable de port), est en cours de développement.

TABLE 4.7 – Ressources SoPC utilisées par un système comprenant 8 noeuds (1 processeur, 1 routeur et 16ko de mémoire locale.)

Target	Nb μP	SLICE	FF	LUTs	BRAM
XC4VLX200	8	27%	14%	15%	19%
XC5VLX330	8	16%	11%	10%	11%

4.4.3.1 Performances d'un système avec routeur sans DMA

Nous avons mis en oeuvre et testé un système de 8 noeuds en connectant les routeurs directement aux processeurs. Nous mesurons donc cette fois les performances de communication du système dans son ensemble, influencés par les performances du processeur, de sa mémoire locale et du routeur.

Nous avons effectué ici l'envoi de messages d'un noeud à un autre à travers un réseau hypercube de dimension 3. Les processeurs émetteurs et récepteurs exécutent un simple programme de test effectuant les opérations décrites par la suite.

Le processeur émetteur lit dans sa mémoire locale un tableau contenant Q éléments de données, et envoie, flit après flit le message fractionné en paquets en entrée du routeur. De l'autre coté du réseau, le processeur destinataire reçoit, flit après flit les paquets et écrit en mémoire les Q données.

Le temps de transfert d'un paquet entre deux noeuds à une distance D avec une technique de routage de type wormhole est donné par la formule suivante :

$$Lat = (D-1) \times r + \frac{T}{Bw} \tag{4.5}$$

[13] Pas d'unité arithmétique optionnelle

Lat : Latence d'un paquet, ***Bw*** : Bande passante (flit/s), ***D*** : distance de communication, ***T*** : taille du paquet (flit), ***r*** : latence de transfert du routeur (s)

La mesure du temps de transfert entre deux processeurs situés à une distance D=3 pour des paquets de 16 flits a été de 78 coups d'horloge par paquet. Le temps que met le processeur à accéder à la mémoire et fractionner les données en paquets est d'environ 50 coups d'horloge. A l'autre bout le récepteur met un temps équivalent à retirer le paquet.

Par conséquent, la bande passante mesurée est de $0.72 \times F$ O.s^{-1} ce qui fait environ 110 Mo.s^{-1} pour un système avec une fréquence de 150 MHz.

On peut noter que la bande passante effective d'un système sans DMA est presque 4 fois moindre que la bande passante propre au routeur. En effet l'essentiel du temps de transfert dépend ici de l'efficacité du processeur à lire les données, et produire les paquets en émission, et à reconstituer les données puis à les placer en mémoire lors d'une réception.

L'insertion d'un interface réseau DMA-routeur permettra d'exploiter efficacement ces canaux 4.2.4.4 au prix d'un coût supplémentaire en ressource FPGA.

4.5 Conclusions

Dans cette section ont été décrit les composants matériels destinés à la mise en oeuvre d'un RHPC ciblant une technologie SoPC Xilinx.

Les résultats d'implantations donnent en premier lieu des éléments tangibles permettant de valider l'approche proposée dans sa globalité. Ils fournissent aussi des repères généraux pour l'exploration d'architecture. Dans la section suivante, nous allons éprouver concrètement la méthode présentée, à travers l'implantation d'une application de stabilisation vidéo.

Chapitre 5

Application de stabilisation vidéo temps réel.

Dans ce chapitre est présenté l'exemple de l'implantation d'une application de vision embarquée en choisissant un problème concret : celui de la stabilisation vidéo temps réel.

La stabilisation électronique d'images joue un rôle important dans plusieurs systèmes de vision artificielle, parmi ceux-ci la télé-opération, la robotique mobile, la reconstruction de scène, la compression vidéo, la détection d'objets mouvants, et beaucoup d'autres. Chacune de ces applications a ses spécificités, et le concept de "stable" peut donc varier selon les besoins et les contraintes de chaque application.

Nous nous intéressons au cas général d'une caméra embarquée sur un système mobile, et fixée à celui-ci de façon rigide. Cette configuration est fréquemment présente dans des systèmes de télé-opération ou d'aide à la conduite. La séquence d'images issue de cette caméra contient des informations concernant le déplacement du véhicule dans son environnement. Ce déplacement peut être scindé en deux composantes : celle due aux mouvements commandés du véhicule, et une deuxième composante issue des mouvements parasites (non-commandés) subis par la caméra (rugosité du terrain, vibrations, etc.). Cette composante parasite peut, selon son amplitude, nuire de façon très significative à la visualisation et à la compréhension des images par un observateur/opérateur humain ou par un système de vision artificielle.

Nous sommes donc dans le cas de la stabilisation sélective : stabiliser une séquence d'image revient à éliminer ou atténuer la composante de mouvements non intentionnels dans la vidéo, tout en conservant les mouvements intentionnels intacts.

Si la stabilisation électronique d'images est un sujet qui a déjà fait l'objet d'un grand nombre de travaux et publications, l'approche architecturale permettant à de tels systèmes de fonctionner en temps réel, tout en respectant les contraintes d'un dispositif embarqué, est un sujet moins étudié. Il est néanmoins important de valider l'approche algorithmique dans des conditions expérimentales réalistes, notamment au niveau temporel. Pour cela, ce type d'algorithme nécessite une architecture plus ou moins spécialisée et/ou capable de traitement parallèle.

Les approches de stabilisation basées sur des algorithmes de stabilisation digitale fournissent une solution viable en terme de performances et présentent surtout l'avantage d'être mise en oeuvre avec une architecture matérielle de traitement d'image moins encombrante et souvent moins couteuse qu'une approche basée sur l'intégration de la camera dans un support de compensation électromécanique.

Nous avons développé une application de stabilisation vidéo électronique que nous avons parallélisée tout d'abord sur un cluster de station de travail[31] puis que nous avons embarqué sur une architecture multiprocesseurs dédiée .

Dans un premier temps, nous avons validé la précision et la robustesse de notre algorithme sur une architecture séquentielle, et dans un second temps nous avons effectué son implantation temps réel en utilisant les parallélismes potentiels des structures COTS. Dans un troisième temps, l'architecture a été portée pour la plateforme matérielle décrite dans le chapitre précédent, l'algorithme a été retravaillé, de manière à respecter les contraintes spécifiques à un système embarqué, tout en conservant le schéma de parallélisation.

5.1 La stabilisation électronique d'images

Depuis quelques années plusieurs méthodes de stabilisation électronique d'images ont été proposées. Ces méthodes peuvent être classifiées dans trois familles principales, selon le modèle d'estimation du mouvement adopté : les méthodes 2D ou planaires [60], les méthodes 3D [37] et les méthodes 2,5D [82]. En effet, les algorithmes de stabilisation sont constitués d'une séquence de traitements de différents niveaux, appliqués sur les images successives de la séquence vidéo. Généralement, trois modules de traitement sont présents :

1. la mise en correspondance entre les images ;
2. l'estimation du déplacement global (au moyen du modèle de mouvement choisi) ;
3. la correction/compensation des mouvements afin d'obtenir une séquence stable.

5.1.1 Mise en Correspondance

L'objectif de la mise en correspondance est de calculer le déplacement d'un point ou région de la scène réelle dans le plan image 2D. Ce déplacement dans le plan image est la projection du déplacement 3D de l'objet dans la scène observée. Deux techniques sont couramment employées : le calcul du flot optique [37], et la détection et suivi de primitives visuelles [60], méthode qui sera développée en section 5.2.

Le calcul du flot optique est un outil classique de traitement d'images, qui a déjà été le sujet d'innombrables publications, dont nous citerons [51] et [15]. Même si cette technique a déjà été employée dans le cadre de la stabilisation d'images, elle présente quelques inconvénients, comme par exemple la complexité des calculs demandés, qui peut être relativement élevée, ou la validité du flot optique comme projection du champ de mouvement 3D, discutée dans [75].

Notre option pour la mise en correspondance des images a donc été la détection et suivi de primitives visuelles. La technique consiste, dans un premier temps, à localiser dans l'image i des régions riches en information visuelle (fort contraste de luminance, coins, bords, etc.), et dans un deuxième temps, à retrouver ces mêmes régions à l'image $i+1$. Pour détecter ces régions riches en informations, appelées primitives, différentes techniques existent. Le détecteur de coins et bords de Harris et Stephens [48] est une des techniques les plus connues. D'autres possibilités sont les opérateurs Laplaciens ou les ondelettes de Harr, technique expliquée dans la prochaine section.

Après avoir détecté les primitives, il faut être capable de les retrouver dans une autre image et notamment dans l'image suivante. Cela est fait à l'aide d'une fonction de corrélation et d'une stratégie de recherche. Les techniques de recherche multi-résolution permettent de réduire le temps de recherche au moyen d'une approche *"coarse-to-fine"*. Différentes fonctions de corrélation peuvent être employées, depuis les plus classiques (SSD, SAD), jusqu'à d'autres robustes par exemple aux changements d'illumination dans la scène, comme la corrélation normalisée [74]. Dans [66], plusieurs techniques de corrélation sont présentées et comparées.

5.1.2 Estimation du Déplacement

Une fois que les images successives de la séquence ont été mises en correspondance, une deuxième étape du traitement commence. Il s'agit de la détermination des paramètres décrivant le mouvement entre les images. Dans ses travaux de thèse[42] Gensolen présente une rétine CMOS intelligente où l'extraction du

mouvement global est effectuée à partir de mesures de déplacements locaux en périphérie des images. Dans le cas de notre application, les paramètres à déterminer dépendent du modèle de mouvement choisi. Les modèles 2D supposent une scène planaire ou presque, c'est-à-dire que tous les points mis en correspondance et utilisés dans l'estimation se trouvent à plus ou moins la même distance de la caméra. Dans ce cas, il y a trois paramètres à estimer : deux translations (horizontale et verticale) et une rotation autour de l'axe optique de la caméra. Un quatrième paramètre peut être introduit, afin de prendre en compte les changements d'échelle dus à l'avancement de la caméra [60].

Les modèles 3D assument généralement que seules les rotations 3D parasites sont significatives. Il suffit donc d'estimer et corriger celles-ci pour stabiliser la séquence. Comme l'effet sur les images dû aux rotations de la caméra est indépendant de la profondeur des points, il est alors possible d'estimer ces rotations, à l'aide par exemple des quaternions [60].

Le modèle 2,5D présenté dans [82] suppose que quelques connaissances à priori sur le mouvement de la caméra sont disponibles, et estime trois paramètres globaux du mouvement, plus un paramètre indépendant pour chaque point analysé, lié à sa profondeur (distance par rapport à la caméra). Cela permet de mieux prendre en compte la complexité structurale d'une scène, sans pourtant faire appel à un modèle 3D.

5.1.3 Compensation des mouvements

Finalement, après l'estimation du mouvement global entre les images, il faut procéder à la correction de sa composante non voulue. Cette dernière étape du traitement est fortement liée à l'application, car c'est celle-ci qui définie quelle partie du mouvement doit être conservée, et quelle partie doit être compensée. Quelques méthodes adoptées sont la compensation totale du mouvement estimé, l'application d'un filtre numérique passe-bas ou inertiel [82], ou l'approximation polynomiale des rotations 3D estimées [37].

5.2 Notre Méthode de Stabilisation

5.2.1 Descriptif général

Nous avons développé [8] [30] une méthode de stabilisation basée sur un modèle de mouvement 2D, avec détection de primitives par ondelettes de Harr. Comme dans [76], le calcul des ondelettes est effectué à partir d'une image transformée

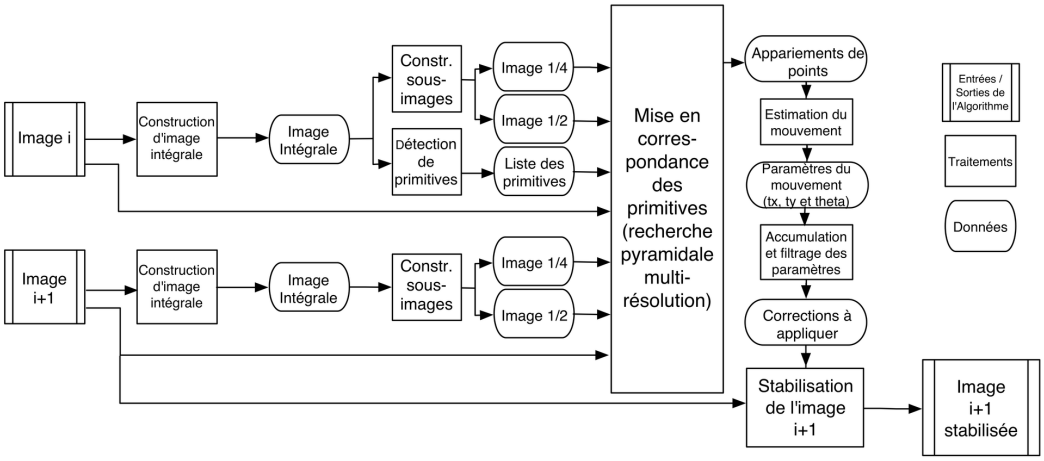


FIGURE 5.1 – Schéma synoptique d'une itération de l'algorithme de stabilisation.

