Exploration architecturale pour la conception d'un système sur puce de vision robotique, adéquation algorithme-architecture d'un système embarqué temps-réel

Thomas Lefebvre

Exploration architecturale pour la conception d'un système sur puce de vision robotique, adéquation algorithme-architecture d'un système embarqué temps-réel

Thomas Lefebvre

Laboratoire ETIS

Université de Cergy-Pontoise

Thèse présentée pour le titre de

Docteur de l'université de Cergy-Pontoise

Soutenue le Lundi 2 juillet 2012 devant la commission d'examen :

Michel Paindavoine : Rapporteur
Jocelyn Sérot : Rapporteur
Serge Weber : Examinateur
Lionel Lacassagne : Examinateur
Nicolas Cuperlier : Examinateur
Benoît Miramond : Encadrant de thèse
Lounis Kessal : Co-Directeur de thèse
François Verdier : Directeur de thèse

Résumé

La problématique de cette thèse se tient à l'interface des domaines scientifiques de l'adéquation algorithme architecture, des systèmes de vision bio-inspirée en robotique mobile et du traitement d'images. Le but est de rendre un robot autonome dans son processus de perception visuelle, en intégrant au sein du robot cette tâche cognitive habituellement déportée sur un serveur de calcul distant. Pour atteindre cet objectif, l'approche de conception employée suit un processus d'adéquation algorithme architecture, où les différentes étapes de traitement d'images sont analysées minutieusement. Les traitements d'image sont modifiés et déployés sur une architecture embarquée de façon à respecter des contraintes d'exécution temps-réel imposées par le contexte robotique.

La robotique mobile est un sujet de recherche académique qui s'appuie notamment sur des approches bio-mimétiques. La vision artificielle étudiée dans notre contexte emploie une approche bio-inspirée multirésolution, basée sur l'extraction et la mise en forme de zones caractéristiques de l'image.

Du fait de la complexité de ces traitements et des nombreuses contraintes liées à l'autonomie du robot, le déploiement de ce système de vision nécessite une démarche rigoureuse et complète d'exploration architecturale logicielle et matérielle. Ce processus d'exploration de l'espace de conception est présenté dans cette thèse.

Les résultats de cette exploration ont mené à la conception d'une architecture principalement composée d'accélérateurs matériels de traitements (IP) paramétrables et modulaires, qui sera déployée sur un circuit reconfigurable de type FPGA. Ces IP et le fonctionnement interne de chacun d'entre eux sont décrits dans le document. L'impact des paramètres architecturaux sur l'utilisation des ressources matérielles est étudié pour les traitements principaux. Le déploiement de la partie logicielle restante est présenté pour plusieurs plate-formes FPGA potentielles.

Les performances obtenues pour cette solution architecturale sont enfin présentées. Ces résultats nous permettent aujourd'hui de conclure que la solution proposée permet d'embarquer le système de vision dans des robots mobiles en respectant les contraintes temps-réel qui sont imposées.

This Ph.D Thesis stands at the crossroads of three scientific domains : algorithm-architecture adequacy, bio-inspired vision systems in mobile robotics, and image processing. The goal is to make a robot autonomous in its visual perception, by the integration to the robot of this cognitive task, usually executed on remote processing servers. To achieve this goal, the design approach follows a path of algorithm architecture adequacy, where the different image processing steps of the vision system are minutely analysed. The image processing tasks are adapted and implemented on an embedded architecture in order to respect the real-time constraints imposed by the robotic context.

Mobile robotics as an academic research topic based on bio-mimetism. The artificial vision system studied in our context uses a bio-inspired multiresolution approach, based on the extraction and formatting of interest zones of the image.

Because of the complexity of these tasks and the many constraints due to the autonomy of the robot, the implementation of this vision system requires a rigorous and complete procedure for the software and hardware architectural exploration. This processus of exploration of the design space is presented in this document.

The results of this exploration have led to the design of an architecture primarly based on parametrable and scalable dedicated hardware processing units (IPs), which will be implemented on an FPGA reconfigurable circuit. These IPs and the inner workings of each of them are described in the document. The impact of their architectural parameters on the FPGA resources is studied for the main processing units. The implementation of the software part is presented for several potential FPGA platforms.

The achieved performance for this architectural solution are finally presented. These results allow us to conclude that the proposed solution allows the vision system to be embedded in mobile robots within the imposed real-time constraints.

À ma mère, Sylvie Lefebvre.

Remerciements

Le document que vous avez entre vos mains est le résultat de plus de quatre ans de travail, et il n'aurait pas pu exister sans l'aide de nombreuses personnes.

Tout d'abord, je souhaite remercier les membres du jury qui ont jugé ce travail digne du grade de docteur : MM. Michel Paindavoine et Jocelyn Sérot, pour leur rôle de rapporteur et le travail qui y est associé ; ainsi que MM. Serge Weber, Lionel Lacassagne, et Nicolas Cuperlier, pour avoir accepté le rôle d'examinateur.

Je remercie le laboratoire ETIS représenté par sa directrice, Inbar Fijalkow, pour l'accueil et les moyens mis à disposition pour l'hébergement de ce projet.

Je remercie aussi le service informatique d'ETIS, notamment Michel Leclerc, Michel Jordan, et bien sûr Laurent Protois, ainsi que les secrétaires Annick Bertinotti et Astrid Cébron pour leur aide précieuse.

Les travaux présentés dans ce document ont été dirigés par trois personnes que je souhaite remercier : François Verdier, pour la confiance et la liberté qu'il m'a accordé sur mes travaux, Lounis Kessal, pour son aide précieuse quant à la conception d'IP, et Benoît Miramond, qui est peu à peu devenu indispensable et qui a beaucoup aidé à orienter cette thèse, notamment en encadrant de nombreux stages et projets à l'université autour de mon sujet de thèse. Je les remercie également tous les trois pour leur travail de relecture sans lequel ce document serait beaucoup moins clair et lisible.

D'autres personnes ont pu m'aider lors de ces travaux, je souhaite donc remercier Laurent Fiack, Fred Demelo, David Bailly, ainsi que les stagiaires avec lesquels j'ai pu travailler.

Un grand merci à Mohamed El Amine Benkhelifa, qui m'a énormément aidé pour mes cours à l'IUT lors de mon monitorat et de mon contrat d'ATER,

merci aussi à toute l'équipe de l'IUT GEII de Neuville, c'était sympa de passer de l'autre côté de la barrière.

Au cours des années passées à ETIS, j'ai pu voir de nombreuses personnes arriver au labo et/ou en partir, je souhaite remercier tout d'abord mes collègues de bureau : Emmanuel Huck, Abdel-Nasser Assimi, Heyckel Houas, et finalement Lotfi Bendaouia et Liang Zhou. Un grand merci aussi pour leur amitié et/ou leur sympathie à Samuel Garcia, Guy Wassi, Jean-Emmanuel Haugeard, David Gorisse, Auguste Venkiah, Julien Gony, Ayman Alsawah, Shuju Zhao, Laurent Rodriguez, Laurent Gantel, Amel Khiar, Jean-Christophe Sibel, Ludovic Danjean, Alexis Lechervy, Leila Meizou, Gaël Rigaud, Ismail Ktata, Erbao Li, Fatine Lahmani et enfin Roland-Christian Gamom Ngounou Ewo. J'en ai surement oublié, je les remercie aussi.

Merci aussi à Bertrand Granado, Emmanuelle Bourdel, Myriam Ariaudo, Philippe-Henri Gosselin, David Picard, et Fakhreddine Ghaffari, des permanents particulièrement sympas.

Enfin, si je suis content d'avoir fait cette thèse, c'est aussi et surtout car j'ai pu rencontrer Sonja Khatchadourian. Merci Sonja pour ton soutien et tes conseils lors de la fin de ma thèse. Merci aussi à Monique Khatchadourian et Sonja pour la préparation du délicieux pot de thèse.

Merci à mon père et à ma sœur, Bruno et Karine Lefebvre, qui ont pu assister à ma soutenance et même en comprendre les grandes lignes !

Merci à ma mère, qui voulait assister à ma soutenance, et dont la mémoire a été ma motivation principale pour tenir jusqu'au bout.

Table des matières

Table des figures

Liste des tableaux

Introduction

Cadre du projet

Le projet présenté dans ce document est hébergé par le laboratoire ETIS (Équipe de Traitement de l'Information et Systèmes). Il est le fruit de la collaboration entre deux entités du laboratoire : l'équipe de robotique Neurocybernétique, et le groupe thématique Architectures de l'équipe ASTRE.

L'équipe Neurocybernétique étudie l'intelligence artificielle bio-inspirée à travers des applications robotiques. Plusieurs robots sont conçus pour répondre à des problématiques différentes, le principal sujet étudié étant la navigation robotique.

Les robots étudiés perçoivent leur environnement visuel par des caractéristiques locales, fournies par un système de vision bio-inspiré qui analyse le flux d'images d'une caméra fixée sur le robot. Ce système de vision est le sujet des travaux présentés dans ce document. La vision artificielle bio-inspirée présente de nombreux avantages, mais son inconvénient principal est un besoin élevé en ressources de calcul. Ainsi, au début de ce projet, le système de vision robotique existe dans une version logicielle simplifiée (monorésolution), déporté sur des stations de travail externes.

L'équipe ASTRE étudie les aspects technologiques des circuits électroniques, et le groupe thématique Architectures travaille entre autres sur le développement d'IP (unités fonctionnelles électroniques) de traitement d'image. Le but du projet est d'embarquer le système de vision au sein des robots mobiles, afin de permettre à ces derniers de gagner en autonomie.

Un système embarqué implique un certain nombre de contraintes : une alimentation sur batterie et un espace restreint limitent les choix de plate-formes de calcul pouvant être utilisées pour le système de vision. Afin de pouvoir utiliser des techniques bio-inspirées complexes dans un robot autonome, il est nécessaire de respecter ces contraintes, tout en proposant une solution suffisamment performante. L'application doit en effet avoir un comportement temps-réel afin de ne pas ralentir le fonctionnement des robots. Dans le cas d'une application de vision bio-inspirée, une cadence de 25 images par seconde est assez courante.

Problématique

L'équipe Neurocybernétique dispose pour ses robots d'un système de vision bio-inspiré sous forme logicielle, déporté sur des stations de travail et limité à une seule bande de fréquence spatiale. Une version de ce système faisant appel à l'approche multirésolution, se rapprochant encore plus d'un comportement biologique, a été conçue, mais la charge de calcul nécessaire à cette version la rend inutilisable en pratique. L'objectif des travaux de recherche présentés dans ce document est l'intégration de ce système de vision bio-inspiré multirésolution au sein du robot lui-même.

Les différents types de robots ciblés diffèrent sur plusieurs points : certains sont mobiles tandis que d'autres non, et divers types de missions sont menées par les robots, de la navigation à la reconnaissance d'expressions faciales. Les besoins variés de ces robots impliquent un ensemble de contraintes auxquelles il est impératif de se plier. Des limites en volume et en énergie électrique sont dictées par le besoin d'autonomie de robots mobiles. Un système paramétrable permettra d'obtenir un comportement adapté aux différentes missions des robots. La performance est un autre aspect important, la résolution et la cadence des images traitées ont un impact majeur sur la perception visuelle des robots. Le but est donc de concevoir un système embarqué paramétrable, travaillant sur des vidéos à haute résolution à une cadence élevée.

Les technologies envisageables pour réaliser ce système sont fortement limitées par ces diverses contraintes. L'énergie électrique et le volume disponibles sur le robot étant

relativement faibles, il est par exemple assez difficile d'utiliser un GPU puissant. Une étape importante sera donc le choix du type de circuit sur lequel déployer le système de vision.

Il est ensuite nécessaire de déployer les fonctions du système de vision en cohérence avec cette technologie. L'adéquation algorithme / architecture est le principal aspect du travail à mettre en œuvre pour atteindre un système cohérent. La diversité des besoins des robots ciblés impose une flexibilité du système, ainsi la conception du système dans son ensemble doit permettre une modification aisée des paramètres.

L'aspect architectural est le cœur du projet présenté dans ce document : l'application est définie et existe au format logiciel, le robot et son réseau de neurones sont du ressort de l'équipe de robotique. Le défi est de proposer une plate-forme hébergeant l'application qui permette d'utiliser cette dernière de façon autonome, embarquée sur un robot, en respectant les contraintes temps-réel.

Structure du document

Le document est divisé en cinq chapitres, organisés de la façon suivante :

Le premier chapitre présente le projet doctoral. Le contexte du projet et l'application de vision sont exposés. Le flot de conception emprunté pour la réalisation du système de vision embarqué est ensuite détaillé.

Le deuxième chapitre aborde l'état de l'art sur les thématiques couvertes par ce projet, de la vision artificielle aux circuits dédiés de ce domaine.

Le troisième chapitre expose l'exploration de plusieurs solutions logicielles. Les algorithmes utilisés dans l'application de vision sont tout d'abord présentés, et les profils dressés à partir de plusieurs versions logicielles de l'application sont présentées. Ces versions se différencient par les plate-formes utilisées, ainsi ce chapitre traite de l'adéquation entre des plate-formes à base de processeurs et les différentes parties du système

de vision.

Le quatrième chapitre présente des accélérateurs matériels conçus pour déployer le système de vision dans le respect des contraintes temporelles. L'organisation globale de ces IP est tout d'abord décrite, puis les architectures de chacune d'entre elles sont détaillées.

Le cinquième chapitre propose un circuit répondant aux contraintes du robot, en s'appuyant sur les résultats des deux chapitres précédents. La validation des parties matérielles du circuit est abordée. Les ressources nécessaires et les performances de ce circuit sont analysées, ainsi que leur sensibilité aux paramètres principaux du système.

Chapitre 1

Présentation du projet

1.1 Contexte

Afin de mieux appréhender le travail présenté dans ce document, il convient de situer le projet dans son contexte. Les équipes du laboratoire ETIS liées à ce projet sont présentées, puis le système à réaliser au sein de son hôte. Les contraintes temps-réel non-strictes et les besoins propres de la robotique mobile sont enfin présentées, afin de mieux cerner les limites imposées pour la conception du système.

1.1.1 Équipes Neurocybernétique et ASTRE

Le projet doctoral présenté dans ce document s'inscrit dans le cadre d'une collaboration entre deux équipes du laboratoire ETIS.

L'équipe Neurocybernétique étudie la robotique bio-inspirée, c'est à dire la modélisation de fonctionnements du cerveau des mammifères pour le contrôle robotique. Leurs études de ce champ de recherche les amènent à valider leurs travaux théoriques par des expérimentations pratiques. À cette fin, l'équipe utilise des robots, mobiles ou non, afin d'explorer, raffiner, et valider des modèles de systèmes neuronaux.

Le simulateur de réseaux de neurones PROMETHE et l'éditeur de réseaux de neurones LETO sont des projets de longue haleine au cœur de l'équipe Neurocybernétique (Rev97). L'application présentée ici, conçue pour la robotique mobile, a pour but de

fournir aux réseaux de neurones de PROMETHE des informations sur leur environnement visuel dans un format pertinent (Mai07).

L'équipe ASTRE, quant à elle, s'emploie à l'étude des systèmes reconfigurables, principalement de type RSoC/RSiP. Cette équipe regroupe trois thématiques :

- Systèmes : études visant à abstraire le matériel, principalement par la modélisation de systèmes.
- Architectures : déploiement d'IP pour le traitement du signal et des images, et définition de nouvelles architectures reconfigurables.
- Technologie : étude des technologies émergentes et de leurs apports pour concevoir des réseaux sur puce (NoC) innovants et performants.

Le projet présenté ici s'inscrit dans la thématique Architectures de cette équipe : la conception du système de vision embarqué passera par le déploiement d'IP de traitement d'images sur systèmes reconfigurables. Une partie de l'exploration architecturale présentée dans ce document a été réalisée en collaboration avec la thématique Systèmes de l'équipe ASTRE (section 3.2.3).

1.1.2 Présentation du système de vision

Le contexte de cette application, du coté du traitement d'image, permet de mieux appréhender l'utilité des différentes parties de l'application. L'application de vision étudiée a pour but de permettre à son hôte de se repérer dans son environnement. La détection de points saillants dans l'image fournie par une caméra lui permet de ne se focaliser que sur les zones les plus caractéristiques de l'image. En étudiant les voisinages de chaque point clé de l'image, le réseau de neurones peut construire une représentation de l'environnement dans lequel il évolue, à travers les différents éléments visuels qu'il rencontre. La limitation des données visuelles aux voisinages des points-clé permet de réduire la quantité d'informations que le réseau de neurones doit analyser, tout en augmentant leur pertinence.

1.1.3 Place du système de vision au sein du robot

La figure 1.1 représente le système de vision dans le système robotique, ainsi que les interactions entre les différents éléments du robot.