(*image intégrale*). Deux autres images de résolution inférieure ($\frac{1}{2}$ et $\frac{1}{4}$) sont aussi calculées à partir de l'image intégrale, afin de permettre la recherche des points correspondants des primitives à partir d'une approche pyramidale sur trois niveaux de résolution. Celle-ci est effectuée en appliquant une fonction de corrélation (SAD - *Sum of Absolute Differences* ou SSD - *Sum of Square Differences*) entre le motif recherché (primitive), et ses potentiels correspondants dans l'image suivante. Une fois obtenus les appariements de points entre deux images successives, les paramètres du modèle 2D (t_x, t_y et θ) sont estimés par la méthode des Moindres Carrés Médians. Cette méthode est robuste aux faux appariements qui peuvent avoir lieu lors de l'étape précédente. Finalement, et afin d'isoler la composante non voulue du mouvement, les paramètres de déplacement sont filtrés par un filtre numérique passe-bas, et ensuite appliqués pour corriger l'image correspondante dans le but de stabiliser la séquence (figure 5.1).

5.2.2 Construction de l'image intégrale

Une image intégrale est calculée à partir de chaque image reçue depuis la caméra (256 niveaux de gris, taille paramétrable). Cette image est utilisée pour la détection par ondelettes, mais aussi pour la génération de deux images sous-échantillonnées ($\frac{1}{2}$ et $\frac{1}{4}$).

L'image intégrale est calculée selon l'équation 5.1. Elle contient à la position (X,Y) la somme de tous les pixels contenus à l'intérieur du rectangle borné par $i(0,0)$ et $i(X,Y)$, où $i(x,y)$ est l'image originale, et $ii(x,y)$ est l'image intégrale :

$$ii(X,Y) = \sum_{\substack{x \leq X \\ y \leq Y}} i(x,y) \tag{5.1}$$

A partir de l'image intégrale, la somme de tous les pixels contenus dans une zone rectangulaire peut être obtenue en quatre accès à l'image et 3 additions, indépendamment du nombre de pixels dans cette zone (fig. 5.2 et eq. 5.2).

$$\begin{aligned} \sum_{\substack{x_1 \leq x \leq x_2 \\ y_1 \leq y \leq y_2}} i(x,y) &= ii(x_2,y_2) + ii(x_1-1,y_1-1) \\ &\quad - ii(x_2,y_1-1) - ii(x_1-1,y_2) \end{aligned} \tag{5.2}$$

10	10	9	9	8	7	6	5	6	5
10	10	12	12	11	10	8	6	6	4
9	8	15	14	14	12	7	5	2	1
8	9	16	18	16	14	9	4	2	2
7	9	12	14	12	11	7	5	1	3
8	9	10	12	11	11	8	5	3	2
12	11	13	13	12	10	10	6	4	4
15	14	15	15	13	11	10	7	4	5
15	14	15	15	13	11	10	7	4	5
19	16	15	12	10	9	8	8	5	5

10	20	29	38	46	53	59	64	70	75
20	40	61	82	101	118	132	143	155	164
29	57	93	128	159	190	211	227	241	251
37	74	126	179	224	271	301	321	337	349
44	90	154	221	282	336	373	398	415	430
52	107	181	260	332	397	442	472	492	509
64	130	217	309	393	468	523	559	583	604
79	159	261	368	465	551	616	559	867	713
97	193	312	432	541	639	715	767	801	832
106	228	362	494	613	720	804	864	903	939

FIGURE 5.2 – Image 10x10 (à gauche) et son image intégrale (à droite).

Afin de réduire le nombre d'opérations effectuées, le calcul de l'image intégrale utilise les formules de récurrence ci-dessous (eqs. 5.3), où $s(x,y)$ est une valeur intermédiaire (somme accumulée ligne par ligne sur une même colonne) :

$$\begin{aligned} s(x,y) &= s(x,y-1) + i(x,y) \\ ii(x,y) &= ii(x-1,y) + s(x,y) \\ s(x,0) &= i(x,0) \text{ et } ii(0,y) = s(0,y) \end{aligned} \tag{5.3}$$

Quand elles sont appliquées de façon directe, les méthodes de détection par ondelettes et de calcul d'images sous-échantillonnées demandent un grand nombre

de sommes effectuées sur l'image. Avec l'utilisation de l'image intégrale comme image intermédiaire, nous pouvons calculer directement la somme sur un rectangle de taille quelconque avec seulement 4 accès à l'image intégrale et 3 sommes [76], au lieu de calculer la somme des pixels un à un.

5.2.3 Détection de primitives

Les primitives sont recherchées en appliquant les ondelettes dans une zone de détection de taille $q \times r$. Nous utilisons la moitié supérieure de l'image, ce qui dans une scène d'extérieur permet de détecter les primitives situées à l'horizon. Ces régions sont normalement assez éloignées de la caméra, ce qui permet de respecter au mieux la contrainte de scène planaire imposée par le modèle 2D.

La zone de recherche est divisée en $\frac{n}{3}$ bandes verticales de taille $(3\frac{q}{n}, r)$, où n est le nombre de primitives recherchées fixé par l'utilisateur. La division en bandes permet d'obtenir une bonne distribution spatiale des primitives détectées. Avec par exemple une image de format $T_x \times T_y = 1280 \times 960$, on introduit les paramètres $q = 1024$ et $r = 384$, résultant dans des bandes verticales de taille 128×384 pour $n = 24$. Les valeurs q et r représentent les dimensions de la moitié supérieure de l'image, moins une marge de détection de chaque côté. Cette marge sert à empêcher la détection de primitives trop proches du bord de l'image, et qui, dû aux mouvements de la caméra, risquent de ne pas être présentes à l'image suivante, rendant ainsi impossible leur future mise en correspondance.

Chaque bande est balayée par trois types d'ondelettes : une verticale, une horizontale et une diagonale. L'application d'une ondelette consiste dans la convolution d'une région 10×10 de l'image avec un des masques présentés dans la figure 5.4. La valeur retournée représente le gradient de luminance du motif dans une direction donnée. Pour chacune des trois ondelettes, le point qui retourne la valeur de gradient la plus élevée est retenu comme primitive pour le suivi (fig. 5.3).

Le calcul des ondelettes est fortement accéléré par l'utilisation de l'image intégrale. Le calcul d'une ondelette 10×10, sans utiliser l'image intégrale, nécessite 100 accès image et 99 opérations de somme. En utilisant l'image intégrale, une ondelette horizontale ou verticale est calculée en seulement 6 accès à l'image intégrale, 5 additions et 2 multiplications. Une ondelette diagonale est calculée en 9 accès, 8 additions et 1 multiplication, pour n'importe quelle taille d'ondelette.

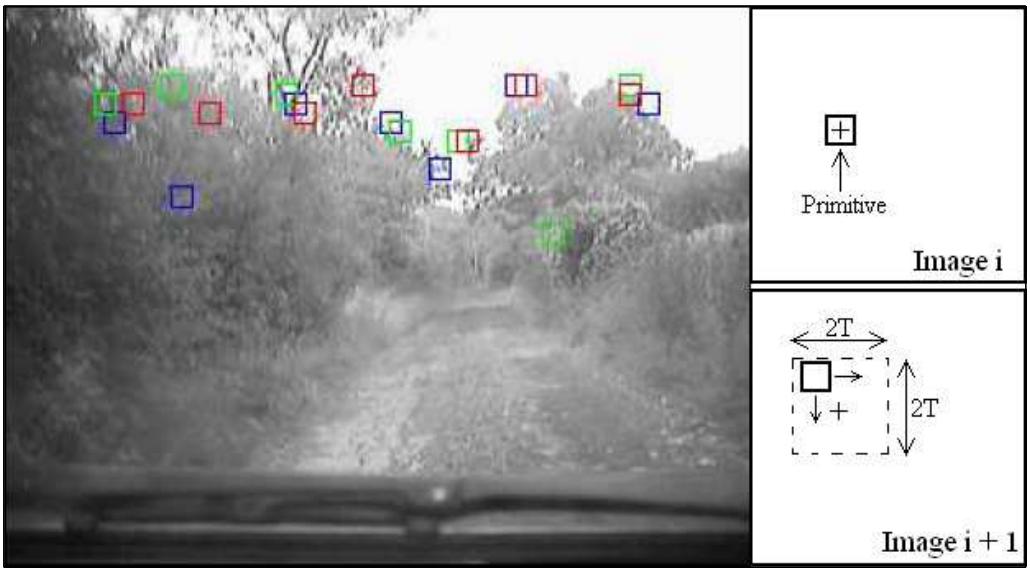


FIGURE 5.3 – A gauche, primitives sélectionnées par les ondelettes de Harr sur une séquence d'images réelle. A droite, schéma de la mise en correspondance des primitives entre deux images.

5.2.4 Construction des images sous-échantillonnées

Dans cette étape sont générées les images sous-échantillonnées utilisées lors de la recherche pyramidale. Ces images sont obtenues en calculant la luminance moyenne sur une zone carrée de 4 pixels (résolution $\frac{1}{2}$) ou 16 pixels (résolution $\frac{1}{4}$). Au moyen de l'image intégrale, le calcul de chaque pixel des deux images sous-échantillonnées nécessite 3 sommes et une division.

5.2.5 Mise en Correspondance de primitives

Considérant que l'étape de détection a été réalisée sur l'image i, nous allons maintenant rechercher dans l'image $i+1$ les correspondants des n primitives retenues. Tout d'abord, une fenêtre de recherche de taille $2T \times 2T$ est définie autour de la position où la primitive a été détectée (le paramètre T défini la zone de recherche comme indiqué figure 5.3). Ensuite, la corrélation entre chaque région à l'intérieur de cette fenêtre et le motif retenu comme primitive à l'image i est calculée. La région de l'image $i+1$ qui présente un meilleur score de corrélation est considérée comme étant la correspondante à cette primitive. Nous avons donc un appariement de points entre les deux images consécutives. Cette opération est réalisée

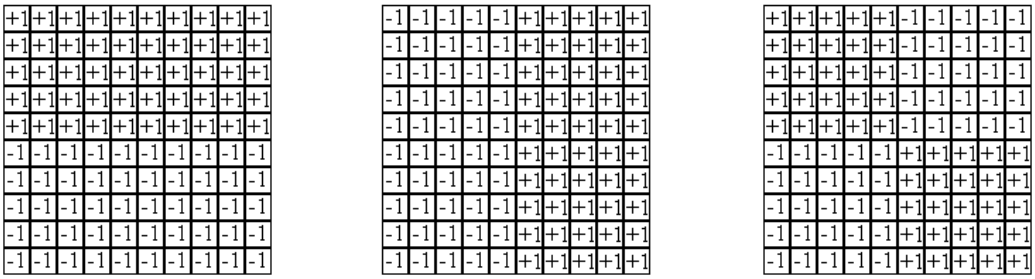


FIGURE 5.4 – Les masques des trois types d'ondelettes employés. De gauche à droite : ondelettes verticale, horizontale et diagonale.

pour chacune des n primitives détectées.

Afin de diminuer le nombre d'opérations à réaliser, la recherche est faite au moyen d'une approche multi-résolution. Nous utilisons d'abord l'image sous-échantillonnée $\frac{1}{4}$, et la recherche est faite à l'intérieur d'une fenêtre de taille $\frac{T}{2} \times \frac{T}{2}$. Cela nous donne une première estimation de la position du correspondant.

A partir de cette estimation, une deuxième recherche est effectuée sur l'image sous-échantillonnée $\frac{1}{2}$. Cette fois-ci, la recherche est faite à l'intérieur d'une fenêtre 3×3 autour de la position estimée précédemment. Une deuxième estimation est donc obtenue, plus précise que la première. Finalement, une dernière recherche est faite sur l'image originale, avec une précision sous-pixelique ($\frac{1}{8}$ de pixel). Les intensités sous-pixeliques sont estimées via une interpolation bilinéaire sur une fenêtre 2×2.