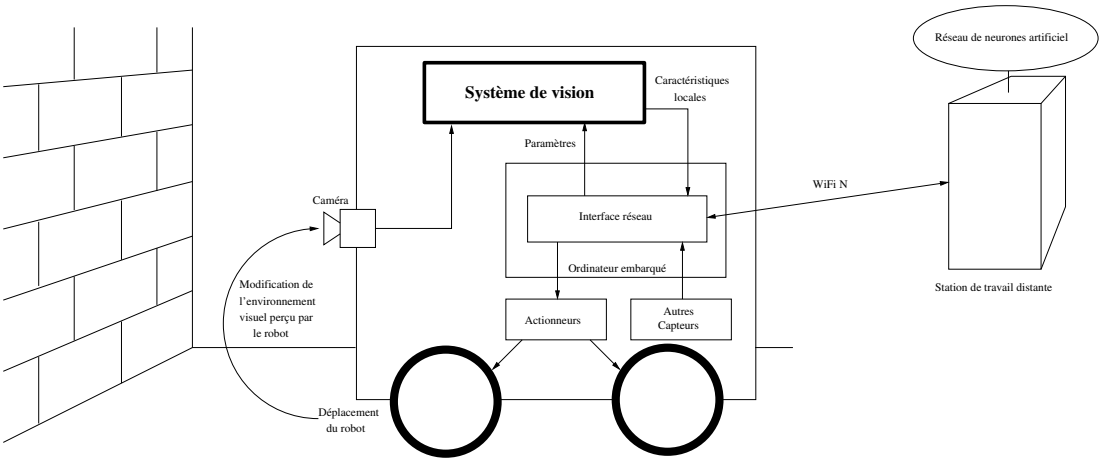


FIGURE 1.1 – **Place du système de vision dans le robot** - Le système de vision est directement connecté à la caméra embarquée. Il envoie ses résultats au PC embarqué, qui lui impose ses paramètres

Le robot hébergeant le système de vision dispose, comme élément central, d'un PC embarqué. Cet ordinateur est responsable de l'interface avec un réseau de neurones, cœur décisionnel du robot. Pour le moment, ce réseau de neurones demande trop de ressources de calcul pour être exécuté directement sur ce PC embarqué, c'est pourquoi il est déporté sur des stations de calcul externes, reliées au PC embarqué par une liaison WiFi N. Le système de vision doit pouvoir se connecter à ce PC pour lui communiquer ses résultats et recevoir ses paramètres. Cette connexion doit se faire à travers une liaison Gigabit Ethernet, standard de communication au sein du robot pour les actionneurs comme pour les capteurs.

Le fonctionnement du robot est le suivant : la caméra capture un flux d'images, qui sont étudiées par le système de vision. Ce dernier cherche les zones pertinentes de l'image, et pour chacune d'elles, génère des caractéristiques locales dans un format bio-inspiré, adapté au réseau de neurones. Le réseau de neurones, cœur du robot, reçoit ces caractéristiques, en plus des informations des autres capteurs, et prend les décisions

d'activation des actionneurs. La navigation du robot se fait à l'aide des roues motrices, les déplacements étant ainsi initiés par le réseau de neurones en fonction de sa perception de son environnement.

On remarque une boucle de rétroaction principale : l'environnement visuel capturé par la caméra influe sur les résultats fournis par le système de vision au réseau de neurones. Ce dernier fait se déplacer le robot en fonction de ces résultats, ce qui modifie le point de vue du robot, et donc l'environnement visuel capturé par la caméra, puis perçu par le réseau de neurones à travers le système de vision. Une boucle de rétroaction secondaire est due aux paramètres du système de vision, qui lui sont fournis par le réseau de neurones, et qui modifient les caractéristiques locales qu'il enverra à ce dernier.

Dans un système mobile réagissant en fonction de ses capteurs, cette rétroaction, ou boucle sensori-motrice (MGGH05), rend la modélisation du système global particulièrement difficile. En effet, pour modéliser de manière fiable le comportement du robot dans un environnement visuel donné, il faudrait être capable de modéliser cet environnement visuel potentiellement complexe, les défauts des capteurs (bruits dans l'image) et des actionneurs (vibrations, temps de réponse, déformations, etc.). Cette modélisation étant bien trop complexe, on utilisera lors des tests, comme stimuli du système de vision, une séquence vidéo capturée lors d'une mission de navigation d'un des robot ciblés par le système de vision.

1.1.4 Contraintes temps-réel non-strictes

L'application de vision reçoit 25 images par seconde, et doit générer les résultats correspondant à chacune de ces images, sans pour autant qu'un dépassement de délai occasionnel ne pose de problème grave au reste du système. L'application fonctionne par conséquent en temps-réel non-strict, on souhaite respecter cette cadence, en restant donc en moyenne en dessous de 40ms par image.

La définition du terme "temps-réel" est un sujet de malentendus et de discussions animées, étant donné les différences fondamentales entre le temps-réel dit "non-strict" et le temps-réel dit "dur", seul vrai "temps-réel" pour la communauté scientifique associée. On parlera de temps-réel dur pour une application qui doit respecter à tout moment

des contraintes temporelles strictes. Le moindre dépassement de ces durées maximales d'exécution est alors considéré comme une défaillance grave du système. Il est à noter que la rapidité du système ne rentre pas en ligne de compte pour déterminer si une application est temps-réel "dur", une application fournissant une réponse à une requête au bout de plusieurs semaines peut être considérée temps-réel "dur", du moment que ce temps de réponse reste toujours inférieur à la limite fixée au préalable. Les systèmes critiques reposant sur un fonctionnement totalement déterministe et d'une fiabilité à toute épreuve, ils sont les principaux concernés par le temps-réel "dur".

Tous les systèmes n'étant pas aussi exigeants du point de vue de la fiabilité, il est souvent possible de déployer une application plus performante en réduisant la fiabilité temporelle (tendance des résultats à être générés avant leurs échéances) dans des limites acceptables. Ainsi, le temps-réel "non-strict", plus flexible, regroupe les applications qui doivent fournir un résultat dans des délais fixés à l'avance, mais qui peuvent accepter des dépassement de ces délais, voire des non-distributions de ces résultats, dans une mesure acceptable pour l'utilisateur (notion subjective). Par exemple, un décodeur vidéo sur un ordinateur peut sauter plusieurs images par seconde sans que l'expérience utilisateur ne soit trop perturbée, le maintien de la vitesse normale de la vidéo est par contre essentiel. Cet exemple est assez représentatif du temps-réel non-strict.

Le système de vision robotique est étudiée ici dans un contexte temps-réel non-strict, ce qui permet de fixer une cadence de fonctionnement plus élevée qu'en temps-réel dur : le temps-réel non-strict gère les éventuels dépassements dûs à cette cadence plus élevée. Le temps de développement est aussi plus court pour mettre en place une application en temps-réel non-strict, une application en temps-réel dur nécessitant une étude temporelle approfondie (durées d'exécution maximales, risque d'interblocage, et autres) pour garantir le respect à tout moment des contraintes. La prédictibilité et par conséquent la fiabilité du système de vision sont les prix à payer pour déployer ce système de vision en temps-réel non-strict.

1.1.5 Consommation électrique et dimensions du système

Le système de vision étant prévue pour des robots mobiles autonomes, il est nécessaire d'identifier et de respecter les contraintes énergétiques, volumiques, et pondérales.

Pour un système alimenté par batteries, la consommation énergétique est souvent un aspect primordial. Un même système peut être considéré comme exigeant ou économe en énergie électrique selon l'environnement dans lequel il sera implanté. En effet, une même puce électronique pourra avoir une consommation négligeable pour un camion, mais trop grande pour un téléphone portable. Pour le déploiement d'un système embarquée, il convient de choisir une plate-forme respectant les contraintes électriques de l'hôte qui l'alimente (le robot dans le cas présent). En fonction de l'énergie qu'il est acceptable de consommer au sein du robot (dimensions de la batterie propre, ou puissance disponible sur la batterie principale), le déploiement du système de vision peut être très fortement contraint.

Parallèlement à la contrainte en puissance électrique, un système embarqué est généralement contraint à un encombrement limité. Il faut alors concevoir ce système en gardant à l'esprit ces contraintes de dimensions. Une capsule endoscopique et le système informatique d'un bateau de croisière n'ont pas les mêmes contraintes de volume, il faut donc évaluer la capacité de stockage physique de l'environnement pour envisager la conception du système. On notera que dans le cas de la capsule endoscopique, les contraintes d'encombrement sont telles, qu'on n'embarquera que le minimum dans la capsule, la quasi-totalité des traitements étant déportés sur des unités de calcul externes. En revanche, dans le cas d'un robot mobile autonome, on ne déportera pas de traitements à l'extérieur du robot, sans quoi on ne parlera plus de robot autonome. Comme pour la consommation électrique, les plate-formes envisageables pour répondre aux besoins du système de vision en accord avec les contraintes d'encombrement sont en nombre limité.

Ensuite vient le problème du poids du système. Afin de ne pas gêner les mouvements du robot, il est aussi utile de choisir un système assez léger pour déployer l'application de vision. De plus, un robot plus lourd consommera plus d'énergie pour se déplacer, ce qui augmentera la consommation électrique globale du robot.

Le système de vision étant prévu pour être embarqué sur des robots mobiles autonomes, les choix de conception se voient réduits en conséquence par ces trois aspects.

Le choix de l'unité de traitement découle non seulement d'une faible place disponible sur le robot, mais aussi de sa quantité d'énergie disponible et d'une estimation de poids acceptable pour le robot. Le robot ciblé par le projet étant un robot RobuLAB 10 (Rob), un ordinateur portable est une solution tout à fait envisageable sur le plan des contraintes de dimensions. C'est en effet un ordinateur qui représente l'unité centrale du robot. Les performances d'un ordinateur portable n'étant cependant pas à la hauteur de la tâche que représente l'application de vision, le déploiement de cette application de vision sur un SoC apparaît plus pertinente.

Une carte FPGA consomme généralement une puissance électrique comparable à celle d'un ordinateur portable : une carte Xilinx ML605 (Virtex 6) consomme dans certains cas plus de 26W (Xil09), tandis que certains ordinateurs portables récents consomment moins de 15W en utilisation poussée (HP DM1 3130, processeur dual core basse consommation AMD E-350, autonomie de 4h en utilisation intensive avec une batterie de 55Wh, soit 13,75W). Le FPGA permet en revanche des traitements hautement parallélisés, des circuits logiques sur mesure, et globalement une puissance de calcul bien plus grande pour le domaine du traitement d'image.

L'alimentation électrique du système de vision pourra être prise en charge par une batterie supplémentaire, afin de ne pas perturber le dimensionnement électrique préexistant d'un robot qui l'hébergerait. Dans la plupart des cas, il est cependant préférable d'alimenter tous les éléments du robot par une même batterie afin de réduire le poids et l'encombrement.

1.2 Découpage fonctionnel

La figure 1.2 donne un aperçu du découpage en blocs de traitement du système de vision.

On remarque la caméra en entrée du système de vision, et les sorties dirigées vers le réseau de neurones du robot.

1.2.1 Gradient

La caméra fournit une image en niveaux de gris, qui est étudiée par le système de vision. La première opération appliqué à cette image est le calcul de l'intensité de gradient. Cette étape permet d'obtenir une information sur les variations de luminance

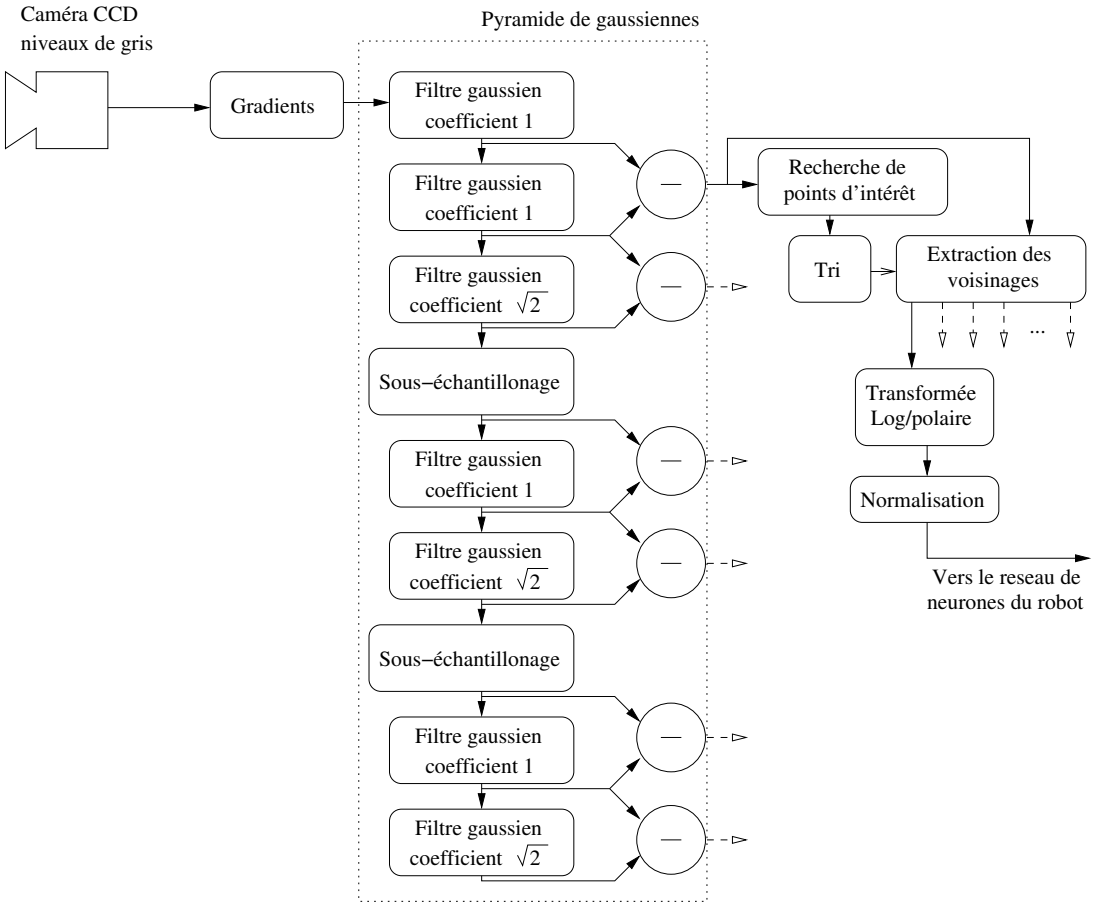


Figure 1.2 – **Description fonctionnelle du système de vision** - Découpage par blocs de traitement, et communications

dans l'image d'origine. Un gradient se représente généralement sous forme de vecteur, indiquant la direction de la variation, ainsi que son intensité. Dans le cas du système de vision, seule l'intensité du gradient, ou la norme du vecteur gradient, est utilisée. En conséquence, on fait l'économie du calcul de sa direction. La sortie de ce bloc est une image représentant par des valeurs élevées les pixels les plus saillants de l'image d'entrée, comme on peut le voir en figure 1.3.

FIGURE 1.3 – **Gradient** - Image provenant de la caméra à gauche, à droite son gradient. Pour une meilleure lisibilité du gradient dans le reste du document, les couleurs sont inversées pour toutes les images, sauf celles de la caméra. Point clair : valeur faible, point sombre : valeur élevée

Le gradient est très utilisée en détection de contours, où la norme du vecteur gradient est utilisée pour détecter les points saillants de l'image, qui constituent les contours. Dans le cas du système de vision, on ne recherche pas les contours, mais les points les plus saillants localement. L'algorithme de détection de contour de Canny (Can86) est un bon exemple de détection de points saillants dans l'image à partir de l'intensité du gradient des pixels. Pour chaque pixel de l'image, cet algorithme génère un vecteur, à savoir une norme et une direction, dans le but d'identifier les contours présents dans l'image. Comme le système de vision étudié n'utilise pas la direction du vecteur, cet élément du gradient ne sera pas présenté dans ce document.

Dans le cas où une étude monorésolution suffit, l'image ainsi calculée pourrait être étudiée directement par les algorithmes de recherche de points d'intérêt. La pyramide Gaussienne permet de séparer son information selon plusieurs bandes de fréquences spatiales.

1.2.2 Pyramide Gaussienne

La pyramide Gaussienne de Crowley et al. (CRP02) est une approche multirésolution du traitement d'image, au même titre que la théorie des ondelettes, qui est aussi utilisée pour la détection de points d'intérêts (FKA06).

Le principe de l'étude multirésolution est de découper le gradient en bandes de fréquence spatiale, ce qui permet d'obtenir des informations visuelles séparées pour chacune de ces bandes. L'étude multirésolution de l'image est utilisée dans le cadre de la vision bio-inspirée (Oue03, GSB04), pour sa modélisation plus précise du fonctionnement du cerveau des mammifères. Elle présente de nombreux avantages quant à la pertinence des résultats obtenus dans les différentes résolutions (Dye87). Le but est ici de permettre au réseau de neurones de mieux représenter son environnement (Mai07), tout en s'inspirant des mécanismes de vision des mammifères.