5.2.6 Estimation du mouvement

En possession des n appariements de points entre les images i et $i+1$, les paramètres du modelée 2D, décrivant le mouvement d'une image à l'autre, peuvent être estimés. Il est supposé que le déplacement peut être modélisé par une matrice de transformation homogène (eq. 5.4), constituée d'une rotation autour de l'axe optique (θ), d'une translation horizontale (t_x) et d'une translation verticale (t_y). Le déplacement d'un point (x, y) entre deux images est donc décrit par le système ci-dessous, où (x', y') est la position du point (x, y) après déplacement :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta & t_x \\ -\sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{5.4}$$

Les trois paramètres inconnus de la matrice (θ, t_x et t_y) sont estimés en ap-

pliquant ce modèle aux n appariements de points retrouvés à l'étape de mise en correspondance. Notons p_1 le nuage de points d'intérêt détectés à l'image i, et p_2 le nuage formé par leurs correspondants trouvés à l'image $i+1$. Les deux nuages de points sont centrés par rapport à leurs barycentres respectifs, donnant deux nuages centrés p'_1 et p'_2, liés par une rotation autour de l'origine : $p'_2 = R.p'_1$. Il faut donc minimiser le critère :

$$S = \sum_i \left\| p'_{2i} - R.p'_{1i} \right\|^2 \tag{5.5}$$

La minimisation du critère S donne le paramètre θ, et une fois en possession de l'angle de rotation les deux paramètres de translation sont déduits à partir du mouvement des barycentres des deux nuages. Dans les équations ci-dessous (eqs. 5.6), (b_{1x}, b_{1y}) et (b_{2x}, b_{2y}) sont les coordonnées du barycentre des nuages p_1 et p_2 respectivement :

$$\begin{aligned} t_x &= b_{2x} - b_{1x}.\cos\theta - b_{1y}.\sin\theta \\ t_y &= b_{2y} + b_{1x}.\sin\theta - b_{1y}.\cos\theta \end{aligned} \tag{5.6}$$

L'estimation des paramètres du mouvement est réalisée au moyen d'un filtrage par les moindres carrés médians, technique robuste qui tolère jusqu'à 50% de données aberrantes.

5.2.7 Accumulation et filtrage des paramètres

Les paramètres estimés sont accumulés avec ceux calculés précédemment, afin de trouver le déplacement global de la caméra au long de la séquence. Les valeurs θ, t_x et t_y calculées précédemment sont appliquées pour obtenir la matrice de transformation de l'image i à l'image $i+1$ (eq. 5.7). Cette matrice est multipliée par la matrice Mat_i, contenant l'accumulation des déplacements depuis le début de la séquence (eq. 5.8). Nous obtenons donc la matrice de transformation Mat_{i+1}, qui décrit le déplacement de l'image $i+1$ par rapport à un repère de référence.

$$Mat_{i \to i+1} = \begin{pmatrix} \cos\theta & \sin\theta & t_x \\ -\sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

$$Mat_{i+1} = Mat_{i \to i+1} \cdot Mat_i \tag{5.8}$$

Un filtre linéaire du premier ordre est appliqué indépendamment à chaque paramètre (eqs. 5.10). Les coefficients des trois filtres peuvent être réglés par l'utilisateur (K_x, K_y, K_θ). Cela permet d'obtenir une stabilisation sur mesure selon les contraintes de l'application. Le découplage des filtres permet encore d'avoir différents niveaux de stabilisation pour la rotation et les translations.

$$\begin{aligned} t_x[i+1] &= Mat_{i+1}[1,3] * (1-K_x) \\ t_y[i+1] &= Mat_{i+1}[2,3] * (1-K_y) \\ \theta[i+1] &= acos(Mat_{i+1}[1,1]) * (1-K_\theta) \end{aligned} \tag{5.9}$$

Finalement, les valeurs filtrées sont utilisées pour obtenir la matrice de déplacement inverse, qui est donc appliquée à l'image $i+1$ afin de la stabiliser, c'est-à-dire, de la ramener vers le repère de référence (fig. 5.5). Ce repère de référence est dynamique, et essaie d'accompagner les mouvements commandés de la camera, selon les coefficients des filtres choisis par l'utilisateur.

FIGURE 5.5 – Séquence d'images synthétiques instable (en haut) et séquence après stabilisation (en bas).

5.3 Qualification de la Stabilisation

Afin de valider l'algorithme de stabilisation expliqué précédemment, nous avons proposé et conçu une méthode de qualification. Ceci s'avère nécessaire car une simple vérification visuelle du fonctionnement de l'algorithme ne nous permet pas

d'obtenir des données quantitatives indispensables. Ces informations permettent de mieux comprendre et prévoir le comportement du système face à diverses situations d'application, sans avoir à tester le dispositif de façon exhaustive.

La qualification de la stabilisation d'images est un sujet peu discuté dans la communauté scientifique. Du fait que le concept de stabilité peut varier en fonction du domaine d'application, qualifier un tel algorithme sans tenir compte du cadre d'utilisation s'avère une tâche complexe. Les différents aspects pouvant intervenir dans la performance d'un tel système sont :

- la précision d'estimation du mouvement de la caméra ;
- le déplacement maximal d'un motif visuel pouvant être mesuré par le système ;
- le confort de visualisation de la séquence résultante ;
- le temps de calcul nécessaire pour traiter chaque image de la séquence.

Dans [14], une méthode de qualification indirecte est proposée. L'idée est d'appliquer la stabilisation comme prétraitement pour un autre algorithme de vision, comme par exemple un dispositif de détection et suivi de cibles. Cette approche permet de vérifier la validation du dispositif face à une application réelle, en mesurant l'apport de la stabilisation sur les performances d'une autre application. Par contre, elle ne permet pas d'obtenir des données quantitatives.

Une méthode d'obtention de la précision d'estimation est présentée dans [60]. La somme des différences entre chaque pixel de deux images consécutives préalablement stabilisées est calculée. En supposant une scène statique, si la stabilisation est parfaite les deux images sont identiques, et la différence est nulle. L'inconvénient majeur de cette méthode est sa forte sensibilité au contenu des images, notamment le contraste. De plus, même si les résultats obtenus sont quantifiables, ils ne sont pas directement liés aux paramètres impliqués lors du processus de stabilisation.

Pour obtenir la précision réelle d'estimation du mouvement de la caméra, il est indispensable d'avoir une vérité de terrain sur le déplacement de celle-ci, ce qui n'est pas possible à obtenir pour des séquences réelles. Nous avons donc choisi de qualifier notre approche en utilisant des séquences d'images synthétiques, dont nous maîtrisons le modèle de déplacement et le contenu des images. Ainsi, il est possible de comparer les déplacements réels avec ceux estimées par l'algorithme, obtenant donc l'erreur d'estimation.

Ces séquences synthétiques de test sont générées à partir d'une image de référence et d'un modèle de déplacement, choisis en fonction du domaine d'application visé. L'image choisie comme référence doit contenir les mêmes textures et objets qui seront retrouvés lors d'une utilisation sur le terrain : arbres, ciel, route, horizon. Une image réelle prise par une caméra embarquée peut être utilisée (fig. 5.5). De même, le modèle de déplacement utilisé doit être cohérent avec les mouvements auxquels la caméra est généralement soumise sur le terrain. Nous utilisons un modèle d'évolution du mouvement extrait d'une séquence réelle, prise à partir d'un véhicule roulant sur un terrain accidentée.

Des transformations 2D rigides sont appliquées successivement à l'image de référence, au moyen d'une matrice de déplacement contenant les paramètres du mouvement définis dans le modèle. Chaque transformation génère une nouvelle image de la séquence, simulant ainsi une caméra qui se déplace parallèlement au plan optique, et observant une scène statique. Ces mouvements sont planaires et homogènes sur toute l'image, respectant parfaitement les contraintes du modèle planaire 2D qui sont assumées dans l'algorithme. Le fonctionnement et la précision de la stabilisation peuvent donc être observés, indépendamment de la validité ou non des hypothèses du modèle planaire 2D sous certaines conditions réelles (cette validité ne peut être vérifiée qu'au cas par cas, selon le type de scène traitée).

Les tests sur plusieurs séquences synthétiques montrent que l'approche proposée est très précise. Les erreurs d'estimation sont inférieures à 0,2 pixels pour les translations, et inférieures à 0,05 degrés pour les rotation.

Le déplacement maximal pouvant être mesuré par notre système est réglé par l'utilisateur (paramètre T), ayant une incidence directe sur le temps de calcul. Le plus loin on cherche, le plus de temps on passera à chercher. Ce couplage évident entre le déplacement maximal et le temps d'exécution est extrêmement important. En effet, la grandeur à maximiser est la multiplication du déplacement maximal (en pixels) par le nombre d'images traitées par seconde. Cela indique la valeur maximale de la vitesse d'un objet, en pixels par seconde, pour qu'il puisse être suivi par le dispositif. Ainsi, le paramètre T doit être réglé en fonction du temps de calcul et de son influence sur celui-ci.

Si l'estimation du déplacement de la caméra est précise, le confort de visualisation obtenu dépend entièrement du filtrage réalisé. Grâce aux filtres paramétrables, notre méthode de stabilisation est capable de s'adapter à un très large éventail de situations, permettant le rendu d'une séquence stabilisée de facile visualisation et compréhension.

Les temps de calcul de l'algorithme, étant très liés à l'architecture matérielle employée, sont présentés et analysés dans les prochaines sections.

5.4 Présentation de l'architecture Babylon

La chaîne de traitement décrite en section 5.2 a été développée dans un premier temps sur une machine de type PC, équipée d'un processeur AMD Athlon XP 1700. Après l'évaluation et la validation de l'efficacité de l'algorithme, celui-ci a été parallélisé et porté sur la plateforme Babylon, pour ensuite être implantée sur notre plateforme SoPC.

La plateforme Babylon est une architecture symétrique biprocesseur à mémoire partagée. Son système d'exploitation est MacOS X. Elle dispose de capacités de traitements parallèles exploitables à trois niveaux :

- Au niveau de l'exécution sur un seul processeur, avec des mécanismes d'exécution super scalaire, et avec le jeux d'instructions SIMD Altivec ;
- Au niveau d'une machine (noeud), possédant deux processeurs pouvant travailler simultanément et partageant la mémoire (SMP) ;
- Au niveau d'un cluster de machines interconnectées, constituant une architecture à mémoire distribuée (MIMD-DM).

5.4.0.1 Jeu d'instruction SIMD

Ce type d'extension est rencontré dans la plupart des microprocesseurs actuels : MMX, SSE et SSE2 pour les processeurs Intel, 3DNOW ! pour les processeurs AMD, MDMX pour les processeurs MIPS et VIS pour les processeurs SPARC. Ces extensions du jeu d'instructions sont toutes fondées sur deux principes :

- Le premier est d'offrir des capacités de traitement de type SIMD : effectuer simultanément une même opération arithmétique ou logique sur un ensemble de données avec une seule instruction (figure 5.6).
- Le deuxième principe est de fournir un jeu d'instructions fortement inspiré des architectures DSP : arithmétique saturée, opérateurs câblés, opérateurs de conversion de type, etc. (figure 5.7).

Ces instructions s'appliquent sur des registres de taille fixe (128 bits pour Altivec), mais dont le nombre d'éléments traités simultanément peut varier : les éléments 32 bits sont traités par lots de 4, ceux de 16 bits sont traités par lots de 8 et ceux de 8 bits par lots de 16.