La première portion de la pyramide Gaussienne se compose d'une cascade de filtres Gaussiens et de ré-échantillonnages, dont le but est de générer des filtrages passe-bas de l'image d'entrée à des fréquences de plus en plus basses. L'image est sous-échantillonnée à chaque octave, ce qui permet de réduire les durées d'exécution des traitements ultérieurs. Ce ré-échantillonnage n'influe pas sur la qualité des informations d'après le théorème de Shannon. Ainsi, une sortie de sous-échantillonnage sera considérée comme un résultat de filtrage Gaussien pour la suite des calculs.

La seconde partie de cette pyramide est constituée de soustractions d'images. Les soustractions pixel par pixel des résultats des filtrages Gaussiens successifs de chaque résolution sont appelées Différences de Gaussiennes (DoG). Ces DoG sont équivalentes à des filtrage passe-bande de l'image d'entrée de la pyramide, dans le cas présent, la norme du gradient de la caméra. Les bandes passantes des filtres équivalents des DoG correspondent à des demi-octaves successives.

Le choix des coefficients σ des filtres Gaussiens permet de régler la largeur de bande fréquentielle en sortie des DoG. On utilisera les valeurs proposées par Crowley et al. qui permettent d'obtenir une largeur de bande fréquentielle d'une demi-octave.

Les demi-octaves sont appairées par octave, pour une même résolution d'image. On parlera alors dans ce document d'échelle ou de résolution, pour désigner une octave, et

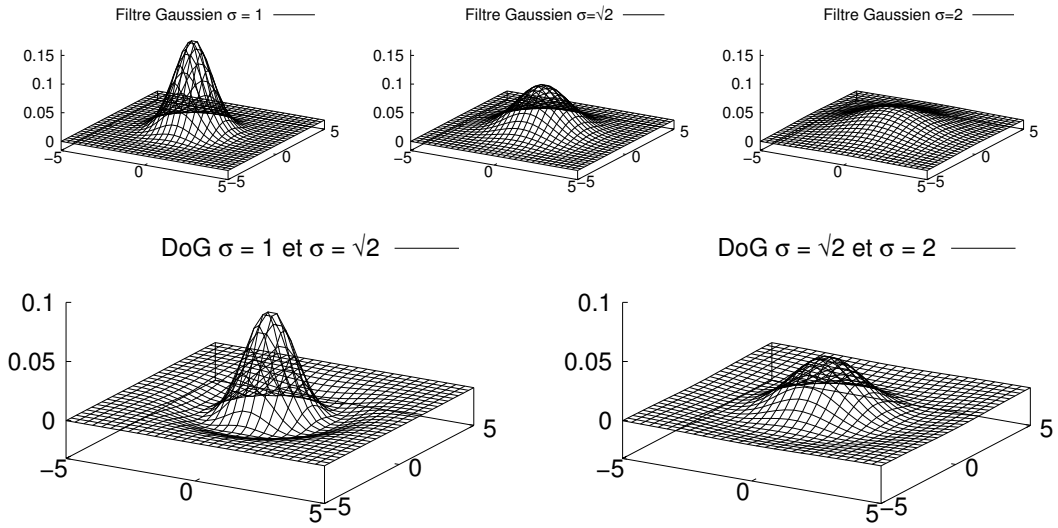


FIGURE 1.4 – **Filtres de la pyramide Gaussienne** - Pour l'octave de plus haute résolution : au dessus les filtres Gaussiens successifs, et en dessous les Différences de Gaussiennes (DoG) correspondant aux deux demi-octaves

de demi-échelle pour l'ensemble des traitements concernant une même demi-octave.

L'aspect des filtres en lui-même est présenté en figure 1.4, où l'on peut voir les filtres utilisés pour la première octave : les trois filtres Gaussiens et les deux DoG correspondantes.

Un exemple des résultats de ces filtrages et sous-échantillonnages successifs sur le gradient d'une image réelle sont visibles en figure 1.5, par groupes de trois images d'une même résolution. Sur la même figure, les groupes de deux images représentent les Différences de Gaussiennes obtenues à partir des deux images directement au-dessus de chacune d'elles.

Comme on peut le voir en figure 1.2 (page 12), six images de DoG sont générées, correspondant chacune à une demi-octave de fréquence spatiale. Le gradient et la pyramide Gaussienne permettent de relever les variations de luminance dans l'image à plusieurs bandes de fréquence. Les DoG permettent ainsi de rechercher les points les plus saillants dans chaque bande de fréquence de l'image fournie par la caméra. Les blocs de traitements ultérieurs de l'application (recherche de points d'intérêt et mise en forme des caractéristiques locales) utilisent directement les images générées par ces six DoG.

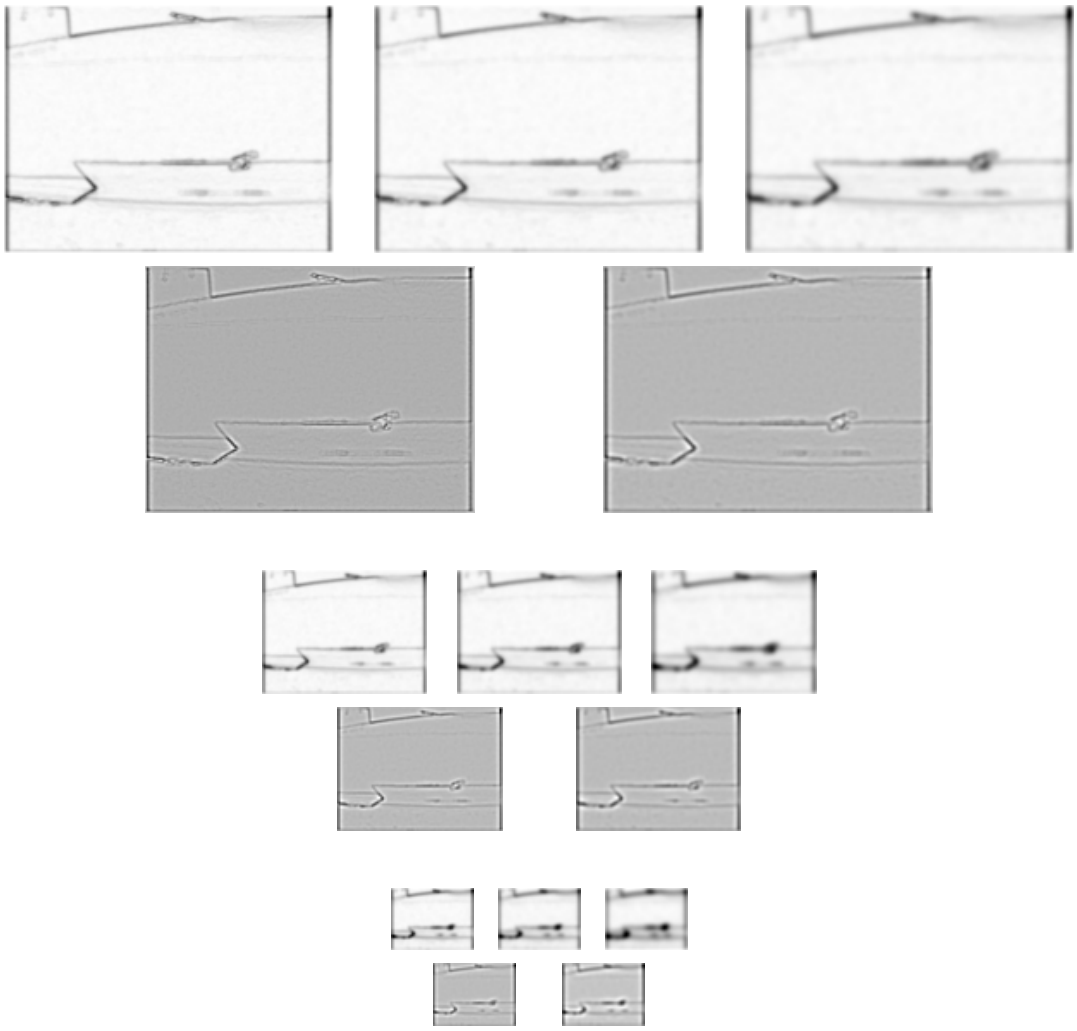


Figure 1.5 – **Pyramide Gaussienne** - Pour chacune des trois résolutions d'image : au dessus les filtrages Gaussiens successifs, et en dessous les Différences de Gaussiennes (DoG) correspondant à deux demi-octaves

1.2.3 Recherche de points d'intérêt

La recherche d'informations dans l'image est un sujet très actif dans le domaine du traitement d'image. Dans (Low99), Lowe s'inspire de la méthode de Schmid et Mohr (SM97) qui étudie les dérivées de Gaussiennes pour localiser des points d'intérêt dans une image, à plusieurs bandes de fréquence spatiale. La variante de Lowe repose sur l'identification d'extrema dans des Différences de Gaussiennes de l'image d'origine.

Dans le cas du système de vision robotique, la pyramide Gaussienne de Crowley a été choisie pour l'étape de découpage multirésolution, cependant la localisation des points d'intérêt se fera de façon similaire à la méthode de Lowe.

Un algorithme de recherche de maxima locaux identifie les points de plus forte luminance locale dans les sorties des DoG. Étant donné que la luminance dans le gradient représente la variation spatiale de luminance dans l'image d'entrée, les maxima locaux dans le gradient représentent les points saillants (les points de grandes variation de luminance) dans l'image de la caméra. Par extension, les maxima locaux dans une DoG représenteront les points les plus saillants dans l'image de départ, dans la bande de fréquence spatiale couverte par cette DoG. Les maxima locaux ne sont pas recherchés dans l'image de sortie du bloc gradient, mais uniquement dans les images de sortie des DoG : les points d'intérêt trouvés seront ainsi toujours associés à une information sur leur bande de fréquence correspondante.

La recherche de maxima locaux se fait sur une fenêtre glissante circulaire autour des pixels testés. Si un pixel dans un rayon R autour du pixel étudié est supérieur à celui-ci, le pixel étudié ne sera pas considéré comme maximum local, et par conséquent ne sera pas identifié comme un point d'intérêt. En effet, on souhaite qu'une distance minimale soit respectée entre deux points d'intérêt, c'est le sens du mot "local" dans le terme "maximum local". Une fenêtre carrée impliquerait que cette distance varierait selon l'angle par rapport auquel deux pixels se trouvent : deux points d'intérêt sur une même ligne devraient respecter un écart de R, tandis que deux points sur une même droite inclinée à 45 $^\circ$ par rapport à l'horizontale devraient respecter un écart de $\sqrt{2}R$.

Les recherches de points d'intérêt sont paramétrables, selon deux variables :

- γ : seuil de détection. Pour être considéré comme point d'intérêt, un pixel d'une DoG doit avoir une valeur supérieure à ce seuil. Le réglage de ce seuil permet de réduire le bruit correspondant à des maxima locaux de valeur trop faible. Ces points, jugés trop peu saillants dans l'image de la caméra, sont donc écartés.
- R : rayon de la zone locale de recherche. Le rôle de ce paramètre est d'empêcher la trop forte proximité de deux points d'intérêt. Si deux maxima locaux trop proches l'un de l'autre (donc de même valeur) sont découverts, un seul des deux sera considéré comme point d'intérêt, l'autre sera inhibé. R évolue proportionnellement avec la résolution d'image, d'une octave à une autre.

La figure 1.6 montre un exemple de points d'intérêt trouvés pour une même image fournie par la caméra, pour chaque bande de fréquence. Chaque point d'intérêt est signalé par un anneau de rayon externe R centré sur ce point : cet anneau sera présenté en section 1.2.5. On remarque que les points d'intérêt des différentes bandes de fréquence semblent se trouver sensiblement aux mêmes endroits, cela dit dans l'octave basse fréquence, on voit que le point d'intérêt qui était trouvé vers le milieu de l'image n'est plus détecté.

1.2.4 Tri des points d'intérêt

Le tri des points d'intérêt trouvés par le bloc de recherche permet au réseau de neurone de hiérarchiser ces points par importance, mais le bloc de tri a une fonction supplémentaire :

Pour chaque point qui est détecté comme un point d'intérêt par le bloc précédent, il reste une dernière étape de discrimination. Le paramètre N représente le nombre maximum de points d'intérêt à identifier dans l'image. Ainsi, si le bloc de recherche précédent détecte plus de N points, il faut supprimer les points superflus.

Le bloc de tri récupère les points d'intérêt détectés par le bloc de recherche, et les trie par valeur de luminance décroissante, afin de favoriser les points d'intérêt les plus saillants. Dans le cas où le nombre de points d'intérêt est inférieur ou égal à N, tous les points d'intérêt identifiés par la recherche sont étudiés par la suite des traitements, du plus fort au plus faible. Dans le cas contraire, seuls les N points d'intérêt les plus saillants sont envoyés aux blocs suivants.

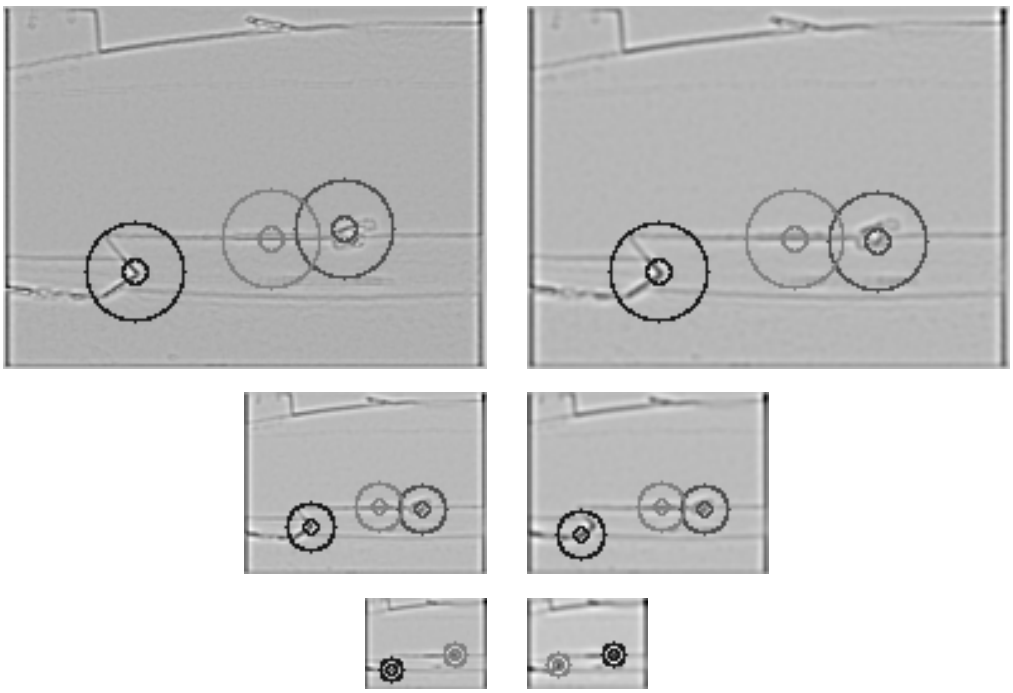


FIGURE 1.6 – **Recherche et tri de points d'intérêt** - Pour les six demi-octaves, les DoG, et leurs points d'intérêt représentés par les anneaux de rayon externe R. Paramètres : {N=15, γ=750, R=[20 ;10 ;5]}

Dans la figure 1.6, le niveau de gris des cercles représente le rang des points d'intérêt après le tri, le plus foncé étant le plus saillant. Ici, γ limite le nombre de points d'intérêt validés dans cette image très monotone. Le paramètre N quant à lui n'intervient pas ici, aucun des points d'intérêt détectés par le bloc de recherche ne sera exclus.

Concernant l'ordre des points d'intérêt, on remarque que les points d'intérêt semblent garder le même ordre, à part pour la demi-octave de plus basse fréquence, pour laquelle on voit que le point d'intérêt le plus à droite devient le plus saillant. On peut donc remarquer que les points les plus saillants ne le seront pas forcément dans toutes les bandes de fréquence spatiale.

1.2.5 Mise en forme des caractéristiques locales

Les points d'intérêt maintenant identifiés et triés, les caractéristiques locales peuvent maintenant être calculées. Ces caractéristiques locales représentent les voisinages de ces points d'intérêt (anneaux de pixels), mis au format log-polaire. Ces résultats seront alors envoyés au réseau de neurones du robot.

En figure 1.7, on peut observer l'aspect d'une image log-polaire, par rapport à l'image en coordonnées cartésiennes d'où elle est extraite. À droite, l'image reconstruite en coordonnées cartésiennes à partir de l'image log-polaire permet d'appréhender la répartition des pixels de l'imagette log-polaire. On remarque une grande précision au centre, des pixels log-polaires qui grossissent en s'en éloignant. Cet exemple utilise des dimensions $\rho_{max} = 40$ et $\theta_{max} = 72$, plus élevées que les dimensions utilisées dans l'application de vision ($\rho_{max} = 5$ et $\theta_{max} = 36$). Cependant, il est à noter que le rayon externe R de l'anneau est de 250 pixels, alors que le même paramètre est de 20 pixels maximum dans l'application de vision. Les dimensions de l'image log-polaire du système de vision restent tout à fait raisonnables.

L'application de vision présentée ici est destinée à la robotique bio-inspirée, le but est donc de modéliser une vision bio-inspirée elle aussi, malgré une caméra traditionnelle. La mise en forme des voisinages des points d'intérêt permet donc au réseau de neurones d'étudier des informations visuelles sous une forme proche de celle observée dans la vision des mammifères. Le format log-polaire est reconnu dans le domaine de la vision biologique pour modéliser efficacement l'information visuelle présente dans le cerveau

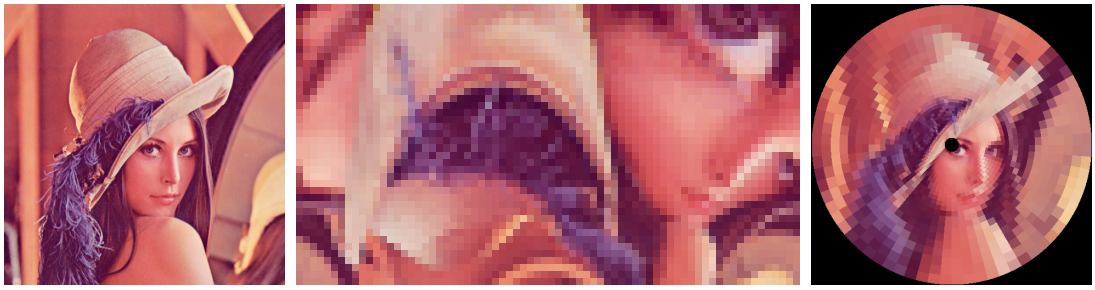


FIGURE 1.7 – **Transformée log-polaire** - (gauche à droite) Image d'origine en coordonnées cartésiennes, image log-polaire, image reconstruite en coordonnées cartésiennes

humain (Sch77). Les avantages de la transformée log-polaire, outre sa pertinence dans un système bio-inspiré, sont une forte robustesse à la rotation et à l'échelle. Pour l'utilisation des informations visuelles dans des tâches de navigation et de reconnaissance, la robustesse à ces éléments est primordiale. La luminosité de l'environnement étant un autre paramètre influant sur l'image fournie par la caméra, il est important de gagner en robustesse aux changement de luminosité. De nombreuses méthodes existent pour normaliser les images de façon à maximiser cette robustesse (SGCZ03), mais restent assez coûteuses en ressources de calculs. Une normalisation des imagettes log-polaires par étalement d'histogramme permet ici de fournir une certaine robustesse aux changements de luminosité, pour un coût en temps d'exécution relativement faible.

Comme on l'a remarqué pour la figure 1.6, une zone en forme d'anneau délimite le voisinage qui sera extrait. Le centre du disque est donc exclu du voisinage de chaque point d'intérêt. Cette zone aveugle est largement présente dans la littérature (TP03, TS93, BSV96). Traver propose plusieurs valeurs pour la taille de cette zone aveugle, et indique les avantages d'une zone aveugle de taille réduite (qualité d'image) ou large (détection de mouvement périphérique). Il est à noter que l'algorithme utilisé prend la taille de cette zone aveugle comme paramètre, en pourcentage par rapport à R, le rayon du voisinage. L'utilisateur peut ainsi modifier la taille de cette zone, ou la supprimer. Les dimensions de la zone aveugle sont pour l'instant laissées aux valeurs proposées par Mickaël Maillard (Mai07), soit $0,2 \times R$.

La figure 1.8 permet de mieux appréhender l'importance des dimensions ρ_{max} et θ_{max} de l'image log-polaire : une certaine harmonie entre ρ_{max} et θ_{max} permettent

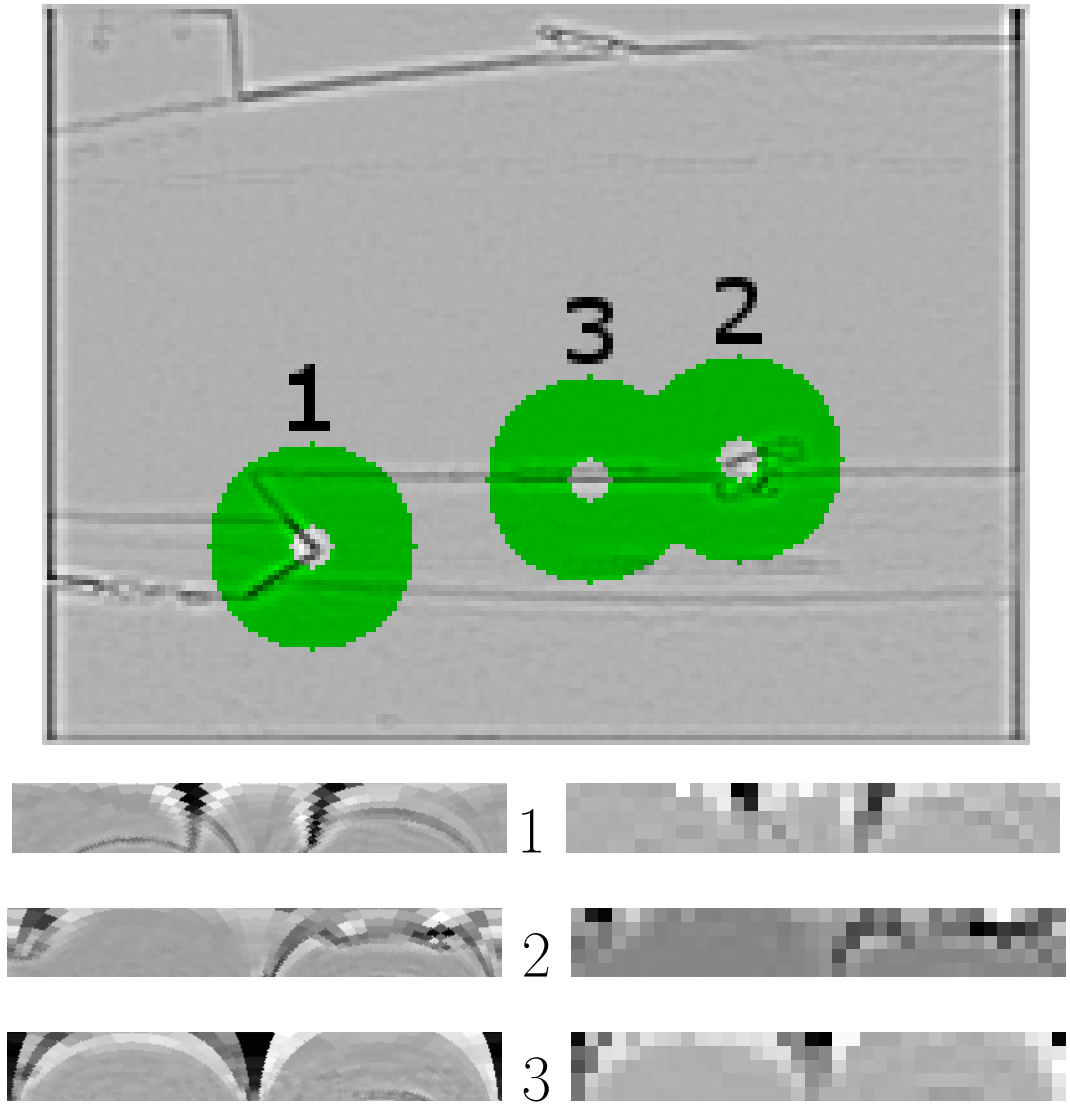


Figure 1.8 – **Extraction de points d'intérêt** - En haut, une DoG, les points d'intérêt identifiés par leur voisinage en anneau. À gauche, $\rho_{max} = 50$ (ordonnée), $\theta_{max} = 360$ (abscisse). À droite, $\rho_{max} = 5$, $\theta_{max} = 36$

d'obtenir des "pixels" log-polaires bien proportionnés pour la modélisation de la vue humaine.

La mise en forme permet au réseau de neurones d'appréhender son environnement visuel sous une forme plus proche de la vision biologique que l'image fournie par une caméra. Cette étape permet non seulement une meilleure modélisation d'un réseau de neurones biologiques, mais aussi une plus grande performance du robot. En effet, si le format cartésien se prête bien au stockage électronique d'images, le cerveau représente les images de toute autre façon, et le format cartésien est assez difficile à appréhender pour un réseau de neurones. Le format log-polaire se rapproche beaucoup plus de la représentation des images dans le cerveau humain, c'est ce format que l'on utilise dans l'application de vision. Du coté des performances, une imagette log-polaire contiendra une représentation pertinente de la zone en anneau, tout en utilisant une quantité de données bien plus faible.

On peut voir en figure 1.8, en haut, une sortie de DoG sur laquelle ont été trouvés 3 points d'intérêt. Les voisinages de ces points d'intérêt, formant des anneaux de pixels autour des points d'intérêt, sont teintés en vert. Les points d'intérêt sont numérotés par valeurs décroissantes. En bas, les voisinages de ces points d'intérêt sont présentés au format log-polaire : horizontalement on remarque le repère θ, qui indique l'angle dans l'anneau, tandis que verticalement le repère ρ indique la distance par rapport au point d'intérêt. Pour la valeur de θ, on utilise la même notation qu'un cercle trigonométrique : $\theta = 0$ à droite, et progression dans le sens anti-horaire dans l'image cartésienne. Les imagettes log-polaires sont présentées en haute résolution à gauche, afin de mieux appréhender le passage de l'anneau en vert à l'imagette log-polaire. À droite, les imagettes log-polaire sont représentées dans la résolution empruntée par le robot. On remarque alors que ce format permet de garder une représentation suffisamment précise vers le centre du voisinage, tout en lissant les parties plus lointaines du voisinage, qui apportent alors une information plus grossière.

1.3 Flot de conception du système de vision

Un flot de conception permet de représenter les étapes nécessaires au déploiement d'un système. Cette section expose le flot de conception suivi durant ce projet. Il permet

de déployer une application sur FPGA, dans l'optique d'un déploiement sous forme hybride logicielle/matérielle. Par logiciel, on entend une exécution de code logiciel sur un processeur ou système multiprocesseur embarqué, la partie matérielle du déploiement étant composé d'IP de traitement dédiées.

En fonction du type d'architecture envisagé, le flot de conception varie fortement : il sera largement plus simple pour un SoC monoprocesseur sans accélération matérielle, que pour un SoC à plusieurs processeurs différents connectés à des unités de traitement dédiées. Un FPGA composé d'une partie logicielle et d'IP matérielles peut répondre aux besoins du système de vision étudié, l'étude logicielle embarquée étant vouée à exposer l'insuffisance d'un système embarqué purement logiciel. En effet, si le système logiciel actuel existe dans une version simplifiée sur des stations de travail puissantes, il est raisonnable d'anticiper qu'un simple portage sur processeurs embarqués ne permettra pas l'accélération des calculs.

Le flot de conception global emprunté est représenté en figure 1.9. Le déploiement d'une architecture hybride dotée d'unités de calcul logicielles et d'IP matérielles nécessite de se pencher sur ces deux aspects de la conception.

Une fois la partie logicielle et la partie matérielle validées séparément, l'intégration de ces deux parties en un seul et même SoC cohérent peut avoir lieu.

Cette section présente les différents aspects de la conception de l'architecture proposée, et approfondit les particularités des différentes étapes du flot.

1.3.1 Flot de conception logiciel embarqué

Au début du projet, l'application est disponible dans un format logiciel traditionnel (section 3.2.1). Ce programme est ensuite porté sur une plate-forme embarquée afin d'étudier son comportement temporel (section 3.2.2).

Le flot de conception logiciel est divisé en deux étapes :

- Un déploiement de l'application complète sur un processeur embarqué de tests, permettant une première caractérisation temporelle.
- À la suite de chaque découpage logiciel/matériel, un déploiement des traitements qui resteront en logiciel, sur le processeur embarqué choisi.

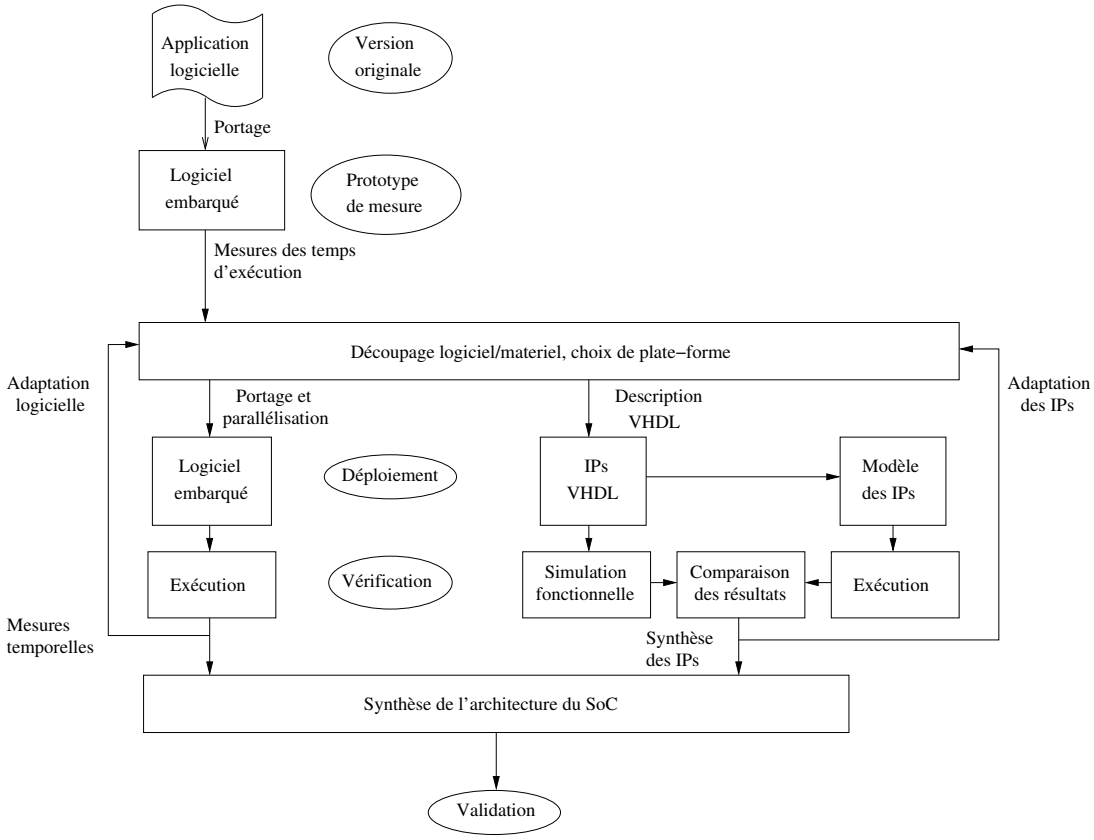


FIGURE 1.9 – **Flot de conception** - Déploiement du circuit avec accélération matérielle. À gauche, le flot de conception logiciel, au centre le flot de conception matériel, et à droite, la modélisation de la partie matérielle

Le découpage logiciel/matériel sera guidé par le comportement temporel de l'application logicielle sur les processeurs embarqués utilisés, et par les contraintes temporelles du robot.

La deuxième étape s'inscrit dans la boucle d'optimisation de l'architecture, et est donc itérative. Elle permet de vérifier le comportement du programme logiciel embarqué définitif (donc délesté des traitements déportés dans la partie matérielle). Des mesures temporelles sont effectuées à chaque déploiement logiciel embarqué. En fonction des résultats obtenus, on pourra raffiner le découpage logiciel/matériel, modifier le choix du ou des processeurs, voire leur nombre. Si les résultats temporels du programme logiciel embarqué parallélisé atteignent des valeurs satisfaisantes, le programme et l'architecture des opérateurs logiciels du circuit sont figés dans leur état quasi-définitif. Il ne reste plus qu'à s'occuper de leur interface avec la partie matérielle. Si en revanche aucune portion essentielle du programme ne respecte les contraintes temporelles en logiciel embarqué, l'application devra être portée intégralement sous forme d'IP, ou utiliser des unités de calcul externes au circuit.

1.3.1.1 Caractérisation de l'application complète

La caractérisation de l'application dans sa version logicielle sur PC permet d'avoir un premier regard sur l'application en connaissant mieux le comportement logiciel de l'application. La connaissance de son comportement temporel permet de guider les choix architecturaux : des traitements qui s'exécutent suffisamment vite restent candidats à un déploiement logiciel embarqué sur le SoC, tandis que les autres traitements nécessitent une accélération matérielle. De plus, ces mesures seront un point de référence pour les mesures sur processeurs embarqués.

L'application est par la suite portée sur un processeur embarqué, son fonctionnement est validé, et les mêmes mesures sont réalisées.

Pour les mesures temporelles sur PC et sur processeur embarqué, l'application étudie une vidéo représentative d'une capture par la caméra du robot en conditions réelles. Cette étape permet d'évaluer les comportements des différentes parties du code logiciel, leurs variations de durée en fonction de paramètres connus, ainsi que leur prédictibilité.

À partir de ces informations, un premier découpage logiciel/matériel peut être effectué.