L'utilisation des jeux d'instructions SIMD permet de mettre en oeuvre un parallélisme à grain fin particulièrement adapté aux traitements réguliers. Dans ce

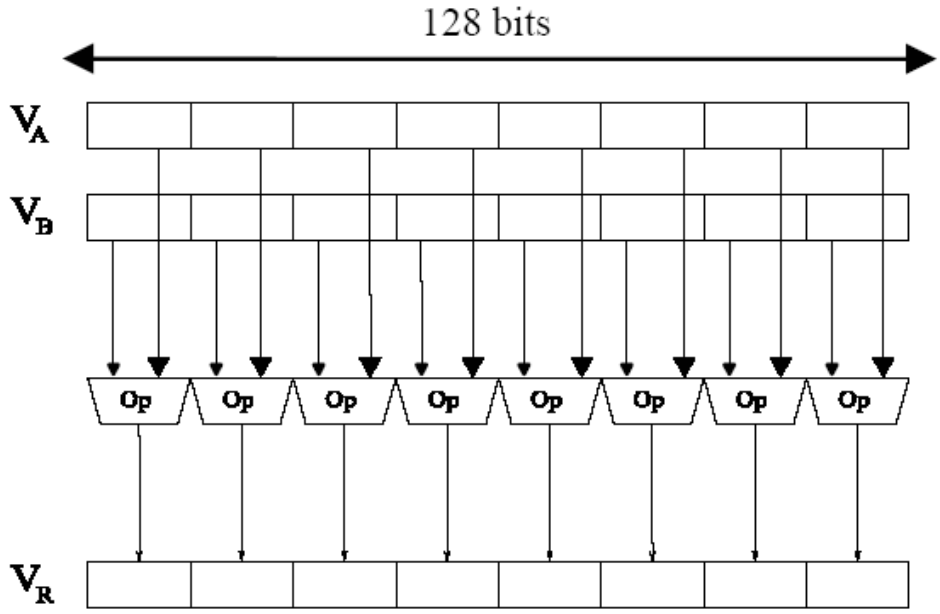


FIGURE 5.6 – Instruction C Altivec à deux opérandes : **VR = vec_add(VA, VB)**.

cas, il est démontré que si le ratio "opérations de calcul/opérations mémoire" d'un traitement est suffisamment élevé, il est possible d'obtenir des accélérations quasi-linéaires (4, 8 ou 16) pour les calculs sur des nombres entiers, voire des accélérations sur-linéaires dans le cas des calculs en virgule flottante. Les performances obtenues grâce à ces jeux d'instructions sont discutées dans [71] et [38].

Cependant, la parallélisation SIMD limite son champ d'application aux traitements réguliers. Un autre inconvénient est qu'elle oblige à rester à un faible niveau d'abstraction. En conséquence, des parties entières du programme qu'on souhaite accélérer sont à réécrire.

5.4.0.2 Architecture SMP

Le second niveau de parallélisme réside dans l'exploitation de deux processeurs communicant via une mémoire partagée. Cette architecture permet de mettre en oeuvre un parallélisme gros grain, en répartissant des tâches ou en découpant des blocs de données entre les processeurs.

Le système d'exploitation MacOS X est conçu pour répartir l'exécution des différentes tâches sur plusieurs processeurs, gérant l'allocation des tâches en cours d'exécution par planification préemptive. L'exécution en parallèle se fait alors de façon complètement transparente pour l'utilisateur. Cependant, afin que le sys-

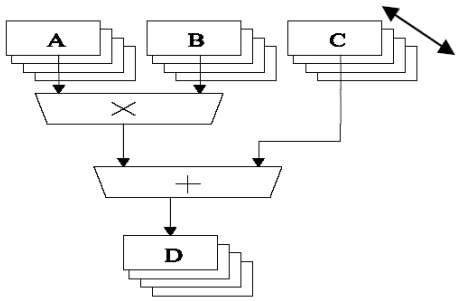


FIGURE 5.7 – Instruction C Altivec de type DSP : **VD = vec_madd(VA, VB, VC)**.

tème exécute des tâches appartenant à une même application sur différents processeurs, il faut séparer ces tâches explicitement, sous forme de processus légers (threads). Dans le cas de la chaîne de stabilisation, deux processus légers sont donc crées et exécutés en concurrence, permettant alors au système de repartir leur exécution entre les ressources processeur disponibles.

La création de ces processus est possible en faisant des appels système avec la bibliothèque de fonctions standard "pthread". Cette bibliothèque définit des primitives de création et fusion de processus, ainsi que des mécanismes de verrous pour permettre de synchroniser et mettre en place des mécanismes d'exclusion mutuelle. A la différence des jeux d'instructions SIMD, le niveau d'abstraction élevé permet une implantation parallèle sans avoir nécessairement à réécrire intégralement les traitements à paralléliser.

5.4.0.3 Architecture MIMD-DM

L'architecture utilisée est composée de 14 machines PowerMac G5, interconnectées avec un switch en réseau Ethernet Gigabit, permettant de mettre en oeuvre un parallélisme gros grain. Comme chaque machine comprend 2 processeurs en mémoire partagée, l'architecture décrite totalise 28 processeurs avec une organisation mémoire hétérogène (NUMA).

Dans un souci d'homogénéité, toutes les communications se font par passage de message, aussi bien entre processeurs de noeuds distincts qu'entre processeurs d'une même machine. Ces communications sont effectuées avec la librairie normalisée MPI (Message Passing Interface). Implantée sur un grand nombre de plateformes, cette bibliothèque apporte plus d'une centaine de fonctions de niveau d'abstraction élevé. Elle définit notamment un ensemble de primitives de communication point à point et d'opérations collectives, simplifiant la mise en oeuvre de structures parallèles sur machines MIMD-DM.

5.5 Implantation parallèle

Afin de proposer un schéma de parallélisation exploitant au mieux les structures architecturales définies précédemment, la démarche proposée tend à optimiser d'une part la rapidité de portage de l'algorithme et d'autre part les résultats en terme de temps d'exécution.

5.5.1 Analyse de l'algorithme séquentiel

Les différentes étapes de la version séquentielle de l'application ont été analysées selon deux critères : ratio "traitements/opération mémoire" et volume de données traitées (tableau 5.1). L'importance relative des différentes étapes de l'algorithme par rapport au temps total d'exécution de la boucle de stabilisation en mode séquentiel est indiquée sur le tableau 5.2. En se basant sur ces analyses, il est possible de concentrer l'effort de parallélisation sur les étapes permettant à priori un gain plus notable. Nous pouvons constater que les étapes *step_1*, *step_2* et *step_4* sont les plus consommatrices de puissance de calcul. Grâce à l'utilisation de l'image intégrale pour le calcul des ondelettes, l'étape *step_3* a des temps d'exécution négligeables par rapport aux autres étapes de l'algorithme.

TABLE 5.1 – Caractéristiques des différentes étapes de l'application.

Etapes	Parallélisme intrinsèque	Ratio traitements / opération mémoire	Volume de données traitées
step_1 : Calcul de l'image intégrale	données	faible	élevé
step_2 : Sous-échantillonnage des images	tâches et données	faible	élevé
step_3 : Détection des primitives	tâches et données	faible	faible
step_4 : Suivi de primitives par corrélation	tâches et données	très élevé	très élevé
step_5 : Estimation du mouvement	séquentiel	faible	faible
step_6 : Filtrage du mouvement	séquentiel	faible	très faible

Il apparaît que *step_4* est l'étape qui a le temps d'exécution le plus important. Cependant, avec l'augmentation de la taille de l'image, l'importance relative du calcul des images intermédiaires (*step_1* et *step_2*) augmente. Un mode d'exécution parallèle de ces trois étapes est donc souhaitable, afin d'observer un gain de performance intéressant. Nous avons choisi d'implanter les quatre pre-

TABLE 5.2 – Importance relative des différentes étapes en fonction du format vidéo et de la distance de recherche.

Format vidéo	**320 × 240**	**640 × 480**	**1280 × 960**	**2560 × 1920**	**5120 × 3840**
valeur de T	15	30	60	120	240
step_1 et *step_2*	1%	3%	6%	9%	10%
step_4	97%	95%	93%	90%	90%
step_5 et *step_6*	2%	2%	1%	1%	0%

mières étapes en parallèle. Les deux dernières, ne présentant pas une complexité importante ($< 2\%$), sont réalisées en mode séquentiel. L'étape *step_3* est parallélisée non pas pour obtenir des gains temporels, car ces temps d'exécution sont déjà assez faibles, mais afin de faciliter la parallélisation, évitant une opération de *merge-split* entre les étapes *step_2* et *step_4*.

5.5.2 Description de l'implantation parallèle

5.5.2.1 Implantation sur Babylone

Il est rappelé que l'architecture SMP, tout comme l'architecture de type grappe, offre deux niveaux de parallélisme : un parallélisme de données avec Altivec (mode SIMD) et un parallélisme de tâches et/ou données sur plusieurs processeurs. Nous avons fait le choix d'utiliser des approches d'implantation les plus proches possibles pour les architectures matérielles SMP et MIMD-DM. Les deux approches conservent néanmoins de fortes différences structurelles. Sur l'architecture SMP, les deux processus légers travaillant sur des structures de données séparées, sont créés au sein d'une même application. Pour l'architecture MIMD-DM, nous utilisons le paradigme SPMD (Single Program Multiple Data), où sont exécutées autant d'instances de l'application que de processeurs.

Le schéma d'implantation des étapes *step_1* à *step_4* en mode SMP et MIMD-DM est essentiellement issu de la distribution des données à traiter (images, puis listes de points) entre les p processeurs disponibles. Le partitionnement des données, avec un recouvrement des images produites lors de *step_1* et *step_2*, permet de minimiser les communications entre processeurs.

Les étapes *step_1* et *step_2* sont effectuées par chaque processeur p sur un bloc de l'image i correspondant à la zone de détection, entouré d'une marge $2T+k$, nécessaire au suivi. Chacun de ces blocs a une taille $(\frac{q}{p}+2T+k, r+2T+k)$(voir section 5.2.3). Le coefficient k dépend de la taille du motif de corrélation et du sous-échantillonnage. Chacun de ces blocs peut contenir une ou plusieurs bandes

de détection.

La phase *step_3* est effectuée par chaque processeur p sur une zone de détection de taille ($\frac{q}{p}$, r), et chaque processeur produit une liste de $\frac{n}{p}$ primitives qui seront suivies lors de *step_4*. Le nombre de bandes verticales est fonction du nombre de points que chaque processeur doit détecter, à raison de 1 bande pour 3 points. Cette repartition est faite aussi équitablement que possible entre les processeurs, avec au plus une différence de 1 point entre le processeur le plus chargé et celui le moins chargé. Par exemple pour $n = 84$, avec 8 processeurs, 4 processeurs détectent 11 points et 4 autres détectent 10 points. Avec 28 processeurs, ils ont tous exactement 3 points à traiter. Il est donc possible d'obtenir une répartition spatiale différente des points détectés en fonction du nombre de points et de processeurs. Pour reprendre l'exemple, avec $n = 84$ points, pour $p = 1$ ou $p = 28$ on obtient 28 bandes de 3 points chacune. Cependant, pour $p = 8$, chaque processeur traite un bloc contenant 4 bandes, ce qui fait au total 32 bandes.

L'étape de suivi (*step_4*) est l'étape la plus coûteuse en temps de traitement. Elle permet le cumul de deux niveaux de parallélisme dans son implémentation parallèle. D'une part en mode MIMD-DM ou SMP, chaque processeur cherche les $\frac{n}{p}$ motifs qu'il a précédemment détecté. D'autre part, les résultats de recherche de maximum de corrélation sont obtenus en utilisant les instructions SIMD Altivec. De cette manière, chaque processeur est capable de traiter jusqu'à 16 pixels en une seule opération, ce qui réduit d'autant le nombre d'opérations. Cette fonction de corrélation existe en deux versions différentes.

Une fonction plus simple, de type SAD, est appliquée lors des deux premiers étages de la recherche multi résolution, travaillant sur des nombres de type entier. Sur le dernier étage, pour obtenir une précision sous-pixelique, une fonction de type SSD est appliquée, travaillant sur des nombres à virgule flottante. Les dimensions des primitives recherchées sont choisies de manière à permettre d'optimiser le nombre de traitements SIMD et les accès aux données. Ainsi, puisque les pixels peuvent être lus en mémoire par blocs de 16, des motifs de taille multiple de 16 sont échantillonnés dans l'image i puis recherchés dans l'image $i+1$. La SAD est calculée en ôtant les minimums de luminance des maximums, sur des nombre entiers 8 bits non signés. Ceci permet d'obtenir un parallélisme de 16, sans la perte de précision à cause du bit de signe qui serait engendrée par une soustraction directe entre les valeurs. La fonction SSD est effectuée sur des nombres de type flottant 32 bits, permettant un parallélisme théorique de 4. La complexité de cette opération est due principalement aux nombreuses opérations de conversion de type (vecteur de nombres entiers vers des vecteurs de nombres flottants), et aux interpolations bilinéaires, plus qu'à l'opération de SSD proprement dite.