1.3.1.2 Découpage logiciel/matériel

À l'aide de ces durées d'exécution logicielles sur processeur embarqué, on effectue une première sélection des traitements à déployer sous forme d'IP matérielles. Ces traitements auront en effet été jugés trop longs pour respecter les contraintes temps-réel sur les plate-formes logicielles envisagées. Ces traitements sont supprimés du programme logiciel, et le comportement temporel de l'application est ré-étudié. On part du postulat que la durée d'exécution des IP n'influe pas sur la durée d'exécution logicielle (pas de surcoût de communication, par exemple). Si le temps d'exécution de l'application partielle sur processeur embarqué reste au delà des contraintes temps-réel du robot, il est toujours possible de paralléliser le code logiciel sur plusieurs processeurs.

Le framework conçu par Emmanuel Huck (HMV07) est utilisé à ce moment de la conception, car il permet l'exploration de plusieurs plate-forme logicielles parallélisées. Cet outil permet, entre autres, d'explorer plusieurs architectures hétérogènes ou homogènes, il est possible de simuler la présence d'IP matérielles. L'utilisation de cet outil permet, à l'aide de temps d'exécutions mesurés par fonction, d'évaluer l'efficacité d'un déploiement sur plate-forme multiprocesseur, en fonction du facteur de parallélisation.

Une limitation de cet outil est qu'il ne propose pas de modélisation temporelle des communications, et donc d'évaluation des bandes passantes consommée par le système modélisé. Une étude théorique des coûts des communications inter-processus est alors nécessaire pour évaluer la faisabilité réelle du système, en fonction des différents canaux de communications envisagés.

L'étude de la consommation de bande passante permet, avec la caractérisation temporelle, de guider les choix de l'architecture du SoC en fonction des contraintes de l'application. Les IP matérielles présentent dans ce cas l'avantage de pouvoir être connectées entre elles par des bus indépendants, tandis que les processeurs embarqués partagent souvent des bus de communication et des mémoires RAM, qui créent des goulots d'étranglement. En déployant plus de traitements consécutifs en matériel, la communication

des données sera accélérée, du moins si les communications du ou des processeurs représentent un coût temporel significatif.

Si les contraintes temps-réel de l'application ne sont pas respectées, on procède à l'optimisation de la partie logicielle, ou à une étape de découpage logiciel/matériel supplémentaire.

1.3.1.3 Optimisation de la partie logicielle du SoC

Une fois que le premier découpage logiciel/matériel est fait, il faut s'assurer que la partie logicielle respecte les contraintes temporelles, si ce n'est pas déjà le cas. Deux voies d'améliorations sont possibles : le choix d'une plate-forme logicielle suffisamment puissante, et l'optimisation du code logiciel.

Le choix d'un type de processeur adapté dépend de l'offre existante dans les domaines soft-core, et hard-core. En soft-core, il peut exister des solutions propriétaires selon le fabricant du FPGA (NIOS II chez Altera, MicroBlaze chez Xilinx), mais d'autres solutions indépendantes de la technologie FPGA existent (LEON3 (Gai) par exemple). Des processeurs hard-core intégrés dans le FPGA sont envisageables (PowerPC dans certains Virtex, ARM Cortex A9 dans les Zynq), mais limitent le choix du circuit.

Dans le cas d'un système multiprocesseur, la parallélisation du code logiciel embarqué fait aussi partie de l'étape d'optimisation de la partie logicielle. Le framework de Huck est de nouveau utilisé pour explorer le parallélisme de la partie logicielle du SoC sur le nouveau type de processeurs.

Malgré tous ces efforts, il se peut que les traitements logiciels ne soient toujours pas réalisables dans le respect des contraintes du robot. Dans ce cas, une nouvelle passe du cycle d'exploration est lancée, et des traitements supplémentaires seront portés sous forme d'IP matérielles au sein du SoC, ou déportés sur des unités de calcul externes, pour alléger la charge du système logiciel du SoC. Le processus de conception converge une fois que toutes les contraintes du robot et du système sont respectées.

1.3.1.4 Validation

Plusieurs outils sont nécessaires à la validation du système logiciel embarqué. Le programme sur PC permet de valider rapidement les résultats théoriques, à partir d'une ou plusieurs images d'entrée. Ces images de test sont utilisées en entrée du système logiciel embarqué, afin d'obtenir les images générées par ce dernier. Il est alors aisé de comparer les images générées par les deux versions logicielles. Une fois que les éventuelles erreurs survenues lors du portage (utilisation de bibliothèques inadaptées à la plate-forme choisie, par exemple) sont corrigées, le résultat de la version embarquée doit être identique à ceux de la version PC, sauf réduction de précision lors du portage. Dans ce dernier cas, l'erreur entre les deux versions est relevée, pour valider ou non les résultats de la version la moins précise.

1.3.2 Déploiement matériel

Une fois que les traitements destinés à être déployés sous forme d'IP sont identifiés, la conception de ces IP reste à effectuer.

1.3.2.1 Parallélisation

La parallélisation des calculs permet de multiplier le débit des données traitées par une IP. Le coût en ressources matérielles de cette solution est son principal défaut : une forte parallélisation est possible avec des unités de calcul légères, mais dès lors qu'une unité de traitement prend une place importante sur le FPGA, il devient difficile d'en déployer un grand nombre. Dans tous les cas, une forte parallélisation implique une forte accélération des traitements, mais une forte augmentation des ressources nécessaires.

Certains algorithmes se parallélisent massivement de façon évidente, tandis que d'autres traitements restent mieux adaptés à un déploiement logiciel.

L'identification de ces différences fera partie de l'optimisation du découpage logiciel/matériel : les résultats de synthèse des IP et les durées d'exécution logicielle embarquée des fonctions correspondantes modèlent l'architecture du SoC. Dans le cas d'algorithmes particulièrement inadaptés au déploiement sous forme d'IP, la parallélisation logicielle pourra se révéler nécessaire, si l'exécution monocœur ne respecte pas les contraintes temps-réel.

1.3.2.2 Mise en cascade

Afin d'augmenter la cadence de fonctionnement des opérateurs, une solution classique pour un concepteur d'IP est la mise en cascade, ou *pipeline*. Un calcul nécessitant plusieurs étapes successives (multiplication puis addition, par exemple) verra ses résultats intermédiaires stockés dans des registres synchrones avant de subir la suite des traitements. Le résultat du calcul n'apparaîtra alors qu'au bout d'un nombre de coups d'horloge égal au nombre d'étages de cascade (nombre de registres à traverser pour une même donnée).

Si le temps de parcours total de l'opérateur (durée du début du flot de pixels en entrée à la fin du flot de pixels en sortie) est ainsi allongé d'une latence due à la cascade, la fréquence d'horloge peut quant à elle être revue à la hausse, en fonction de l'élément le plus lent de la cascade. On peut ainsi augmenter fortement la quantité de données traitées dans un même intervalle de temps, au prix d'une latence un peu plus élevée et de registres supplémentaires.

1.3.2.3 Validation - modèle logiciel d'IP

Sur la figure 1.9, on peut voir deux parties parallèles pour la partie matérielle. Afin de s'assurer que les IP fonctionnent correctement, leur simulation fonctionnelle est effectuée. Un modèle logiciel de chaque IP VHDL est réalisé, afin de permettre plus facilement et plus rapidement d'étudier l'impact des choix architecturaux sur le résultat fonctionnel du système. Ce modèle permet de modifier les algorithmes utilisés, mais surtout les tailles des bus de données de chaque variable afin de les régler finement dans le code des IP. De plus, les résultats de la simulation fonctionnelle sont comparés aux résultats générés par le modèle logiciel, afin de déceler les erreurs de synchronisation des données à l'intérieur des IP.

Les résultats générés par le modèle logiciel permettent de valider le fonctionnement attendu de l'IP, en sauvegardant les images générées par le modèle. La simulation fonctionnelle des IP ne permettant pas d'obtenir immédiatement un résultat visuel, on pourra comparer les résultats numériques sauvegardés par le modèle et ceux que fournit l'outil de simulation.

La simulation fonctionnelle permet de vérifier la validité du code VHDL en étudiant le résultat généré par l'IP, à partir d'un stimuli déterminé au préalable. Un modèle logiciel des IP est utilisé pour accélérer ces vérifications, en modélisant les signaux internes de l'IP et ses opérateurs tels qu'ils sont censés fonctionner. Si les résultats générés par le modèle et la simulation fonctionnelle sont identiques, on peut supposer que l'IP est validée. Dans le cas contraire, on étudiera les signaux internes de l'IP afin de localiser les dysfonctionnements pour les corriger.

1.3.3 Architecture globale du système de vision

Les IP matérielles et la plate-forme logicielle faisant partie d'un même SoC, il reste à les assembler dans un même FPGA. Le choix de la plate-forme logicielle a une grande influence sur le circuit : le choix d'un type de processeur soft-core (NIOS II, MicroBlaze) peut limiter à certaines plate-formes (respectivement : FPGA de marque Altera, Xilinx). Un processeur hard-core limitera encore plus l'éventail de FPGA envisageables.

1.3.4 Tests et validation

La validation du système au cœur du robot est l'étape finale de la conception de ce système de vision. Il s'agit de tester l'intégration du système de vision avec une caméra et un réseau de neurones, sur un robot mobile dirigé par ce dernier. Le SoC communique au robot les caractéristiques locales qu'il a extraites de l'image de la caméra, et le PC envoie au SoC les valeurs de ses paramètres variables.

Plusieurs étapes se dégagent de cette phase de tests :

- Intégration de la caméra : au sein du robot, la caméra est disposée sur une plate-forme mobile, tout comme la tête d'un mammifère lui permet de regarder dans plusieurs directions sans devoir changer la position du reste du corps. Il est alors important d'utiliser une caméra qui soit compatible avec ce besoin de mobilité. La mise en place de l'interface entre les IP et la caméra est aussi une étape à prévoir.
- Alimentation du système de vision : la batterie principale du robot n'étant pas nécessairement suffisante pour l'alimentation de la carte FPGA, il sera peut-être nécessaire d'alimenter cette dernière par une batterie et un circuit d'alimentation dédié.

- Connexion au réseau de neurones : les capteurs et actionneurs du robot sont vus par le réseau de neurones à travers une interface Gigabit Ethernet. Il est important de mettre en place et valider la communication à travers ce canal pour que les tests se déroulent bien.
- Enfin, l'étape principale consiste à faire fonctionner le robot dans diverses missions, et à régler les différents paramètres du système de vision afin d'obtenir le meilleur comportement possible du robot.

Chapitre 2

État de l'art

2.1 Vision artificielle

2.1.1 Vision robotique

L'industrie a depuis longtemps recours à des robots percevant leur environnement par capteurs vidéo, que ce soit dans l'industrie automobile ou aéronautique, ou dans d'autres secteurs moins représentatifs du marché. On peut citer la détection de défauts dans une chaîne de production : dans (LZ07), l'auteur propose une méthode pour accélérer la classification d'échantillons.

La robotique mobile, quant à elle, exhibe la plupart du temps des besoins particuliers, liés au besoin d'embarquer les unités de calcul au sein de la plate-forme mobile. Il est alors nécessaire de se pencher sérieusement sur l'architecture système qui permettra un fonctionnement acceptable du robot tout en respectant les contraintes de mobilité du robot (autonomie électrique, volume, poids).

Dans (WLY08), Wu et al présentent un système de contrôle embarqué, déployé sur une carte ARM, pour les véhicules automatisés guidés par lignes au sol. Une carte DSP présente dans le contrôleur permet d'accélérer les traitements d'image, trop lourds pour le microcontrôleur ARM. Ce dernier, pouvant héberger un OS temps-réel de type μC/OS II, permet le déploiement d'une partie logicielle plus complexe. Les auteurs s'appuient sur des tests effectués sur un robot mobile, dans le cadre d'un suivi de tracé au sol, puis présentent les résultats obtenus à l'aide de leur contrôleur, afin de juger de sa fiabilité et de sa performance. L'utilisation d'une architecture hétérogène a permis d'optimiser

le système en fonction du type de traitements effectués par le robot.

Certaines applications de vision artificielles nécessitent une grande puissance de calcul, ce qui ne semble pas correspondre aux capacités des robots mobiles de dimensions réduites. La conception de systèmes dédiés à ces applications est souvent nécessaire pour leur permettre de satisfaire les contraintes de la robotique mobile. Les circuits FPGA présentent l'avantage d'être flexibles tout en restant puissants : une architecture matérielle adaptée aux traitements d'image peut être déployée au plus près des besoins de l'application, et modifiée selon l'évolution des besoins.

Dans (BMC08), Bonato et Al. présentent un système de détection de caractéristiques multirésolution, appliqué à la robotique. Leur approche se base sur la méthode SIFT (Low04) pour détecter les caractéristiques dans des images provenant d'un capteur CMOS 320×240. Ce système est très similaire à celui que nous étudions, mais les caractéristiques générées sont très différentes. Le système étudié dans notre cas sera préféré à ce système, car les caractéristiques locales étant basées sur une modélisation biologique, ce qui n'est pas le cas ici. Le déploiement s'effectue sur FPGA : la plupart des traitements sont déployés sous forme d'IP matérielles, et un processeur embarqué de type NIOS II exécute la génération des descripteurs des caractéristiques de l'image. L'étude multirésolution se fait à l'aide d'une pyramide Gaussienne, ce qui implique des IP de filtrage Gaussien, ainsi que des Différences de Gaussiennes que l'on retrouve dans le projet présenté dans ce mémoire. L'orientation et l'intensité du gradient sont calculées non pas sur l'image fournie par la caméra, mais sur les images de DoG. L'algorithme CORDIC est choisi dans le calcul du gradient, pour l'efficacité de son déploiement sur FPGA. L'architecture globale du système se compose d'une partie matérielle, qui rassemble la pyramide Gaussienne, le bloc CORDIC et la détection de points d'intérêt, ainsi que d'une partie logicielle embarquée : un processeur NIOS-II d'Altera, qui génère les descripteurs SIFT. Les performances de ce système permettent de traiter des images de 320×240 pixels à une cadence de 30 image par seconde. Par rapport à la version logicielle de ce système, la version FPGA permet une accélération d'environ un ordre de grandeur. La conclusion des auteurs est que le processeur soft-core utilisé limite fortement les performances du système, et qu'un déploiement sur un FPGA doté

de processeurs hard-core permettrait de réduire fortement cette limite.

À Brigham Young University, la vision robotique accélérée par FPGA a été utilisée comme support pédagogique, pour mettre en application un cours de vision robotique (ALA05). Les étudiants disposaient de petits robots mobiles autonomes, équipés de FPGA Virtex II. Les robots étaient testés à l'aide de plusieurs épreuves de navigation, telles que le contournement d'obstacles. L'article présente l'approche empruntée par l'une des équipes d'étudiants. Le FPGA sélectionné héberge un processeur MicroBlaze avec l'OS μC/OS II, permettant un découpage des programmes en tâches et leur exécution en temps-réel. Une IP matérielle de vision stéréoscopique, étudiant les histogrammes d'images capturées par deux caméras, est ensuite présentée.

Dans le contexte de la détection d'objets pour l'assistance à la conduite, on peut évoquer les travaux de Schauland et al. Dans (SVK08), un déploiement sur FPGA de filtres 3D pour la détection de mouvement est présenté. Les filtres déployés sont ici de faible complexité, et ne nécessitent qu'un FPGA Spartan3E de Xilinx. Les résultats obtenus par ces filtres matériels, illustrés par une application de détection de voitures par une caméra de surveillance routière, montrent de bonnes performances, de sorte que les auteurs jugent opportun de pousser ces travaux jusqu'à un déploiement de SoC complet de détection d'objets.

2.1.2 Vision artificielle bio-inspirée

Une grande partie des systèmes électroniques de vision bio-inspirée s'inspirent de la rétine des mammifères, la plupart du temps de la rétine humaine.

On peut citer parmi les rétines artificielles embarqués, les travaux d'Elouardi et al. (EBD$^+$04). Les auteurs présentent PARIS, un prototype de rétine CMOS sous forme de système sur puce, dont les résultats sont traités par un processeur ARM7.

Dans (GRRD07), une puce de rétine artificielle est utilisée pour effectuer des tâches de vision bas niveau, comme la détection de mouvements. Un ordinateur reçoit, via un FPGA qui joue le rôle d'interface, les informations provenant de la rétine artificielle. Le modèle de rétine utilisé dans ces travaux (Boa02) est inspiré de la rétine du chat,

et modélise les signaux générés par les couches de traitement bas niveau de la rétine biologique.

La génération automatique de rétines artificielles a aussi été étudiée (MRP+05). Le code de description matérielle (HDL) synthétisable est généré automatiquement à partir d'une description haut niveau, elle-même conçue et simulée à l'aide d'un programme dédié. Afin d'obtenir de meilleures performances temporelles, les systèmes décrits par les auteurs peuvent être déployés sur ASIC plutôt que sur FPGA. Le but principal de ces travaux est la génération de puces destinées à l'implantation in-vivo avec des électrodes, sous forme de neuro-prothèses. L'intérêt présenté d'un déploiement sur FPGA est ici de pouvoir modifier des paramètres du système pour adapter la puce en fonction du patient.

Bolduc présente les différences entre les rétines artificielles traditionnelles qui utilisent une cartographie de cellules réceptrices non recouvrantes, et un type plus avancé de rétines artificielles (BL98). Ces dernières modélisent le fonctionnement des cellules ganglionnaires, qui induisent un recouvrement partiel des cellules réceptrices dans la génération des données de sortie de la rétine. Si les premières fonctionnent généralement à des plus hautes cadences d'images, les dernières présentent une piste de recherche particulièrement prometteuse.

2.2 Caméras intelligentes

Ces dernières années, la thématique des caméras intelligentes et rétines artificielles gagne de l'ampleur. On observe plusieurs angles d'attaque sur cette thématique : l'étude technologique, tendant à concevoir des capteurs intelligents au niveau circuit, et une étude système utilisant des capteurs visuels existants, couplés avec des systèmes destinés au traitement de l'image. Certaines études se situent entre les deux approches, utilisant une première étape de pré-traitement des pixels sur le circuit même du capteur, et une deuxième étape de pré-traitements plus complexe sur un élément de calcul couplé au capteur.

Le terme "caméra intelligente" est encore relativement jeune, et est utilisé pour divers types de systèmes, comprenant les réseaux de caméras de surveillance intercommunicantes et les capteurs optiques de souris, ainsi que les rétines artificielles conte-

nant une ou plusieurs couches de traitement. En général, on appelle caméra intelligente un système où un capteur optique se voit associé à un circuit de traitements dédiée, sur une même puce où juste sous la forme d'un système à fort couplage. Dans le livre (Bel09), un historique des caméras intelligentes permet d'aborder ce sujet plus en profondeur. Cette section se focalise sur les architectures matérielles des caméras intelligentes.

L'avantage principal d'une caméra intelligente, dû au couplage étroit d'un capteur avec une unité de traitements dédiée, est une faible latence entre la capture d'une image et son utilisation par l'unité de traitements. Le système peut ainsi corriger plus rapidement certains paramètres de capture en fonction des résultats de l'architecture dédiée qui y est associée, augmentant ainsi la qualité de l'information générée par la caméra intelligente. Un avantage essentiel que l'on retrouve sur certains systèmes est une considérable réduction de la quantité de données, à l'aide d'étages de pré-traitements à la sortie du capteur. Cet aspect est essentiel dans les systèmes où une très faible partie des images capturées sont jugées pertinentes, ou/et pour lesquels la distribution de ces pré-traitements sur un réseau de capteurs est préférable à une exécution centralisée (Kha10). Cela permet de réduire, en amont du système client, la charge des canaux de communication, et l'utilisation mémoire, ainsi que la charge de calcul du client qui n'a plus à exécuter ces pré-traitements. Lorsque les flux d'images traités ne sont pas d'une taille trop imposante, une intégration de la caméra à même le circuit de traitements n'est généralement pas indispensable. Une caméra déportée sur carte fille, ou reliée par un autre canal de communication dédié, génère une latence minime et permet l'étude d'images à haute résolution à une cadence suffisante pour beaucoup de systèmes.

Un exemple assez représentatif des caméras intelligentes, présenté dans (LMY04), répond à une problématique proche de la notre. Deux applications de traitement d'image qui ne peuvent pas être déployées en temps-réel sur des stations de travail classiques, se voient accélérées à l'aide d'un FPGA pour devenir utilisables. Pour la première application, les éléments les plus chronophages de la version logicielle pré-existante sont déployés sur FPGA, permettant l'exécution du système en temps-réel avec une latence globale très faible. Dans les deux cas, les gains obtenus sont de l'ordre d'un facteur 20 par rapport aux programmes logiciels pré-existants, exécutés sur un processeur Intel Xeon à 1,5GHz.

Dans (MDBS09), l'auteur présente une caméra intelligente composée non seulement d'une caméra et d'une plate-forme de traitements dédiés, mais aussi de capteurs inertiels. La plate-forme intelligente se compose d'un FPGA central pilotant un DSP. Un processeur virtuel est présenté afin de faciliter le déploiement d'applications sur le système, à travers la programmation en assembleur simplifié. L'interprétation par le processeur virtuel de ce code assembleur se charge d'utiliser les modules de traitement présents sur le système de façon transparente pour le programmeur.

La plupart des travaux existants portent sur l'utilisation de capteurs CMOS traditionnels (lecture par colonnes de pixels), comme par exemple (Koz06), où une solution de suppression du bruit sur les pixels est proposée, sans remettre en question l'organisation en lignes et en colonnes traditionnelle. Parallèlement, plusieurs travaux ont été réalisés pour intégrer l'architecture de traitement au plus près des capteurs photosensibles, afin de permettre plus de traitements au niveau pixel. Une architecture pour la vision à haute vitesse présentée dans (KII97) est un exemple représentatif de cette nouvelle étape dans l'évolution des capteurs vidéo. La conception passe tout d'abord par un prototypage sur FPGA, pour être envoyée à la fabrication en full-custom. L'architecture présentée se compose d'un tableau de 8×8 éléments de calcul hébergeant chacun un photo-récepteur. Chaque élément de calcul se compose d'une ALU, d'une mémoire de 24 bits, et reçoit les valeurs des pixels de ses quatre voisins les plus proches. L'architecture est ensuite déployée sous la forme d'un tableau de 64×64 pixels (IK01).

L'architecture EYE-RIS (ACJG$^+$09) est un exemple plus récent construit autour d'un processeur à plan focal. La famille de plate-formes bio-inspirée EYE-RIS repose sur un étage de traitements parallèles effectués à même le capteur visuel, sous forme analogique, de façon similaire à une rétine. Ce capteur actif est appelé processeur à plan focal (PPF), et est ici intégré sur le même circuit imprimé qu'un FPGA. Au fil du remaniement de l'architecture, plusieurs PPF ont été utilisés : les versions 1.0 et 1.1 du système utilisent un ACE16K-v2 (RVLCC$^+$04), et une nouvelle puce, Q-Eye (RVDCF$^+$07), est développée et employée dans la version 1.2, augmentant la densité des cellules, réduisant la consommation de la puce tout en augmentant la puissance de calcul des cellules. Au second plan des traitements de l'image, le FPGA (un Cyclone d'ALTERA pour la version 1.1 de EYE-RIS) héberge un processeur soft-core NIOS II qui exécute des programmes chargés depuis un ordinateur, donnant une plus grande

flexibilité au système. L'ordinateur reste connecté au système et dispose des résultats de la caméra intelligente.

Au LEAD, un autre système à plan focal, cette fois une puce de vision SIMD rapide (GDHP10), se construit sur le plan focal sous la forme d'une grille de processeurs SIMD reconfigurables dynamiquement. Chaque pixel est composé, en plus d'une photo-diode, de deux blocs analogiques permettant de mémoriser les valeurs du pixel pour l'image présente et l'image précédente, et d'une unité arithmétique analogique responsable de combiner les quatre pixels adjacents à l'aide de quatre multiplieurs analogiques et d'un noyau de convolution 2×2. Les auteurs présentent un prototype de démonstration de 64×64 pixels, permettant une capture RAW à 10000 image par seconde, ou 5000 images par seconde en appliquant un traitement basique au niveau pixel.

L'émergence des technologies de puces 3D a eu un impact sur la communauté des caméras intelligentes, on voit en effet apparaître assez tôt des rétines artificielles exploitant ces nouveaux outils. Dans (KNL$^+$00), une puce de vision utilisant la technologie LSI 3D est présentée. Comme pour les capteurs intelligents de l'architecture EYE-RIS, les traitements effectués sur l'image se font en analogique. La couche de surface héberge les photo-récepteurs, les deux couches suivantes hébergeant les pré-traitements analogiques. Cette fois aussi, l'inspiration biologique va dans le sens d'un capteur fortement couplé aux premiers opérateurs. La puce créée se rapproche d'une rétine biologique sur le plan fonctionnel, à part la densité des capteurs qui reste stable sur la totalité de la puce, contrairement à une rétine humaine. Le but du rapprochement aux systèmes biologique est alors d'en exploiter les qualités sans se priver des performances proposées par l'approche classique. Le coût de fabrication d'une puce 3D étant loin d'être négligeable, on peut comprendre que la plupart des architectures proposées aujourd'hui restent majoritairement sur technologies 2D.

Au sein de l'équipe ETIS, hébergeant le projet présenté dans ce document, les architectures reconfigurables pour le traitement d'images rapide ont déjà été étudiées en profondeur, notamment avec la plate-forme ARDOISE (AKD03). Ce système de mezzanine de cartes FPGA permet de tirer parti du principe de reconfiguration dynamique permis par ses circuits AT40K d'ATMEL.

Le LASMEA, quant à lui, héberge le projet de caméra intelligente SeeMOS (CB07), basée sur un FPGA Altera Stratix, connecté au client par USB2. L'imageur est déployé sur une carte fille à l'orthogonale de la carte FPGA conçue spécifiquement pour le projet.

Une méthodologie de design basée sur le langage CAL est proposée pour permettre aux concepteurs logiciels de déployer leurs applications sur cette plate-forme plus aisément (RBSE10).

Au sein de la communauté, une tendance qui prend de l'ampleur depuis quelques années est l'utilisation de caméras intelligentes distribuées communicantes (RWS+08). Les auteurs proposent une méthode de classement des différents types de caméras intelligentes, en fonction de leur autonomie, leur puissance de calcul, leur degré de distribution. Trois catégories sont proposées pour répartir les différents systèmes de caméras intelligentes : les systèmes à caméra unique, à caméras distribuées, et les systèmes à caméras sans fil. Le travail présenté ici se situe dans la première catégorie, soit les systèmes à caméra unique.

2.3 Traitement d'image

2.3.1 Caractérisation d'images

Le principe de fonctionnement visuel du robot est la caractérisation de son environnement visuel, qui lui permet de modéliser son environnement à l'aide de différentes caractéristiques visuelles capturées par sa caméra. Le SoC de vision aura pour but de simplifier les données visuelles qu'aura à traiter le réseau de neurones, qui pourra ainsi modéliser son environnement de façon pertinente tout en déployant moins de ressources qu'en étudiant directement l'image de la caméra. La caractérisation d'images est ici la solution, cet aspect du traitement d'image se focalisant sur l'extraction d'informations pertinentes dans les images.

Plusieurs méthodes existent pour la détection et l'extraction de caractéristiques dans une image, certaines sont incontournables. En 1980, les travaux de thèse de H.P. Moravec (Mor80) s'appuient sur un robot mobile utilisant une caméra embarquée pour étudier son environnement visuel. Le détecteur de coins de Moravec est alors créé afin de permettre au robot de mieux appréhender son environnement visuel. Ces travaux font de Moravec un pionnier de la caractérisation d'images moderne.

Plus tard, la méthode SIFT de Lowe (Low99, Low04), propose un détecteur de point clés d'une image, ainsi que la génération de descripteurs permettant d'identifier ces points clés dans différentes scènes visuelles. Les descripteurs proposés par Lowe exhibent une invariance à l'échelle et à la rotation de l'image étudiée, ainsi qu'une

robustesse aux changements de luminosité, de point de vue, et de netteté. Ces qualités rendent la méthode SIFT incontournable dans le domaine de la caractérisation d'images.

Un aspect important de la méthode SIFT est l'étude multirésolution de l'image. Les méthodes de traitement d'image multirésolution ont émergé en 1971, dans une optique de planification (Kel71), l'utilisation en caractérisation d'image aura pris de nombreuses années, du fait de la puissance de calcul disponible à l'époque. En 2002, James L. Crowley, Olivier Riff, et Justus H. Piater présentent une nouvelle méthode multirésolution : la pyramide Gaussienne (CRP02). Leur approche se base sur les pyramides de Laplaciens de Gaussiennes, et vise une accélération des calculs en évitant une trop forte diminution des performances. L'opérateur Laplacien est remplacé par une simple différence, comme on peut en trouver dans le SIFT, car elle présente des résultats très proches avec un coût en ressources fortement réduit.

La méthode SURF (BETG08), quant à elle, s'avère plus efficace que SIFT, que ce soit en vitesse ou en précision. Un aspect très attrayant de cette méthode pour la navigation robotique vient d'une variante de la méthode SURF : le U-SURF, pour SURF droit (upright). Cette variante du SURF propose l'abandon de la robustesse en rotation, au profit de la durée d'exécution. Bay remarque que les caméras embarquées sur les robots sont généralement fixes sur un axe vertical, ce qui permet de se séparer de la robustesse à la rotation non seulement pour économiser des ressources mais aussi pour gagner en discrimination. En effet, la navigation robotique s'appuie sur des caractéristiques visuelles fixes dans l'environnement, appelées amers. Il est alors utile de reconnaître plus facilement les objets fixes de l'image, et moins facilement les objets mobiles, quand le but du robot est la navigation pure. Les objets fixes dans l'environnement permettent ainsi une cartographie fiable. Le U-SURF ne pourrait être utilisé par un robot mobile que s'il n'exécute que des tâches de navigation, on lui préférera le SURF pour la reconnaissance d'objets.

Dans (BSP07), les auteurs présentent les différences entre plusieurs implantations des méthodes SIFT et SURF, ainsi que le détecteur de Harris. Le détecteur de Harris (HS88), présenté par Harris et Stephen, permet de détecter les coins dans une image, mais ne génère pas de descripteurs de caractéristiques, et ne sera donc pas tout à fait comparable aux méthodes SIFT et SURF. Bauer et al. comparent les algorithmes sur plusieurs points : leur robustesse au bruit, à la rotation, au changements de luminosité et d'échelle ainsi qu'au changement de point de vue. Les méthodes SIFT sont présentées

comme les plus puissantes, les méthodes SURF sont légèrement en retrait. Cependant, les méthodes SURF présentent l'avantage de générer moins de caractéristiques, ce qui réduit considérablement le temps de calcul des traitements postérieurs. SURF est ainsi présenté comme supérieur à SIFT pour les projets pour lesquels le temps de calcul est un paramètre essentiel.

Dans (FKA06), une approche basée sur la transformée en ondelette discrète pour la détection de points d'intérêt est présentée. Les résultats montrent une méthode plus robuste à la rotation que la méthode SIFT, et qui exhibe moins de redondance dans les points d'intérêt détectés.

D'autres études proposent encore d'autres méthodes de caractérisation d'images, comme la méthode CHoG (CTC+09) et les travaux de Mikolajczyk et al. (MS04).

2.3.2 Cartographie log-polaire

La cartographie log-polaire est utilisée par l'application de vision étudiée, pour obtenir dans l'image des caractéristiques locales dans une représentation bio-inspirée. Une étude assez récente présente l'état actuel de l'imagerie log-polaire en robotique (TB10). Le document comporte de nombreux graphiques très clairs représentant l'invariance en rotation et en échelle du format log-polaire. L'auteur propose une vision des efforts pertinents à fournir dans le futur pour faire évoluer la recherche sur l'utilisation du format log-polaire en robotique. Il apparaît dans ce document que les techniques les plus utilisées semblent reposer sur des capteurs adaptés à la vision log-polaire. Dans le cas d'algorithmes de transformation d'images capturées par des caméras classiques, la plupart des méthodes utilisées reposent sur des déploiements logiciels. Dans un article précédent (TP08), l'auteur examine les effets de différents paramètres de la cartographie log-polaire. D'après l'auteur, dans la littérature, le choix de ces paramètres est soit peu étudié, soit peu justifié, ce qui mène à penser que le choix de paramètres est souvent empirique ou raffiné expérimentalement. L'auteur souligne que le choix de paramètres optimaux dépend fortement de l'application qui fera usage des images log-polaire. Les différents paramètres utilisés sont présentés puis leur influence sur l'information générée est étudiée soigneusement. Un outil pour l'optimisation de ces paramètres selon l'utilisation envisagée du capteur est proposée.

Du coté des architectures de transformation log-polaire, Wong et. al (WCLT10) présentent un déploiement sur FPGA dans le cadre de la lecture d'images capturées par

une caméra à objectif "fish-eye" et équivalents. Le capteur, orienté vers le haut, capture l'environnement visuel sur 360 ° sur le plan horizontal. L'algorithme de transformation de l'image est ici utilisé pour dérouler l'image capturée en une image panoramique au format cartésien. Dans ce système, l'image de départ n'est pas pertinente pour l'œil humain, la transformation log-polaire permet de rétablir l'image de façon à ce qu'elle puisse être étudiée par un humain. L'application de vision présentée en section 1.1, quant à elle, se base sur une image tout à fait adaptée à la lecture par un humain, et transforme en log-polaire des zones de l'image pour les rendre plus pertinentes à la lecture par un réseau de neurones, en utilisant un format qui sera cette fois illisible à un œil humain non-averti. Les résultats du déploiement sur une carte d'évaluation Xilinx ML402 (Virtex 4 SX35) montrent une accélération inférieure à un facteur 2 par rapport à un déploiement Matlab sur un ordinateur portable (Intel Core 2 Duo à 1.83GHz, 2Go de RAM DDR2). Les résultats présentés dans cet article illustrent les difficultés d'un déploiement matériel efficient d'une transformation log-polaire, alors même que les dimensions de l'image en coordonnées cartésiennes, et celles de l'image en coordonnées log-polaires sont fixes.