L'étape de fusion des résultats (merge) n'existe que dans la version parallèle :

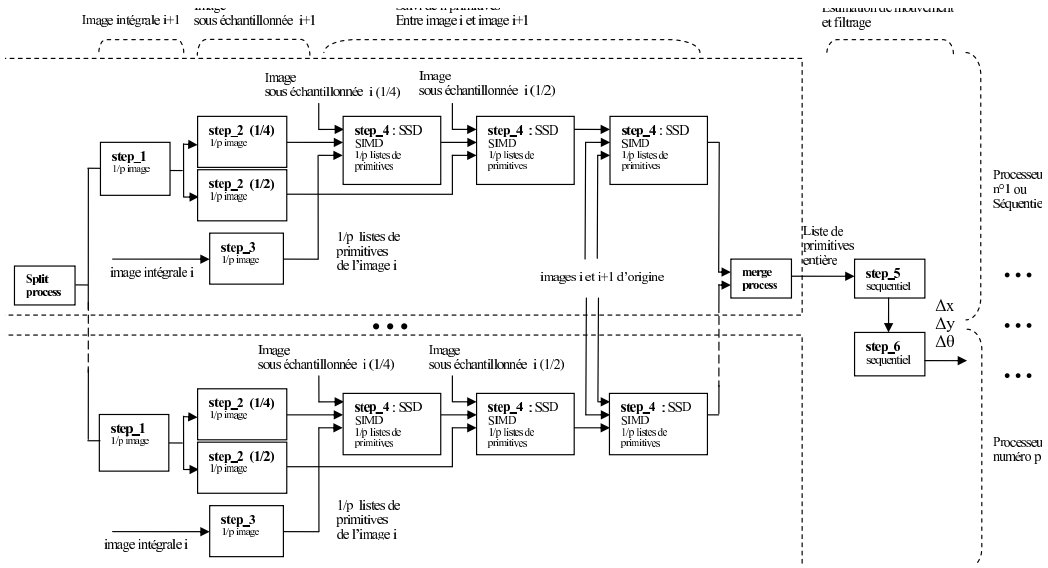


FIGURE 5.8 – Schéma de parallélisation de l'algorithme.

après la phase de suivi, toutes les listes d'appariement de points obtenues par chaque processeur sont concaténées en une seule liste. En mode MIMD-DM, ceci implique la collecte des résultats par un seul processeur (*mpi_gather*). L'essentiel des communications entre les processeurs intervient pendant cette opération. En mode SMP, les deux processus sont aussi joints dans cette étape.

Finalement, les étapes d'estimation du mouvement et filtrage (*step_5* et *step_6*), implantées en séquentiel, permettent d'achever le traitement et réaliser la stabilisation. Un schéma de la parallélisation de l'application sur plusieurs processeurs est montré en figure 5.8.

5.5.2.2 Implantation sur SoPC

Nous avons porté l'algorithme parallélisé sur notre architecture. Une architecture embarquée est bien plus limitée qu'une station de travail en terme de quantité de mémoire disponible pour stocker l'application et les données. Nous avons par conséquent effectué des choix d'implémentation différents de ceux effectués pour l'architecture BABYLON et pour l'architecture SoPC. Dans la première implémentation de l'algorithme de stabilisation, un certain nombre de techniques ont été utilisée afin de réduire les quantités de traitements ou pour augmenter la précision. Quelque unes de ces techniques n'ont pas été employées dans la version SoPC dans le but de réduire la quantité de mémoire totale requise par l'application :

- Le détecteur à ondelettes de Harr est calculé en employant une image transformée (image intégrale), qui est calculée avant la phase de détection. La mémorisation de cette image intégrale nécessite une très importante quantité de mémoire : 4 $\times$ la quantité de mémoire requise par l'image 255 niveau de gris stockée dans la mémoire de chaque processeur étant donnée que chaque pixel intégré doit être stocké en utilisant un mot de 32 bits. Dans la version SoPC il est possible de calculer la réponse du détecteur directement à partir des données de l'image d'entrée, évitant ainsi d'avoir à stocker l'image intégrale.

- La recherche de motifs correspondants entre deux images est initialement effectuée en utilisant une analyse multi-résolution utilisant 3 échelles. Il est ainsi nécessaire de stocker les image en demi et en quart de précision en plus de l'image d'entrée. Dans la version SoPC est utilisée une stratégie de recherche directe. La précision ce cette étape est par conséquent dégradée à une précision au pixel.

- Pour finir, au lieu de stocker une image i et une image i-1 (en utilisant une technique de tampons circulaires permettant d'éviter les copies de données), seuls les motifs sont mémorisés entre deux itérations de l'algorithmes, ce qui rajoute une étape de mémorisation du motif à l'intérieur de l'étape de recherche de motifs correspondants 5.9.

5.6 Résultats

5.6.1 Implantation sur Babylone

Les performances temporelles de la stabilisation ont été évaluées avec 2 séries de mesures. La première série (tableaux 5.4 et 5.5) a été effectuée sur les trois configurations de machine décrites précédemment (tableau 5.3) : [*arch_1*] de type compatible PC, et [*arch_2*] et [*arch_3*] de type POWERPC. Dans cette série, le traitement est appliqué à une séquence d'images enregistrée, format 320x240, avec un nombre réduit de primitives recherchées ($n = 24$, $T = 15$, paramètres *bench_1*). La deuxième série de mesures (tableaux 5.6 à 5.13) a été effectuée sur [*arch_3*], montrant les temps de traitement de chaque étape de l'algorithme en fonction du nombre de processeurs utilisés. Chaque paire de tableaux correspond à un format d'image, allant de 640x480 (*bench_2*, tableaux 5.6 et 5.7) à 5120x3840 (*bench_5*, tableaux 5.12 et 5.13). L'algorithme est paramétré pour rechercher un nombre plus important de primitives ($n = 84$), à une distance T proche de 5% de la largeur des

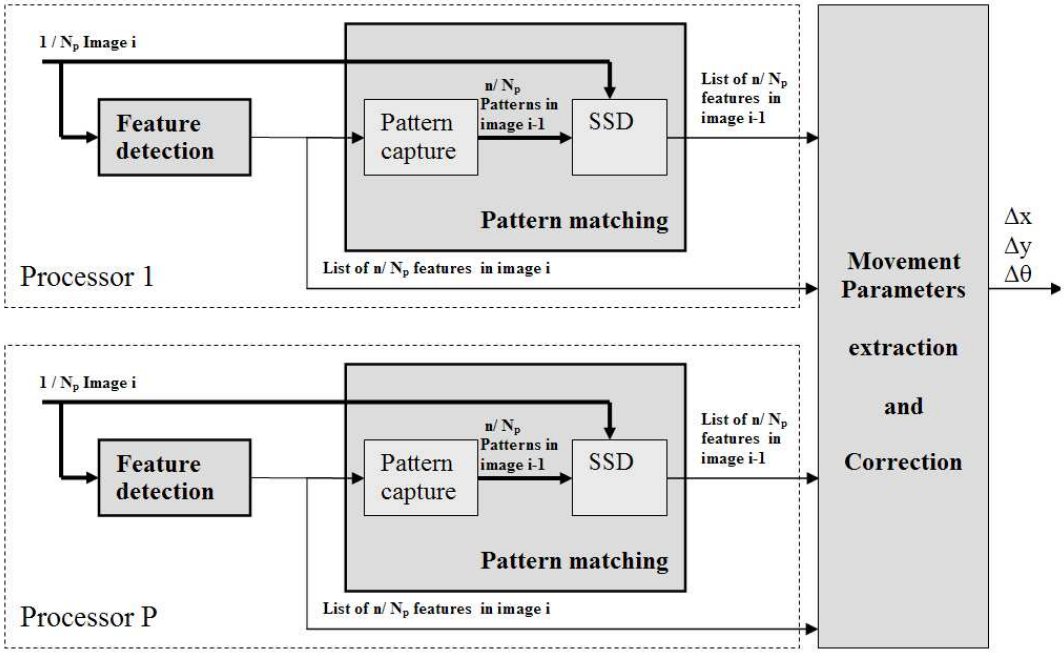


FIGURE 5.9 – Schéma d'implantation parallèle sur SoPC.

images (de $T = 30$ pixels pour *bench_2* à $T = 240$ pixels pour *bench_5*). Dans les tableaux 5.4 à 5.13 sont indiqués :

- Les temps d'exécution des étapes séquentielles et parallélisées ;
- Le temps d'exécution d'une boucle de stabilisation complète (de *step_1* à *step_6*) ;
- Pour l'implémentation SMP, les temps de création et d'arrêt des processus légers ;
- Pour l'implémentation MPI, les temps de communications entre processeurs.

Notre cluster ne disposant pas d'un bus vidéo de bande passante suffisante, les vidéos sur lesquels l'algorithme de stabilisation est appliqué proviennent des mémoires de masse reparties sur chaque noeud. Les temps d'acquisition et d'affichage ne sont donc pas comptabilisés. Les mesures correspondent au temps écoulé entre deux appels de fonctions UNIX *get_time()*, moyennées sur 1000 itérations de l'algorithme. A côté du temps d'exécution de chaque étape est indiquée entre

TABLE 5.3 – Caractéristiques des systèmes utilisés.

Plateforme	**[*Arch_1*] AthlonXP 1700+**	**[*Arch_2*] PowerMac G4**	**[*Arch_3*] PowerMac G5**
Système	Windows XP	Mac OS 10.3	Mac OS 10.3
Compilateur	gcc 3.2	gcc 3.3	gcc 3.3 + lam MPI 7.1.1
Nombre de μP	1	2	de 2 à 28
μProcesseur	Athlon XP	MPC7455	PowerPC970
Fréquence	1,47 GHz	1 GHz	2 GHz
Mémoire vive	512 MB	512 MB	1 GB
Cache L1 **L2** **L3**	128 KB 256 KB	64 KB 256 KB 1 MB	64 KB 512 KB

parenthèse l'accélération obtenue, d'une part entre la version séquentielle et l'implantation SIMD, d'autre part entre l'implantation SMP ou MPI et l'implantation SIMD.

TABLE 5.4 – Performances temporelles de [*arch_1*] et [*arch_2*] - images 320x240 et paramètres *bench_1* ($n = 24$ primitives, $T = 15$ pixels).

Etape	[*arch_1*] séquentiel	[*arch_2*] séquentiel	[*arch_2*] SIMD	[*arch_2*] SIMD + SMP ($2\mu P$)	[*arch_2*] SIMD + MPI ($2\mu P$)
step_1 et *step_2* (ms)	1,1	1,5	1,5	1,3 (1,2)	1,5 (1)
step_3 (ms)	0,3	0,5	0,5	0,4	0,3
step_4 (ms)	18,8	32,7	2,8 (11,4)	1,5 (1,9)	1,4 (2)
step_5 et *step_6* (ms)	1,4	1,8	1,8	1,8	1,8
Comm./thread overhead (ms)	-	-	-	0,9	0,7
Total (*step_1* à *step_6*) (ms)	21,1	36,5	6,6 (5,5)	5,8 (1,1)	5,6 (1,2)
Accélération Totale				**6,3**	**6,5**

5.6.1.1 Implantation SIMD

Il est difficile de déterminer une accélération théorique à partir de l'implantation de *step_4*, les calculs de corrélation (SAD et SSD) étant effectués aussi bien sur des nombres entiers 8 bits (accélération théorique de 16) que sur des nombres flottants (accélération théorique de 4). De plus, le modèle d'exécution des instructions utilisées n'est pas seulement de type SIMD, mais aussi de type DSP, comme expliqué dans la section 5.4.0.1 et illustré dans la figure 5.7.

TABLE 5.5 – Performances temporelles de [*arch_3*] - images 320x240 et paramètres *bench_1* ($n = 24$ primitives, $T = 15$ pixels).

Etape	[*arch_3*] séquentiel	[*arch_3*] SIMD	[*arch_3*] SIMD + SMP ($2\mu P$)	[*arch_3*] SIMD + MPI ($2\mu P$)
step_1 et *step_2* (ms)	0,7	0,7	0,5 (1,4)	0,5 (1,4)
step_3 (ms)	0,3	0,3	0,2	0,1
step_4 (ms)	18,5	1,5 (12,3)	0,8 (1,9)	0,7 (2,1)
step_5 et *step_6* (ms)	0,8	0,8	0,8	0,8
Comm./thread overhead (ms)	-	-	0,6	0
Total (*step_1* à *step_6*) (ms)	20,3	3,3 (6,2)	3,0 (1,1)	2,2 (1,5)
Accélération Totale			**6,8**	**9,2**

TABLE 5.6 – Performances temporelles de [*arch_3*] - images 640x480 et paramètres *bench_2* ($n = 84$, $T = 30$).