Dans (MA06), une modélisation particulière de la rétine est présentée. Comme les images log-polaires, les images fovéales présentées respectent la différence de résolution au centre de l'image et en périphérie que l'on trouve dans la rétine humaine. La particularité des images fovéales est que la résolution évolue en suivant des rectangles concentriques plutôt que les cercles des images log-polaires. Il est ainsi plus facile de générer une image fovéale à partir d'une image cartésienne. Une solution FPGA est proposée pour un système de transformation d'images cartésiennes en images fovéales. Le système présente des performances intéressantes, mais l'imagerie fovéale ne présente pas de robustesse en rotation, qui reste un point fort de la transformée log-polaire.

On remarquera dans (TB10) que le nombre de déploiements matériels de transformation log-polaire à partir d'images en coordonnées cartésiennes est très réduit. L'utilisation la plus courante du format log-polaire dans la littérature semble très réduite concernant l'utilisation de caméras classiques sans équipement supplémentaire (lentille "fish-eye", miroir hyperbolique, etc.), et les auteurs considèrent le déploiement parallèle de cartographie log-polaire efficaces comme une étape future qu'empruntera la communauté scientifique.

2.4 Aspects technologiques

L'aspect architectural étant le cœur du projet présenté dans ce document, il est essentiel d'étudier les circuits capables de respecter ces contraintes de mobilité et de puissance de calcul.

Un tour d'horizon des déploiements matériels de ce type d'applications est présenté dans cette sous-section, afin de mieux situer le projet qui est présenté par rapport aux réalisations existantes en traitement d'image embarqué, robotique, et/ou bio-inspiré.

2.4.1 Systèmes sur puces (SoC)

L'application de vision étant à déployer sur un système sur puce, il est important de bien saisir les nuances de ce terme qui est apparu ces dernières années. Il regroupe de façon générale les systèmes complexes traditionnellement déployés à l'aide de plusieurs puces distinctes, que l'on aura préféré déployer sur une seule puce, pour une plus grande performance, un moindre coût, un encombrement moindre, ou encore une consommation électrique moindre.

L'exemple le plus médiatisé du SoC est ce que l'on trouve dans les téléphones portables et tablettes, souvent issu des séries OMAP de Texas Instruments.

Les SoC sont généralement réalisés à l'aide de technologies ASIC, comme l'exemple précédent, ou sur FPGA pour des SoC fabriqués en quantités réduites : un ASIC ne devient rentable qu'à partir d'un très grand nombre de puces identiques, comme les SoC de TI, qui sont utilisés dans de très nombreux systèmes. Du coté des FPGA, Altera et Xilinx proposent chacun des outils permettant de déployer des systèmes à base de processeurs soft-core pouvant interagir avec des IP de traitement matérielles, le tout dans la même puce FPGA. Altera propose pour ses FPGA le processeur soft-core Nios II avec son bus AVALON, tandis que chez Xilinx, le processeur soft-core MicroBlaze utilise un bus PLB. Altera expose dans (Alt11) son intention de diversifier sa gamme SoC, et annonce des FPGA utilisant des processeurs ARM et MIPS, en addition aux processeurs soft-core et aux SiP Intel présentés plus loin. Xilinx a diversifié son offre avec une série de puces hybride, les Zynq, qui hébergent un processeur double cœur ARM Cortex A9, cadencé à 800MHz, autour d'un bus AXI-4. La particularité de ce type de circuits est qu'en plus du SoC à base de processeur en dur, le composant contient une zone de type

FPGA reconfigurable : les traitements logiciels disposent de la puissance de calcul du Cortex A9, tandis que les traitements sur IP matérielles peuvent être déployés sur la zone FPGA. Les précédents circuits FPGA de Xilinx dotés de processeurs en dur étaient équipés de PowerPC (PPC-405 pour Virtex II Pro et Virtex 4FX, et PPC-440 pour les Virtex 5 FXT). Altera prévoit des circuits Arria V et Cyclone V disposant de Cortex A9, pour remplacer les systèmes Excalibur en fin de vie.

Toutefois, cette vision industrielle du SoC n'est que la partie visible de l'iceberg, la vision du système sur puce des chercheurs de ce domaine s'étend à plusieurs concepts qui ne sont pas encore très utilisés en production industrielle :

- Les systèmes de type Many-core, hébergeant un grand nombre de cœurs de calcul (32 et au delà). Ils sont pour le moment assez peu représentés dans l'industrie du fait de la jeunesse de cette technologie et du changement de paradigme chez les développeurs logiciels, introduit par la programmation logicielle fortement parallèle.
- Les réseaux sur puce ou NoC (MB08), visant à optimiser les communications internes d'un SoC contenant de nombreuses unités de calcul.
- Les systèmes mixtes, hébergeant en plus d'unités de calcul numériques, des éléments analogiques (communications optiques par exemple), ou des systèmes micro-électromagnétiques (MEMS (GL11)), tels que les accéléromètres que l'on trouve dans certains téléphones portables.

Le SoC n'est pas la seule évolution possible du système traditionnel regroupant plusieurs puces sur un circuit imprimé. Le System in Package (SiP) présente beaucoup de similitudes avec le SoC, à la différence que le SoC est déployé sur un seul bloc de silicium (puce), tandis que le SiP utilise plusieurs puces superposées dans un même boîtier de composant. Le Multi Chip Package (MCP), quant à lui, est très similaire au SiP de par le fait qu'il contienne plusieurs puces de silicium, mais cette fois-ci les puces sont disposées sur un même plan horizontal, à coté les unes des autres. Un bon exemple de MCP est la série Atom E6x5C d'Intel (Int), qui héberge dans un même boîtier une puce Atom (600MHz à 1,3GHz selon modèle) et un FPGA Altera Arria II GX. Le Package on Package (PoP) se rapproche du SiP, mais se compose de plusieurs boîtiers superposés, au lieu d'une superposition de puces dans un seul boîtier.

2.4.2 Parallélisation des traitements

La parallélisation des données ou/et des unités de calcul est une technique permettant l'accélération des traitements de nombreux algorithmes. Même à une cadence plus faible que celle d'un processeur traditionnel (GPP), les architectures hautement parallélisées permettent généralement un gain considérable en vitesse d'exécution globale.

On distingue plusieurs techniques principales de parallélisation des traitements :

- les systèmes multiprocesseur et/ou multicœur, déjà largement utilisés dans les ordinateurs personnels, permettent de paralléliser l'exécution d'un programme pour un coût de développement relativement faible. On utilisera généralement des langages de programmation identiques à un système monoprocesseur, en utilisant une couche logicielle supplémentaire permettant la communication entre les processeurs (par exemple OpenMP, MPI, Intel TBB). D'autres langages dédiés au multi-processeur existent aussi (comme OpenCL).
- les GPU permettent de déployer des traitements sur de nombreuses unités de traitement (SIMD - même instruction exécutée sur plusieurs données simultanément). Leur architecture excelle dans les traitements réguliers de grosses quantités de données. Comme pour les systèmes multiprocesseur, la modification des traitement se limite à de la modification de code, l'architecture restant identique.
- la description matérielle, que la cible soit un ASIC ou un FPGA, permet un contrôle extrêmement fin de l'architecture du système. Cette solution propose une très grande flexibilité au moment du développement. L'ASIC permet de grandes cadences de fonctionnement, tandis que le FPGA propose une flexibilité de l'architecture, qui pourra être modifiée au cours de la vie du système. Toutefois, un désavantage conséquent de cette technique est le temps de développement du système, généralement plus long que les deux solutions précédentes.

Les systèmes multiprocesseur et les systèmes multicœur sont considérés comme un même domaine de solutions dans ce document, malgré leurs différences du point de vue du développeur. Le système multicœur met à la disposition des cœurs un ou plusieurs niveaux de mémoire cache partagée (le niveau L1 n'est presque jamais partagé). Le développeur peut ainsi éviter d'avoir recours à d'autres méthodes de communication inter-processeurs. Leurs différences sont principalement technologiques : un système

multiprocesseur utilise plusieurs processeurs ayant chacun leur boîtier, tandis qu'un système multicœur regroupe sur un même boîtier plusieurs cœurs de processeur.

La parallélisation sur ce genre de systèmes permet d'obtenir un facteur accélérateur théoriquement limité au nombre de cœurs (en pratique, les communications entre les unités de traitement empêchent d'atteindre ce plafond). Les systèmes multicœur ainsi que les outils de programmation parallèle destinée à ces systèmes étant de plus en plus répandus, cette solution présente des avantages certains. La consommation des systèmes contenant un grand nombre de cœurs étant généralement élevée, il faudra être prudent quant au choix de ces systèmes, dans le cas de plate-formes embarquées. Des systèmes multicœur à faible consommation existent cependant, l'exemple le plus connu actuellement étant ARM avec sa série Cortex, qui propose des systèmes allant jusqu'à quatre cœurs. La limitation principale de ces systèmes se tient cependant dans le facteur accélérateur. Cette limitation tient principalement au nombre de cœurs déployés en parallèle, qui reste faible du fait de la jeunesse de cette technologie. Les évolutions attendues vers des systèmes "many-core" devraient permettre d'atténuer cette limitation. Si les traitements que l'on souhaite paralléliser ne peuvent être déployés sur un système multicœur dans le respect des contraintes du système, il faudra étudier les deux autres solutions présentées ici.

Les GPU sont des unités de calcul fortement SIMD, avec des jeux d'instructions réduits destinés à la manipulation de vecteurs et de matrices. Ces systèmes sont capables de traiter une grande quantité de données simultanément, leur gestion de la mémoire étant plus adaptée à la parallélisation que les GPP multicœur. Les particularités des GPU font qu'ils sont très utilisés pour le traitement d'image, au delà de leurs fonctions originelles qui se limitaient au rendu 3D et au décodage vidéo. Si ces systèmes, de par leur jeux d'instructions réduits, sont moins adaptés aux traitement généralistes que les GPP et systèmes multicœur, leur efficacité dépasse de loin ces derniers dans les traitements réguliers de grandes quantités de données. On arrive alors à des facteurs accélérateurs bien plus élevés qu'avec une simple parallélisation multicœur. Les GPU sont parfois assez coûteux en énergie électrique, et par conséquent relativement inadaptés à une utilisation embarquée, mais ce point particulier a beaucoup évolué au cours des dernières années.

Les ASIC et les FPGA sont les moyens les plus courants de concevoir une architecture spécifique dans les moindres détails. Un ASIC est une puce électronique fondue pour un système précis, avec une architecture fixe (il reste possible de déployer une partie reconfigurable sur un ASIC). Son avantage principal est sa vitesse, les limitations matérielles sont aussi levées par rapport aux FPGA. Le développement des ASIC passe généralement par une étape de prototypage fonctionnel sur FPGA.

Un FPGA est une puce électronique composée d'une grille d'éléments réguliers indépendants configurables, d'une interconnexion aussi configurable, et d'éléments supplémentaires propres au fondeur du FPGA. Le FPGA permet de déployer une architecture décrite dans un langage de description matérielle (VHDL ou Verilog), sur ce réseau d'éléments, en utilisant au besoin les mémoires, multiplieurs câblés, ou autres éléments propres au FPGA. La description matérielle pour ces deux cibles permet d'obtenir des facteurs d'accélération très élevés, selon le type de traitements et la méthode de développement utilisée (génération automatique de code, ou description manuelle).

Il est à noter que les traitements parfaitement adaptés aux GPU (notamment les calculs flottants) seront généralement plus rapides sur un GPU que sur une architecture dédiée sur FPGA, du fait des limitations technologiques du FPGA (cadence de fonctionnement plus basse que les GPU récents, ressources logiques utilisables dans un même FPGA, quantité de mémoire disponible, entre autres). Cependant, une gestion adaptée de la bande passante peut permettre une forte accélération par rapport aux GPU : l'interconnexion étant configurable aux plus bas niveaux du FPGA, un parallélisme de données interne bien plus important que celui d'un GPU est possible.

Des travaux ont déjà étudié le déploiement de traitements parallèles sur ces trois types d'architectures, afin de comparer les performances réelles de chacun. Dans (KD09), les FPGA Virtex 5 LX330T et Virtex 6 LX760 de Xilinx sont comparés à des GPU NVidia 9600GT et AMD 9270, au processeur CELL d'IBM de la Playstation 3, ainsi qu'à deux processeurs multicœur d'Intel, le Xeon 5160 et l'i7 965. Pour chaque cible, un programme nécessitant des calculs flottants sur des données massivement parallèle est déployé à l'aide d'outils de génération de code. La consommation électrique mesurée est aussi présentée. Les conclusions des auteurs sur la supériorité en puissance de calcul d'un GPU sur les autres architectures testées est à nuancer : l'étude porte sur le déploiement de calculs en virgule flottante, pour lesquels les FPGA sont reconnus pour être

effectivement assez peu efficaces. Un fait marquant en faveur des FPGA est la consommation électrique : les deux FPGA restent entre 20W et 30W, tandis que le GPU le plus économe en énergie testé ici nécessite entre 59W et 70W. L'utilisation d'un FPGA dans un système embarqué semble alors plus intéressant, du moment que les algorithmes peuvent être déployés sur un FPGA dans le respect des autres contraintes. Il est alors recommandé d'utiliser des données en entier ou en virgule fixe dès que possible, pour accélérer leur traitement de façon significative.

Dans une comparaison des FPGA et des GPU pour le domaine spécifique du traitement de flot optique (CNB+08), Chase et Al. remettent en question le choix des FPGA comme alternative aux solutions logicielles, place qu'ils occupaient depuis de nombreuses années. Les GPU sont ici évalués face à un système précédemment déployé sur FPGA (WLNM07).

Si les FPGA et ASIC permettent un contrôle extrêmement précis de l'architecture du système, la contrepartie est importante. Le déploiement d'une application sur un ASIC, et dans une moindre mesure sur un FPGA, prend généralement beaucoup plus de temps que sur une solution multiprocesseur ou GPU, et la modification d'une architecture précédemment validée et déployée sur un ensemble de plate-formes représente aussi un travail plus fastidieux. En règle générale, le recours à ce type de solution n'est envisagé que quand les plate-formes multiprocesseur et les GPU se révèlent trop peu puissants ou trop consommateurs en électricité pour satisfaire les contraintes du projet.

Chapitre 3

Exploration logicielle

3.1 Algorithmes

3.1.1 Norme de gradient 2D

L'opérateur gradient permet d'obtenir à partir d'une image, une information sur les variations de valeur entre les pixels.

Dans le cas de l'application de vision robotique étudiée, où seule la norme du gradient est utilisée, le choix de l'opérateur n'est pas définitif. Pour ce document, on optera pour l'opérateur de Sobel (figure 3.1), mais des tests ont été faits avec l'opérateur de Prewitt, et il est envisagé d'utiliser l'opérateur de Deriche.

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad B = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G = \mathrm{sqrt}((A*E)^2 + (B*E)^2)$$

FIGURE 3.1 – **Opérateur de Sobel** - Intensité de gradient de l'opérateur de Sobel. L'image d'intensité de gradient G est calculée à partir de l'image d'entrée E. Le symbole *** représente l'opérateur de convolution**

Les effets de bord sont réduits ici aux pixels du tour de l'image, pour lesquels on ne dispose pas des huit pixels limitrophes. On choisit une méthode simple pour gérer

ces effets de bord : pour le tour de l'image, le résultat de l'opérateur est simplement remplacé par une valeur nulle.

3.1.2 Pyramide Gaussienne

Une fois que l'intensité du gradient de l'image fournie par la caméra est obtenue, il reste à découper l'information fréquentielle de celle-là en bandes de fréquences. C'est ici qu'intervient la pyramide Gaussienne de Crowley et. al (CRP02).

Pour rappel, la décomposition en sous-blocs, présentée en section 1.2.2, est la suivante :

- filtrages Gaussiens successifs
- sous-échantillonnage à chaque octave (en fréquence spatiale)
- soustraction deux à deux des résultats de filtrage pour obtenir les DoG (Différences de Gaussiennes)

3.1.2.1 Filtrages Gaussiens

L'algorithme 1 représente le filtrage Gaussien par convolution avec un noyau de coefficients.

Les valeurs de σ qui sont utilisées pour la pyramide de l'application de vision sont celles proposées par Crowley et al., soit 1 et $\sqrt{2}$ selon la position du filtre dans la pyramide (figure 1.2). La formule de la fonction Gaussienne est $\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$. Dans cette formule, le coefficient $\frac{1}{2\pi\sigma^2}$ permet d'obtenir une surface de valeur 1 sous la courbe Gaussienne. Dans le cas d'une convolution avec un noyau de coefficients, la valeur de ce coefficient n'est théoriquement juste que pour un noyau de coefficients de dimensions infiniment élevées. Dans notre cas, la taille du noyau étant finie, la somme des coefficients n'atteint pas 1, s'ils sont calculés à partir de cette formule. Il faut donc passer par une étape de normalisation des coefficients. On calculera une version temporaire des coefficients du noyau avec la formule $e^{-\frac{x^2+y^2}{2\sigma^2}}$, afin de calculer leur somme. Chacun des coefficients temporaires est alors divisé par cette somme, afin que la somme des coefficients définitifs du noyau soit égale à 1. La formule adaptée de la fonction Gaussienne sera alors $G(x,y) = \frac{1}{\sum_{a=x_{min}}^{x_{max}} \sum_{b=y_{min}}^{y_{max}} e^{-\frac{a^2+b^2}{2\sigma^2}}} e^{-\frac{x^2+y^2}{2\sigma^2}}$.