Etape	séquentiel	SIMD	SIMD + SMP ($2\mu P$)	SIMD + MPI ($2\mu P$)
step_1 et *step_2* (ms)	2,7	2,7	1,7 (1,6)	1,7 (1,6)
step_3 (ms)	0,4	0,4	0,3	0,2
step_4 (ms)	78,2	7,1 (11,1)	3,7 (1,9)	3,6 (2,0)
step_5 et *step_6* (ms)	1,1	1,1	1,1	1,1
Comm./thread overhead (ms)	-	-	0,9	0,1
Total (*step_1* à *step_6*) (ms)	82,5	11,4 (7,2)	7,6 (1,5)	6,7 (1,7)
Accélération Totale			**10,9**	**12,3**

Les instructions SIMD de calcul sur des nombres flottants sont plus efficaces que leur équivalent scalaire. Dans notre implantation, la proportion du nombre d'opérations sur des flottants par rapport au nombre d'opérations sur des entiers diminue avec l'augmentation de la fenêtre de recherche, ce qui fait varier l'efficacité de notre implantation avec T. On observe en effet pour la plateforme [*arch_3*] une accélération allant de 12,3 sur *bench_1* ($T = 15$) à 8,1 sur *bench_5* ($T = 240$). Néanmoins, du point de vue général, on observe une accélération très satisfaisante : pour [*arch_3*] et *bench_3*, l'accélération de *step_4* est de 9,3 et le temps de boucle est inférieur à 30 ms.

TABLE 5.7 – Performances temporelles de [*arch_3*] - images 640x480 et paramètres *bench_2* ($n = 84$, $T = 30$).

Etape	SIMD + MPI (4μP)	SIMD + MPI (8μP)	SIMD + MPI (16μP)	SIMD + MPI (28μP)
step_1 et *step_2* (ms)	1,2 (2,3)	1,0 (2,7)	0,8 (3,4)	0,8 (3,4)
step_3 (ms)	0,1	0,1	0,1	0
step_4 (ms)	1,8 (3,9)	1,0 (7,1)	0,6 (11,8)	0,4 (17,8)
step_5 et *step_6* (ms)	1,1	1,1	1,1	1,1
Comm./ thread overhead (ms)	0,6	0,9	1,4	2
Total (*step_1* à *step_6*)(ms)	4,8 (2,3)	4,0 (2,9)	3,9 (2,9)	4,2 (2,7)
Accélération Totale	**17,2**	**20,6**	**21,2**	**19,6**

TABLE 5.8 – Performances temporelles de [*arch_3*] - images 1280x960 et paramètres *bench_3* ($n = 84$, $T = 60$).

Etape	séquentiel	SIMD	SIMD + SMP (2μP)	SIMD + MPI (2μP)
step_1 et *step_2*(ms)	9	9	5,9 (1,5)	5,7 (1,6)
step_3(ms)	0,6	0,6	0,4	0,4
step_4(ms)	137,6	14,8 (9,3)	7,5 (2,0)	7,5 (2,0)
step_5 et *step_6*(ms)	1,1	1,1	1,1	1,1
Comm./ thread overhead (ms)	-	-	0,9	0
Total (*step_1* à *step_6*)(ms)	148,4	25,6 (5,8)	15,7 (1,6)	14,7 (1,7)
Accélération Totale			**9,5**	**10,1**

5.6.1.2 Implantation SMP

L'accélération due au SMP est proche de la valeur attendue pour *step_1*, *step_2*, *step_3* et *step_4* . Pour *bench_1*, l'accélération SIMD + SMP est de seulement 1.1, car les coûts de création et suppression des processus sont peu compensés par la diminution des temps de traitements. Les résultats sont meilleurs sur les autres séries de mesures, avec des formats d'image plus importants et un plus grand nombre de primitives détectées et suivies. Les accélérations dues au mode SMP varient entre 1,5 pour *bench_2* et 1,8 pour *bench_5*. Du point de vue de l'application, l'implantation SIMD+SMP permet à [*arch_3*] de diminuer le temps d'exécution sur *bench_3* (images 1280x960) en dessous de 20 ms, avec une accélération globale de 9,5.

TABLE 5.9 – Performances temporelles de [*arch_3*] - images 1280x960 et paramètres *bench_3* ($n = 84$, $T = 60$).

Etape	SIMD + MPI (4μP)	SIMD + MPI (8μP)	SIMD + MPI (16μP)	SIMD + MPI (28μP)
step_1 et *step_2*(ms)	4,0 (2,3)	3,0(3,0)	2,5 (3,6)	2,3 (3,9)
step_3(ms)	0,2	0,1	0,1	0
step_4(ms)	3,8 (3,9)	2,0 (7,4)	1,2 (12,3)	0,7 (21,1)
step_5 et *step_6*(ms)	1	1,1	1,1	1
Comm./ thread overhead (ms)	0,7	0,9	1,4	2,2
Total (*step_1* à *step_6*)(ms)	9,7 (2,6)	7,1 (3,6)	6,2 (4,1)	6,1 (4,2)
Accélération Totale	**15,3**	**20,9**	**23,9**	**24.3**

TABLE 5.10 – Performances temporelles de [*arch_3*] - images 2560x1920 et paramètres *bench_4* ($n = 84$, $T = 120$).

Etape	séquentiel	SIMD	SIMD + SMP (2μP)	SIMD + MPI (2μP)
step_1 et *step_2*(ms)	34,2	32,7	20,4 (1,6)	20,2 (1,6)
step_3(ms)	1,4	1,4	0,6	0,4
step_4(ms)	357,1	42,5 (8,4)	21,6 (2,0)	21,8 (2,0)
step_5 et *step_6*(ms)	1,1	1,1	1,1	1,1
Comm./ thread overhead(ms)	-	-	0,9	0,1
Total (*step_1* à *step_6*)(ms)	393,8	77,7 (5,1)	44,6 (1,7)	43,8 (1,8)
Accélération Totale			**8,8**	**9**

5.6.1.3 Implantation MPI

Les performances de l'implantation MPI avec $p = 2$ processeurs sont particulièrement proches de celles de l'implantation SMP, utilisant la même approche de parallélisation. Cependant, les temps de communications de l'implantation MPI sont inférieurs au temps de création et destruction des processus légers dans l'implantation SMP. L'accélération des étapes *step_1* et *step_2* augmente de plus en plus faiblement avec le nombre de processeur p. Ceci s'explique par la présence du coefficient $2T + k$, indépendant de p, dans l'expression de la taille des blocs à traiter ($\frac{q}{p} + 2T + k$, $r + 2T + k$). D'autre part, les temps supplémentaires de synchronisation et de communication augmentent avec le nombre de processeurs p. Pour *bench_4* et *bench_5* (tableaux 5.10 à 5.13), indépendamment du nombre de processeurs, les temps de traitement restent suffisamment élevés pour compenser les coûts de communications. Par contre, pour *bench_2* et *bench_3*, avec des dimen-

TABLE 5.11 – Performances temporelles de [*arch_3*] - images 2560x1920 et paramètres *bench_4* ($n = 84$, $T = 120$).

Etape	SIMD + MPI (4μP)	SIMD + MPI (8μP)	SIMD + MPI (16μP)	SIMD + MPI (28μP)
step_1 et *step_2*(ms)	12,9 (2,5)	9,4 (3,5)	7,7 (4,3)	6,8 (4,8)
step_3(ms)	0,2	0,2	0,1	0,1
step_4(ms)	11,1 (3,8)	5,9 (7,2)	3,5 (12,1)	2,0 (21,3)
step_5 et *step_6*(ms)	1,1	1,1	1,1	1,1
Comm./ thread overhead (ms)	0,7	1	1,7	2,2
Total (*step_1* à *step_6*)(ms)	25,9 (3,0)	17,3 (4,5)	13,8 (5,6)	11,9 (6,5)
Accélération Totale	**15,2**	**22,8**	**28,5**	**33,1**

TABLE 5.12 – Performances temporelles de [*arch_3*] - images 5120x3840 et paramètres *bench_5* ($n = 84$, $T = 240$).

Etape	sequentiel	SIMD	SIMD + SMP (2μP)	SIMD + MPI (2μP)
step_1 et *step_2*(ms)	124	123,9	74,8 (1,7)	72,0 (1,7)
step_3(ms)	1,9	1,9	0,6	0,5
step_4(ms)	1226	152,3 (8,1)	77,9 (2,0)	80,0 (1,9)
step_5 et *step_6*(ms)	1,1	1,1	1,1	1,1
Comm./ thread overhead (ms)	-	-	0,9	0,4
Total (*step_1* à *step_6*)(ms)	1353,1	279,2 (4,8)	155,2 (1,8)	154,0 (1,8)
Accélération Totale			**8,7**	**8,8**

sions d'image moins importantes, les temps de communication à 28 processeurs (environ 2 ms) deviennent trop importants pour être compensés par l'accélération des temps de traitement. Pour finir, l'accélération de *step_4* due à l'implantation MPI augmente de manière quasi-linéaire avec p.

5.6.2 Implantation sur SoPC

Les résultats d'implémentation sur la machine Babylon on permet de vérifier que les besoins en communications de l'algorithme de stabilisation parallélisé sont très faibles relativement aux besoins en puissance de traitement. Parmi les modes de communications disponibles, présentés dans le chapitre précédent, il a donc été choisi d'utiliser le mode de communication le plus simple parmi ceux que nous avons mis en place. Nous utilisons donc les connections point à point par FIFO de manière à pouvoir placer un plus grand nombre de processeurs. Nous avons

TABLE 5.13 – Performances temporelles de [*arch_3*] - images 5120x3840 et paramètres *bench_5* ($n = 84$, $T = 240$).

Etape	SIMD + MPI (4μP)	SIMD + MPI (8μP)	SIMD + MPI (16μP)	SIMD + MPI (28μP)
step_1 et *step_2*(ms)	44,9 (2,8)	32,4 (3,8)	24,7 (5,1)	21,4 (5,8)
step_3(ms)	0,3	0,2	0,1	0,1
step_4(ms)	39,7 (3,8)	21,7 (7,0)	11,9 (12,8)	6,2 (24,6)
step_5 et *step_6*(ms)	1,1	1,1	1,1	1,1
Comm./ thread overhead (ms)	0,8	0,9	2	4,5
Total (*step_1* à *step_6*)(ms)	87,1 (3,2)	56,0 (5,0)	39,7 (7,0)	32,9 (8,5)
Accélération Totale	**15,5**	**24,2**	**34,1**	**41,1**

généré des architectures basées sur un type de communication FSL (FIFO 64×32 bits), avec une configuration de processeur simple et une mémoire locale a été configurée au maximum disponible relativement à la capacité du composant et au nombre de processeur utilisé.

Si aucune unité arithmétique optionnelle n'est utilisée, chaque processeur possède cependant une unité de multiplication employant les cellules de multiplieurs spécifiques disponible dans le SoPC Xilinx, ce qui augmente considérablement (d'environ un facteur 3) les performances de la mise en correspondance de motifs.

En faisant varier le nombre de processeurs. Les résultats donnés dans le tableau 5.14 montrent qu'une solution à 32 processeurs peut se placer facilement dans un Virtex-4 LX200. On peut remarquer aussi que ce type de lien profite nettement des optimisations de l'architecture Virtex 5 pour implémenter des FIFOs, le cout total du système étant pratiquement réduit à celui des processeurs et de la mémoire comme le montre le tableau 5.15.

TABLE 5.14 – ressources SoPC consommées pour 4, 8, 16, et 32 processeurs avec des connections point à point FSL sur Virtex4.

Cible	Nb μP	SLICE	FF	LUTs	BRAM
XC4VLX200	4	5%	1%	4%	9%
XC4VLX200	8	11%	2%	10%	19%
XC4VLX200	16	26%	4%	24%	38%
XC4VLX200	32	60%	7%	55%	76%

Pour valider le flot complet de notre approche, nous avons utilisé une plate-

TABLE 5.15 – ressources SoPC consommées pour 4, 8, 16 et 32 processeurs avec des connections point à point FSL sur Virtex5.

Cible	Nb μP	SLICE	FF	LUTs	BRAM
XC5VLX330	4	3%	2%	3%	6%
XC5VLX330	8	6%	5%	6%	11%
XC5VLX330	16	12%	10%	14%	22%
XC5VLX330	32	23%	17%	28%	44%

forme de test intégrant un SoPC Virtex 4 LX60 [12].

Les résultats pour 1 à 16 processeurs proviennent de la mesure directe sur carte. Les systèmes 32 et 64 processeurs ne pouvant pas être contenus dans notre plateforme de test, les temps d'exécution au delà de 16 processeurs on été obtenus à partir de données issues de simulation. Les temps de traitements donnés dans le tableau 3 permettent ainsi d'évaluer les accélérations obtenues entre une solution matérielle et une autre en fonction du nombre de noeuds choisi et le temps d'exécution de l'application.

TABLE 5.16 – Variation des paramètres d'application parallèle en fonction du nombre de noeuds.

Nombre de noeuds	1	4	8	16	32	64
H_{PE} × W_{PE}	528 × 192	144 × 192	80 × 192	48 × 192	32 × 192	24 × 192
Facteur division Image	1	3,66	6,6	11	16,5	22
Nb de motifs par noeud	64	16	8	4	2	1

On peut voir dans le tableau 5.16 la taille des bandes images traitées localement par chaque noeud ainsi que le nombre de primitives détectées et ensuite mises en correspondance. Le tableau 5.17 montre la taille mémoire (par noeud) nécessaire à l'exécution du programme en fonction du nombre de processeurs utilisés. La quantité de mémoire interne au circuit étant répartie entre les noeuds, la quantité de mémoire disponible par noeud diminue avec le nombre de processeurs. Ceci peut mener à une taille mémoire d'application (instruction et données) trop importante pour une configuration matérielle donnée. La taille mémoire requise pour stocker les instructions, au lieu de réduire avec le nombre de processeurs, augmente, car la complexité des communications augmente avec le nombre de noeuds. Avec le découpage en bande des données, la quantité de données image

TABLE 5.17 – Comparaison de la consommation mémoire de l'application et de celle disponible par noeud (Ko) en fonction du nombre de processeur (1 à 64).

Nombre de noeuds	1	4	8	16	32	64
taille de code	3.76	4.54	4.71	5.30	5.64	5.74
taille données image	99	27	15	9	6	4.5
Total taille Application par noeud	**102.76**	**31.54**	**19.71**	**14.30**	**11.64**	**10.24**
mémoire disponible XV4LX60	256	64	32	16	8	
mémoire disponible xv4fx140 & xv5lx330	1024	256	128	64	32	16

stockée en mémoire locale diminue avec le nombre de noeuds. On peut voir dans le tableau 5.17 que des système allant jusqu'à 64 processeurs peuvent être embarqués dans les plus gros SoPC Xilinx comme le virtex4FX140 ou le virtex5LX330 qui contiennent suffisamment de mémoire interne. Sur notre plateforme de test la taille mémoire disponible est suffisante pour la repartir entre 16 noeuds.

TABLE 5.18 – Temps d'exécution (ms) de l'application pour 1 à 64 processeurs.

Nombre de processeurs	1	4	8	16	32	64
temps de Détection	217.152	55.919	27.965	13.968	6.928	3.468
temps de mise en Correspondance	31.787	7.948	3.975	1.988	0.992	0.497
temps de Communication		0.016	0.016	0.016	0.016	0.016
temps total	**248.939**	**63.883**	**31.956**	**15.972**	**7.936**	**3.981**

TABLE 5.19 – Accélération de l'application pour 1 à 64 processeurs.

Nombre ode Processeur	1	4	8	16	32	64
Détection Accélération	1.00	3.88	7.78	15.55	31.34	62.72
Mise en Corr. Accélération	1.00	4.00	8.00	15.99	32.04	63.00
Accélération globale	**1.00**	**3.90**	**7.79**	**15.60**	**31.36**	**62.53**

Les temps de traitement sont donnés à partir d'un format vidéo $H_{Tot} \times W_{Tot} = 528 \times 384$, en détectant 64 primitives visuelles et en faisant une recherche dans les 6 pixels alentours pour la mise en correspondance.

On peut voir que dans la table 5.18 qu'avec 16 processeurs et plus le temps de traitement est bien inférieur a 40ms, ce qui laisse une marge de temps satisfaisante à l'algorithme complet (incluant les parties séquentielles) de s'exécuter à 25 images/s. A 32 processeurs, les temps de traitements sont suffisamment bas pour envisager d'employer cet algorithme comme un prétraitement d'une processus de vision de plus haut niveau. On peut observer dans le tableau 5.19 que l'accélération évolue de manière quasi linéaire avec l'accroissement du nombre de processeurs. Ces résultats s'expliquent par des temps de traitements de détection et de mise en correspondance proportionnels au nombre de primitives traitées par noeud (cf tableau 5.16) et par la très faible quantité de communication entre noeuds de traitement due au choix de placement/ordonnancement des processus. Notons enfin que les résultats prennent en compte uniquement les traitements implantés dans le FPGA.

5.7 Conclusion

Dans ce chapitre est présenté l'implémentation d'une application réaliste de traitement d'image sur cible SoPC. Un réseau homogène comprenant un nombre élevé de processeurs sur SoPC se montre efficace pour ce type d'application.

Les contributions présentées dans le chapitre précédent permettent d'implémenter aisément l'architecture voulue et de tester rapidement différentes configurations de manière à sélectionner, parmi les différentes options architecturales la solution qui réponds le mieux aux besoins de l'application.

Conclusion et perspectives

La vision par ordinateur est un domaine qui nécessite la mise en oeuvre d'architectures dédiées. Les capacités d'intégration des composants microélectroniques actuels sont telles qu'il est possible de concentrer l'essentiel, voire la totalité d'un système complet, dans une même puce. Faire un *System on a Chip* est actuellement une approche de conception privilégiée. La conception de ce type de composant nécessite cependant de respecter un ensemble de contraintes sévères. Ceci amène à devoir rechercher un nouveau compromis pour chaque application.

Il est possible d'aborder le problème de conception de SoC, dans le contexte des applications de vision en tirant parti du parallélisme inhérent des algorithmes de Traitement d'Image. En abordant le problème sous cet angle, la recherche d'un compromis en terme de partitionnement logiciel-matériel classiquement menée avec une approche codesign est déplacée vers celle d'un compromis du point de vue de la parallélisation.

Dans cette thèse est ainsi proposée une méthode de conception permettant l'implantation d'algorithmes de Traitement d'Images sur systèmes multiprocesseurs monopuces dédiés à la vision. Cette méthode repose d'une part sur la parallélisation de l'algorithme de vision et d'autre part sur la conception d'un SoC multiprocesseur optimisé pour l'application.

Les contributions de cette thèse portent essentiellement sur l'élaboration d'un environnement de conception fortement automatisé de l'architecture matérielle. Nous avons adapté un environnement de conception de SoPC au flot de conception proposé, avec ajout d'un outil permettant la définition d'une architecture parallèle.

La notion de Réseau Homogène de Processeurs Communicants, a permis de formaliser notre approche d'implantation matérielle. A travers une phase d'exploration de l'espace des architectures, le concepteur procède par raffinements successifs, à la transformation d'une architecture générique de type RHPC, en une architecture optimisée pour l'application. La mise en oeuvre de l'architecture proprement dite est faite par la sélection et le paramétrage de composants matériels parmi ceux disponibles dans une bibliothèque.

Dans cette thèse ont été particulièrement développés les aspects de conception

des réseaux de communication. Nous avons mis en oeuvre divers liens physiques de communications, allant de solutions de communication très simples (connexion de type FIFO) à des solutions très élaborées (réseau de communication embarqué utilisant un routeur). Ces solutions de communications ont des caractéristiques très distinctes en termes de coût en ressources et de performances. La diversité de ces modes de communication permet au concepteur de choisir la plus adaptée aux besoins en communication de l'application.

Enfin, nous avons montré un cas pratique d'implantation d'algorithme de TI sur technologie SoPC avec l'approche RHPC à travers une application de stabilisation vidéo.

Les travaux qui seront réalisés à l'avenir dans la continuité de cette thèse sont les suivants. Nous continuerons la confrontation de la validité de l'approche par l'implantation d'autres applications de TI, présentant des caractéristiques distinctes en terme de parallélisme et/ou de communications.

Les travaux autour de la thématique des communications seront poursuivis, notamment à travers la mise en oeuvre de divers composants d'Entrée/Sortie vidéo parallèles, permettant de choisir le mode d'accès aux données images apportant des possibilités d'optimisation de l'architecture en fonction des besoins de l'application. Nous prévoyons dans la même optique d'étendre le choix à plusieurs topologies réseau à travers des évolutions de l'outil de génération d'architecture, et l'adaptation du routeur.

Nous avons vu dans le chapitre 4 qu'un mode de programmation des communications homogène était nécessaire pour pouvoir tester rapidement l'application parallèle avec chaque mode de communication. Une des possibilités pour y parvenir est l'écriture d'une bibliothèque de communication permettant de masquer les différences entre les différents liens de communications, en s'inspirant de la librairie MPI. Une autre possibilité est l'adaptation des outils d'aide à la parallélisation pour leur permettre la génération d'un code parallèle ciblant directement les architectures embarquées avec les différents modes de communication.

Enfin, il est prévu d'étendre les possibilités de choix d'optimisation au niveau IP processeur. L'extension progressive de la base de processeurs pourra se faire en intégrant à celle ci des IP de processeur libres. Il est prévu de plus d'utiliser un ADL comme LISA[70] qui permettra de compléter la base de processeur avec des architectures classiques, mais aussi de concevoir ponctuellement un processeur *taillé sur mesure* avec une approche ASIP. Une des perspectives ouvertes par cette approche est notamment l'intégration de capacités de traitement parallèle au niveau du processeur.

Bibliographie

[1] Altera - avalon interface specifications.

[2] Arm - amba.

[3] Arm sotfcore processor products.

[4] Ibm - coreconnect.

[5] Sonics - silicon backplane.

[6] Stmicroelectronics - st bus.

[7] Mpi : A message-passing interface standard, May 2008.

[8] Nicolas Allezard, Lionel Damez, and Jean Pierre Dérutin. Stabilisation d'images temps réel par mise en correspondance, parallélisation à grain fin et à grain moyen. Rapport de recherche, LASMEA - Université Blaise Pascal, Mar 2003.

[9] Altera. Site web altera.

[10] Adrijean Andriahantenaina, Hervé Charlery, Alain Greiner, Laurent Mortiez, and Cesar Albenes Zeferino. Spin : a scalable, packet switched, on-chip micro-network. In *Design Automation and Test in Europe Conference (DATE'2003)*, pages 70–73, Munchen, Germany, March 2003.

[11] ARM. Amba axi protocol v1.0 specification. Technical report, 2004.

[12] AVNET. Xilinx virtex-4 lx evaluation kit.

[13] Johnathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *IEEE Workshop on FPGAs for Custom Computing Machines*, 1999.

[14] S. Balarskirsky and R. Chelllappa. Performance characterization of image stabilization algorithms. Rapport de recherche, Center for Automation Research - University of Maryland, 1996.

[15] J. Barron, D. Fleet, S. Beauchemin, and T. Burkitt. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1) :43–77, 1994.

[16] Gordon Brebner and Delon Levi. Networking on chip with platform fpgas. In *Proceedings of 2003 IEEE International Conference on Field-Programmable Technology*, pages 13–20, 2003.

[17] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[18] C. Cameron. Using fpgas to supplement ray-tracing computations on the cray xd-1. *HPCMP Users Group Conference*, 0 :359–363, 2007.

[19] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. In *Proceedings of the 39th conference on Design automation*, pages 789–794. ACM Press, 2002.

[20] Pierre Chalimbaud and François Berry. Embedded active vision system based on an fpga architecture. *EURASIP J. Embedded Syst.*, 2007(1) :26–26, 2007.

[21] Hanen CHNINI. Coeurs de processeur software paramétrables pour l'implantation d'applications de traitement d'images sur cibles sopc. Master's thesis, LAboratoire des Sciences et Matériaux pour l'Electronique, et l'Automatique (LASMEA, UMR CNRS 6602), 2009.

[22] M. Collin, R. Haukilahti, M. Nikitovic, and J. Adomat. Socrates - a multiprocessor soc in 40 days. In *Conference on Design, Automation and Test in Europe*.

[23] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra. Spidergon : a novel on-chip communication network. In *the International Symposium on System-on-Chip*, page 115, November 2004.

[24] Coware. Processor designer.

[25] Stephen Craven and et al. Configurable soft processor arrays using the openfire processor. In *Proceedings of 2005 MAPLD International Conference*, pages 250–256, 2005.

[26] William J. Dally and Brian Towles. Route packets, not wires : On-chip interconnection networks. In *Design Automation Conference*, pages 684–689, 2001.

[27] W.J. Dally and H. Aoki. Deadlock-free adaptive routing in multicomputer networks using virtual channels. In *IEEE Transactions On Parallel And Distributed Systems*, volume 4, pages 466–475, 1993.

[28] W.J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. In *IEEE Transactions On Computer*, volume C-36, 1987.

[29] Giovanni De Micheli and Luca Benini. *Networks on Chips : Technology And Tools*. Morgan Kaufmann Publishers, systems on silicon edition, 2006.

[30] F. Dias Real de Oliveira. *Analyse, développement et qualification d'un algorithme de stabilisation électronique d'images*. Thèse de master, Université Blaise Pascal, 2004.

[31] Jean Pierre Derutin and et al. Simd, smp and mimd-dm parallel approaches for real-time 2d image stabilization. In *CAMP 2005. Computer Architecture for Machine Perception*, pages 73–80. IEEE Computer Society, 2005.

[32] J.P. Dérutin and al. A parallel vision machine : Transvision. In *Proc. of Computer Architecture for machine perception*, pages 241–251, december 1991.

[33] T Domany, Mb Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, J Gagliano, A Gara, R Garg, R Germain, Me Giampapa, B Gopalsamy, J Gunnels, B Rubin, A Ruehli, S Rus, Rk Sahoo, A Sanomiya, E Schenfeld, M Sharma, S Singh, P Song, V Srinivasan, Bd Steinmacher-burow, K Strauss, C Surovic, Tjc Ward, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae (ibm Server Group, K Dockser (ibm Microelectronics, and L Kissel. An overview of the bluegene/l supercomputer the bluegene/l team, 2002.

[34] José Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parallel Distrib. Syst.*, 4(12) :1320–1331, 1993.

[35] José Duato. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parallel Distrib. Syst.*, 6(10) :1055–1067, 1995.

[36] J. Dubois, D. Ginhac, M. Paindavoine, and B. Heyrman. A 10 000 fps cmos sensor with massively parallel image processing. *IEEE Journal of Solid-State Circuits*, 43(3) :706–717, March 2008.

[37] Z. Duric and A. Rosenfeld. Shooting a smooth video with a shaky camera. *Machine Vision and Applications*, 13(5-6) :303–313, 2003.

[38] J. Falcou and J. Serot. E.v.e., an object oriented simd library. In *International Conference and Computation Science - ICCS'2004*, pages 323–330, 2004.

[39] J. Falcou, J. Serot, T. Chateau, and J.T. Lapresté. Quaff : Efficient c++ design for parallel skeletons. *Parallel Computing*, 7 :604–615, 2006.

[40] Joël FALCOU. *Un cluster pour la Vision Temps RéelArchitecture, Outils et Applications*. PhD thesis, École DoctoraleSciences Pour l'Ingénieur de Clermont-Ferrand, 2006.

[41] M.J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960, septembre 1972.

[42] F. Gensolen, Guy Cathebras, Lionel Martin, and Michel Robert. An image sensor with global motion estimation for micro camera module. In Jacques Blanc-Talon, Wilfried Philips, Dan C. Popescu, and Paul Scheunders, editors, *ACIVS*, volume 3708 of *Lecture Notes in Computer Science*, pages 713–721. Springer, 2005.

[43] Antonio Gentile, José L. Cruz-rivera, D. Scott Wills, Leugim Bustelo, José J. Figueroa, Javier E. Fonseca-camacho, Wilfredo E. Lugo-beauchamp, Ricardo Olivieri, Marlyn Quiñones-cerpa, Alexis H. Rivera-ríos, and Michelle Viera-vera. Real-time image processing on a focal plane simd array, 1999.

[44] K. Goossens and et al. Networks on silicon : Combining best-effort and guaranteed services. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 423–425, March 2002.

[45] SILICON GRAPHICS. Sgi rasc technology-a complete fpga solution for orders-of-magnitudeperformance improvement and dramatic application speedup.

[46] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression : A language for architecture exploration through compiler/simulator retargetability. In *DATE*, 1999.

[47] L. Hammond and K. Olukotun. Considerations in the design of hydra : A multiprocessor-on-a-chip microarchitecture. Technical Report CSL-TR98 -749, Stanford University, Computer Systems Laboratory, February 1998.

[48] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceeding of the 4th Alvey Vision Conference*, pages 147–151, 1988.

[49] Stephan Hengstler and Hamid Aghajan. A smart camera mote architecture for distributed intelligent surveillance. In *In ACM SenSys Workshop on Distributed Smart Cameras (DSC*, 2006.

[50] W. Daniel Hillis. *The connection machine*. MIT Press, Cambridge, MA, USA, 1986.

[51] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17 :185–204, 1981.

[52] A. A. Jerraya, R. Bergamaschi, I. Bolsens, R. Gupta, R. Harr, K. Keutzer, K. Olukotun, and K. Vissers. Are Single-Chip Multi-processors in Reach ? *IEEE Design and Test of Computers, Volume 18 Number 1*, Jan–Feb 2001.

[53] Gaisler Jiri. *The LEON-2 Processor Users Manual*, 2003.

[54] Faraydon Karim, Anh Nguyen, Sujit Dey, and Ramesh Rao. On-chip communication architecture for oc-768 network processors. In *Design Automation Conference*, Jun 2001.

[55] R. Kleihorst, H. Broers, A. Abbo, H. Ebrahimmalek, H. Fatemi, H. Corporaal, and P. Jonker. An simd-vliw smart camera architecture for real-time face recognition. In *IN PROCEEDINGS OF PRORISC 2003*, pages 1–7, 2003.

[56] S. Kumar, A. Jantsch, J. Soininen, M. Forsell, J. Öberg M. Millberg, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings of the ISVLSI'02*, Pittsburgh, Pennsylvania, April25–26 2002.

[57] R. Mateos, J.L. Lazaro, and F. Espinosa. Hardware/software co-simulation environment for csoc with soft processors. In *Field-Programmable Technology*, pages 445–448, 2004.

[58] Raul Mateos. *Técnicas de cosimulación HW/SW para el diseño y verificación de sistemas CSoC*. PhD thesis, Universidad de Alcalá Escula Politécnica Superior Departemento de Electrónica, 2006.

[59] Samy Meftali. *Exploration d'architectures et allocation/affectation mémoire dans les systèmes multiprocesseurs monopuce*. PhD thesis, Université Joseph Fourier - Grenoble 1, 2002.

[60] C. Morimoto. *Electronic Digital Stabilization : Design and Evaluation, with Applications*. Thèse de doctorat, University of Maryland, 1997.

[61] R. Mosqueron, J. Dubois, and M. Paindavoine. High-speed smart camera with high resolution. *EURASIP J. Embedded Syst.*, 2007(1) :23–23, 2007.

[62] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[63] Pierre Niang, Thierry Grandpierre, Mohamed Akil, and Yves Sorel. Syndex-ic : un environnement logiciel pour l'implantation optimisée d'applications temps réel sur circuits reconfigurables. In *Journée francophones sur l'Adéquation Algorithme Architecture*, 2005.

[64] OCP-IP. Open core protocol specification, version 2.0, September 2003.

[65] Opencores organisation. Site web opencores.

[66] H. Pourreza, M. Rahmati, and F. Behazin. Weighted multiple bit-plane matching, a simple and efficient matching criterion for electronic digital image stabilizer application. In *6^{th} International Conference on Signal Processing*, volume 2, pages 957–960, 2002.

[67] P.A. Revenga, J. Serot, L. Lazaro, and J.P. Derutin. A beowulf-class architetcure proposal for real-time embedded vision. In *International Parallel and Distributed Processing Symposium*, 2003.

[68] Taha Ridene and Antoine Manzanera. Mécanismes d'attention visuelle sur rétine programmable. *Traitement et Analyse de l'Information : Méthodes et Applications*, pages 301–306, may 2007.

[69] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the memory wall : The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[70] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. Architecture implementation using the machine description language lisa. In *ASP-DAC '02 : Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 239, Washington, DC, USA, 2002. IEEE Computer Society.

[71] J. Sebot. Impact des extensions simd sur les performances d'applications multimedia. *Technique et Science Informatiques*, 21(2) :185–204, Mar 2002.

[72] J. Serot and D. Ginhac. Skeletons for parallel vision : an overview of the skipper project. *Parallel Computing*, (28) :1785–1808, December 2002.

[73] Tensilica. Tensilica's processor technology.

[74] D. Tsai, C. Lin, and J. Chen. The evaluation of normalized cross correlations for defect detection. *Pattern Recognition Letters*, 24 :2525–2535, 2003.

[75] A. Verri and T. Poggio. Motion field and optical flow : Qualitative properties. In *IEEE Trans. Pattern Analysis and Machine Intelligence*, volume 11, pages 490–498, 1989.

[76] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition*, 2001.

[77] VSIA. Virtual component interface standard.

[78] Scott Weber, Matthew W. Moskewicz, Manuel Loew, and Kurt Keutzer. Multi-view operation-level design – supporting the design of irregular asips. Technical Report UCB/ERL M03/12, University of California, Berkeley, April 2003.

[79] Xilinx. Site web xilinx.

[80] P. Yiannacouras, J. Rose, and J.G. Steffan. The microarchitecture of fpga-based soft processors. In *Proc. of CASES'05*, page 202212, September 2005.

[81] H. Yoshimoto, D. Arite, and R. Taniguchi. Real-time image processing on ieee1394-based pc cluster. In *15th International Parallel and Distributed Processing Symposium*, 2001.

[82] Z. Zhu, G. Xu, Y. Yang, and J. Jin. Camera stabilisation based on 2.5d motion estimation and inertial motion filtering. In *International Conference on Intelligent Vehicles*, 1998.

Résumé

La conception de prototypes de systèmes de vision en temps réel embarqué est sujet à de multiples contraintes sévères et fortement contradictoires. Dans le cas de capteurs dits "intelligents", tout ou parti des traitements sont effectué à proximité de la rétine, il est nécessaire de fournir une puissance de traitement suffisante pour exécuter les algorithmes à la cadence des capteurs d'images avec un dispositif de taille minimale et consommant peu d'énergie. La densité d'intégration des transistors permet de nos jours de concentrer l'essentiel, voire l'intégralité de l'application dans un seul composant. La conception d'un tel composant (appelé système monopuce ou SoC) et l'implantation d'algorithmes de plus en plus complexes pose problème si on veut l'associer avec une approche de prototypage rapide d'applications scientifiques.

Afin de réduire de manière significative le temps et les différents coûts de conception, le procédé de conception est fortement automatisé. La conception matérielle est basée sur la dérivation d'un modèle d'architecture multiprocesseur générique de manière à répondre aux besoins de capacité de traitement et de communication spécifiques à l'application visée. Les principales étapes manuelles se réduisent au choix et au paramétrage des différents composants matériels synthétisables disponibles. La conception logicielle consiste en la parallélisation des algorithmes, qui est facilitée par l'homogénéité et la régularité de l'architecture de traitement parallèle et la possibilité d'employer des outils d'aide à la parallélisation.

Avec l'approche de conception sont présentés les premiers éléments constitutifs qui permettent de la mettre en oeuvre. Ceux ci portent essentiellement sur les aspects de conception matérielle. Ils sont de nature diverse, outils d'aide à la conception permettant la génération automatisée d'un réseau de processeurs à mémoire distribuée, ou composants matériels (IP) pour les communications dont notamment un routeur de paquets. L'approche proposée est illustrée par l'implantation d'un traitement de stabilisation temps réel vidéo sur technologie SoPC.

Mots-clés : Architectures de vision, Système monopuce, FPGA, architecture parallèle, mémoire distribuée, passage de message, prototypage rapide.

Abstract

The conception of a real time embedded vision system imposes multiple and severe design constraints, which could be conflicting. Smart camera applications usually requires integration of sofisticated processing near the transducer, with sufficient processing power to run the algorithms at the information flow rate, and using a system of minimal size that consumes little power. Today, transistor integration density enables concentration of the main part, or even all, of a complete system on a sole component (System on a Chip - SoC).

A strongly automated design flow is proposed, it reduces the design effort and conception costs, in order to enable fast implementation of complex algorithms into a SoC. Our overall hardware implementation method is based upon meeting algorithm processing power requirement and communication needs with refinement of a generic parallel architecture model. Actual hardware implementation is done by the choice and parameterization of readily available reconfigurable hardware modules and customizable commercially available IPs. Software conception is based upon parallelisation of the algoritms. An homogeneous and regular hardware architecture model is chosen which enables to use parallelisation tools.

With the design method, most of the works presented in this thesis are focused on enabling a automated hardware design enviromnent. This includes various works, tools enabling automated generation of a homogeneous network of communicating processors, or hardware components (IP) for communication network, including a paquet router. The presented approach is illustrated with the embedding a real time image stabilization algorithm on SoPC technology.

Keywords : Vision architectures, System on a Chip, FPGA, Parallel architecture, Distributed Memory, Message Passing, rapid prototyping.