Entrée : **E** : image de dimensions $H \times W$
G : noyau de coefficients Gaussien de dimensions $h \times w$ impaires, centré sur (0,0) (somme des coefficients = 1)
Sortie : **S** : image de dimensions $H \times W$

```
1  for Y ← 1 to H do
2      for X ← 1 to W do
3          S(X,Y) = 0
4          for y ← −(h − 1)/2 to (h − 1)/2 do
5              for x ← −(w − 1)/2 to (w − 1)/2 do
6                  S(X,Y) = S(X,Y) + E(X + x, Y + y) × G(x,y)
7              end
8          end
9      end
10 end
```

Algorithme 1: Filtrage Gaussien par convolution 2D, effets de bords non représentés

L'algorithme 1 présente un temps d'exécution relativement long, du fait des nombreuses multiplications effectuées pour chaque pixel de l'image de sortie. Le filtrage Gaussien 2D étant séparable en deux passes de convolution 1D (verticale + horizontale) sans perte de précision, il sera peut-être préférable de choisir cette solution.

L'algorithme 2 présente la convolution en 2 passes à 1 dimension : l'image subit d'abord une convolution avec une fenêtre d'un pixel de haut, et large de la dimension du noyau 2D, puis le résultat subit une autre convolution, cette fois avec la même fenêtre disposée verticalement (ces deux passes sont interchangeables). Le nombre de multiplications par pixel de l'image de sortie passe alors d'une complexité en $O(n^2)$ à une complexité en $O(n)$. Plus concrètement, une fenêtre de convolution carrée de taille 9 nécessite 81 multiplications (carré de 9×9 pixels) dans le cas de l'algorithme 1, alors qu'il n'en nécessite que 18 (deux lignes de 9 pixels) dans le cas de l'algorithme 2, soit moins du quart. On peut donc supposer que cette optimisation permettra d'accélérer fortement l'exécution, et permettra de réduire les ressources arithmétiques utilisées par le déploiement matériel. Cela dit, il reste à voir si cet algorithme peut être déployé en matériel sans trop de compromis sur le comportement temporel, ni sur l'utilisation de la mémoire. L'algorithme en 2 passes sera donc utilisé pour tout déploiement logiciel,

Entrée : **E** : image de dimensions $H \times W$
G : noyau de coefficients Gaussiens de dimensions w, centré sur (0) (somme des coefficients = 1)
Paramètres : $\boldsymbol{\sigma}$: écart-type du filtre Gaussien
Sortie : **S** : image de dimensions $H \times W$
Variables : **T** : tableau de valeurs temporaire de dimensions $H \times W$

```
1  for Y ← 1 to H do
2  |  for X ← 1 to W do
3  |  |  T(X,Y) = 0
4  |  |  for x ← −(w − 1)/2 to (w − 1)/2 do
5  |  |  |  T(X,Y) = T(X,Y) + E(X + x, Y) × G(x)
6  |  |  end
7  |  end
8  end
9  for Y ← 1 to H do
10 |  for X ← 1 to W do
11 |  |  S(X,Y) = 0
12 |  |  for y ← −(w − 1)/2 to (w − 1)/2 do
13 |  |  |  S(X,Y) = S(X,Y) + T(X, Y + y) × G(y)
14 |  |  end
15 |  end
16 end
```

Algorithme 2: Filtrage Gaussien par deux passes de convolution 1D

mais le premier algorithme reste une solution envisagée pour un éventuel déploiement matériel : il faudra comparer les déploiements des deux algorithmes afin de choisir le plus adapté.

3.1.2.2 Sous-échantillonnages

Entrée : **E** : image de dimensions $H \times W$
Sortie : **S** : image de dimensions $H/2 \times W/2$

```
1  for Y ← 1 to H/2 do
2      for X ← 1 to W/2 do
3          S(X,Y) = 0
4          for y ← 0 to 1 do
5              for x ← 0 to 1 do
6                  S(X,Y) = S(X,Y) + E(X+x, Y+y)
7              end
8          end
9          S(X,Y) = S(X,Y)/4
10     end
11 end
```

Algorithme 3: sous-échantillonnage de facteur 2

L'algorithme 3 correspond au sous-échantillonnage d'une image d'un facteur 2. L'information fréquentielle spatiale est alors divisée par deux.

3.1.2.3 Différences de Gaussiennes

Entrée : **E1** : image de dimensions $H \times W$
E2 : image de dimensions $H \times W$
Sortie : **S** : image de dimensions $H \times W$

```
1  for Y ← 1 to H do
2      for X ← 1 to W do
3          S(X,Y) = E1(X,Y) − E2(X,Y)
4      end
5  end
```

Algorithme 4: Différence de Gaussiennes

L'algorithme 4 décrit le fonctionnement de la différence de Gaussiennes. Les deux images d'entrée passent pixel par pixel dans un opérateur de soustraction pour générer l'image de sortie.

3.1.3 Recherche de points d'intérêt

L'algorithme 5 (version simplifiée pour des raisons de mise en page) permet d'identifier des maxima locaux dans une image de DoG.

```
   Entrée      : E : image de dimensions H×W
   Paramètres  : N : nombre maximal de points d'intérêt à trouver
                 γ : seuil de détection de point d'intérêt
                 R : rayon de recherche du maximum local
   Sortie      : X et Y : tableaux de coordonnées de dimension N
                 V : tableaux de valeurs de points d'intérêt de dimension N
                 n : nombre de points d'intérêt trouvés (entre 0 et N)
 1 n=0
 2 for y ← R+1 to H-R do
 3     for x ← R+1 to W-R do
 4         if E[x,y] ⩾ γ then
 5             if ∄ point d'intérêt déjà trouvé à R pixels ou moins de E[x,y] then
 6                 if E[x,y]⩾ tout pixel dans un rayon R autour de E[x,y] then
 7                     V=Tri(E[x,y],x,y,n)
 8                     if n < N then
 9                         n=n+1
10                     end
11                 end
12             end
13         end
14     end
15 end
```

Algorithme 5: Recherche de points d'intérêt dans une image

Le test de l'existence de points d'intérêt précedents dans le rayon d'inhibition se fait via la mesure de la distance aux coordonnées des points d'intérêt connus. La DoG est parcourue dans sa quasi-totalité, pixel par pixel. Seuls les pixels à moins de R pixels

du bord de l'image ne seront pas testés comme points d'intérêt. Pour chaque pixel de valeur supérieure au seuil γ, les pixels dans la zone circulaire de rayon R autour du pixel testé est comparée à ce dernier. Tout d'abord, l'algorithme vérifie qu'aucun pixel dans le disque de recherche n'ait déjà été identifié comme un point d'intérêt. La moitié du disque est parcourue pour ce test, l'autre moitié étant composée de pixels qui n'ont pas encore été testés. Ensuite, tous les pixels du disque de rayon R autour du point étudié doivent être inférieurs ou égaux au pixel étudié, afin que ce dernier soit définitivement considéré comme un point d'intérêt. Si c'est le cas, il est rajouté à la fin de la liste triée des points d'intérêts, et le tri de la liste est lancé.

La version logicielle utilisée pour la mesure des durées d'exécution est optimisée afin d'économiser une partie importante de l'étude de l'image d'entrée : la zone circulaire parcourue autour d'un point d'intérêt est délimitée à l'initialisation de l'application, afin d'économiser du temps de calcul sur ce parcours de zone largement récurrent durant l'étude d'une image. On stockera les frontières en x du disque, dans un tableau T adressé en y. Le parcours de l'image se fera de -R à R en y, et de -T[y] à T[y] en x pour chaque valeur de y.

3.1.4 Tri des points d'intérêt

Comme on a pu le voir dans la section précédente, les points d'intérêt sont triés au fur et à mesure qu'ils sont découverts dans l'algorithme de recherche. Le tri qui a été choisi est un tri à bulle. Il est assez peu efficace pour trier un tableau entièrement en désordre, mais sa rapidité de mise en œuvre, ainsi que les tailles de tableaux qu'il devra trier (le réseau de neurone utilisera rarement plus de 30 points d'intérêts par demi-échelle) font que l'optimisation de cet algorithme ne présente pas un besoin réel. On verra par la suite que la durée d'exécution du tri des points d'intérêt trouvés est suffisamment insignifiant pour se focaliser plutôt sur l'accélération des autres algorithmes.

Les points d'intérêt dans le tableau sont triés par valeur V des pixels en sortie de DoG, les points d'intérêt restant sous forme de bloc (V,x,y). Le paramètre N est pris en compte afin de limiter le nombre de points d'intérêt dans la liste triée. Ainsi, un nouveau point d'intérêt ne rentrera dans la liste que si la liste n'est pas encore de taille N, ou si la valeur du pixel du nouveau point d'intérêt est plus forte qu'au moins un point de la liste.

3.1.5 Mise en forme des voisinages de points d'intérêt

Le voisinage en anneau autour de chaque point d'intérêt doit être fourni au réseau de neurones sous forme log-polaire. Certaines variables sont utilisées pour la transformation log-polaire :

- Le tableau M, de mêmes dimensions que l'image log-polaire ($\rho_{max} \times \theta_{max}$). Il référence, pour chaque pixel de l'image log-polaire, le nombre de pixels correspondants dans l'anneau encerclant le point d'intérêt dans l'image d'entrée. Ainsi, si $M(4,8) = 19$, on comprend que dans l'image log-polaire, le pixel de coordonnées $\rho = 4$ et $\theta = 8$ sera calculé par la moyenne de 19 points de l'image cartésienne d'entrée.
- Les tableaux $x_{(\rho,\theta)}$ et $y_{(\rho,\theta)}$, de dimensions $\rho_{max} \times \theta_{max}$, contiennent pour chaque (ρ, θ), un tableau de $M(\rho, \theta)$ coordonnées respectivement en x et en y. Pour poursuivre l'exemple de M, $x_{(4,8)}$ sera un tableau de 19 coordonnées en x, et $y_{(4,8)}$ sera un tableau de 19 coordonnées en y.

Ces variables permettent pour tout point d'intérêt trouvé en (X, Y) dans la DoG E, d'extraire une imagette log-polaire S, de son voisinage en anneau de pixels. Cette image log-polaire est construite selon la formule $S(\rho,\theta) = \frac{1}{M(\rho,\theta)} \times \sum_{n=1}^{M(\rho,\theta)} E(X + x_{(\rho,\theta)}(n), Y + y_{(\rho,\theta)}(n))$.

L'algorithme 6 s'occupe de l'initialisation des variables M, x et y. Cette partie des traitements se fait une fois lors du lancement du programme, et ne sera relancée que si l'on souhaite donner à l'utilisateur (le réseau de neurones) la liberté de modifier en cours d'exécution les dimensions de l'image log-polaire, ou encore les dimensions de l'anneau délimitant le voisinage. Ce n'est pas le cas pour l'instant, le besoin de changer ces dimensions en cours d'exécution n'ayant pas été identifié.

L'algorithme fonctionne en deux parties : une première partie permet de calculer, pour tous les pixels de l'anneau dans l'image E, les coordonnées log-polaire (ρ, θ) correspondantes, et ainsi remplir les listes de coordonnées $x_{(\rho,\theta)}$ et $y_{(\rho,\theta)}$, tout en ajustant la valeur de $M(\rho, \theta)$. Dans certains cas, tous les pixels de l'image log-polaire ne seront pas remplis, surtout lorsqu'on se rapproche du rayon intérieur de l'anneau. C'est ici que la deuxième partie intervient, en calculant en sens inverse le pixel en coordonnées cartésiennes qui correspond le mieux aux pixels log-polaires laissés vides par la première

Paramètres : **R** : rayon autour du point d'intérêt délimitant le voisinage à extraire

P : dimensions de la zone aveugle au centre de l'image, en pourcentage de R

Variables : **D** : distance d'un pixel par rapport au centre de l'anneau

Sortie : $\boldsymbol{M_{(\rho,\theta)}}$: tableau de dimensions $\rho_{max} \times \theta_{max}$: nombre de pixels dans E correspondant à chaque pixel S(ρ, θ)

$\boldsymbol{x_{(\rho,\theta)}}, \boldsymbol{y_{(\rho,\theta)}}$: tableaux de dimensions $M_{max} \times \rho_{max} \times \theta_{max}$: coordonnées (x,y) des $M(\rho, \theta)$ pixels de E correspondant à chaque S(ρ, θ)

for $y \leftarrow -R$ **to** R **do**
  for $x \leftarrow -R$ **to** R **do**
    $D = \sqrt{x^2 + y^2}$
    if $D \leqslant R$ ***and*** $D \geqslant R \times P$ **then**
      if $x > 0$ **then**
        $\theta = -atan(y/x)$
      else
        $\theta = -atan(y/x) + \pi$
      end
      $\rho = \rho_{max} \times \frac{log(D) - log(R \times P)}{log(R) - log(R \times P)}$
      $x_{(\rho,\theta)}(M(\rho, \theta), \rho, \theta) = x$
      $y_{(\rho,\theta)}(M(\rho, \theta), \rho, \theta) = y$
      $M(\rho, \theta) = M(\rho, \theta) + 1$
    end
  end
end
for $\rho \leftarrow 0$ **to** $\rho_{max} - 1$ **do**
  for $\theta \leftarrow 0$ **to** $\theta_{max} - 1$ **do**
    if $M(\rho, \theta) = 0$ **then**
      $x_{(\rho,\theta)}(0, \rho, \theta) = cos(\frac{\theta \times 2\pi}{\theta_{max}}) \times e^{\frac{\rho \times (log(R) - log(R \times P))}{\rho max} + log(R \times P)}$
      $y_{(\rho,\theta)}(0, \rho, \theta) = sin(\frac{\theta \times 2\pi}{\theta_{max}}) \times e^{\frac{\rho \times (log(R) - log(R \times P))}{\rho max} + log(R \times P)}$
      $M(\rho, \theta) = 1$
    end
  end
end

Algorithme 6: Transformée log-polaire : Initialisation des variables

partie. Les pixels dont les coordonnées cartésiennes sont trouvées par cette deuxième partie interviennent déjà dans le calcul d'un autre pixel log-polaire, ce fonctionnement implique donc une redondance dans l'image log-polaire. Une fois ces deux parties effectuées, les variables permettant le calcul de l'image log-polaire sont utilisables.

L'algorithme 7 représente le calcul des pixels d'une imagette correspondant à un point d'intérêt $E(X, Y)$, en suivant la formule présentée au début de cette section. Pour chaque $S(\rho, \theta)$, la somme des $M(\rho, \theta)$ pixels est tout d'abord calculée par addi-

Entrée : **E** : image de dimensions $w \times h$
(X,Y) : Coordonnées du point d'intérêt autour duquel E est centrée
Sortie : **S** : image de dimensions $\rho_{max} \times \theta_{max}$
Variables : **M** : tableau de dimensions $\rho_{max} \times \theta_{max}$: nombre de pixels dans E correspondant à chaque pixel (ρ, θ) S
$\boldsymbol{x_{(\rho,\theta)}}, \boldsymbol{y_{(\rho,\theta)}}$: tableaux de dimensions $M_{max} \times \rho_{max} \times \theta_{max}$: coordonnées (x,y) des pixels de E correspondant à chaque pixel (ρ, θ) de S
Somme : somme des pixels de E correspondant à un même pixel de S

```
1 for ρ ← 0 to ρ_max − 1 do
2     for θ ← 0 to θ_max − 1 do
3         Somme = 0
4         for m ← 0 to M(ρ, θ) − 1 do
5             Somme = Somme + E(X + x_(ρ,θ)(m, ρ, θ), Y + y_(ρ,θ)(m, ρ, θ))
6             S(ρ, θ) = Somme/M(ρ, θ)
7         end
8     end
9 end
```

Algorithme 7: Transformée log-polaire d'une imagette carrée

tions successives, puis divisée par $M(\rho, \theta)$.

3.2 Caractérisation de l'application

Afin de déployer l'application sur un SoC, il est utile de commencer par obtenir une première version fonctionnelle de l'application, quelle que soit la cible. Un déploiement logiciel étant généralement plus rapide à mettre en œuvre, plus facile à contrôler et à caractériser qu'un déploiement matériel, la première étape du déploiement du SoC sera logicielle. En déployant l'application en logiciel et en étudiant son comportement, il sera plus facile d'orienter le déploiement du SoC de façon générale.

3.2.1 Durées d'exécution sur PC

Une version logicielle sur un ordinateur traditionnel permet d'établir un prototype fonctionnel très rapidement. Le prototype ne respectera pas forcément les contraintes qui sont données à l'application, mais permettra d'effectuer plusieurs mesures utiles pour la conception du SoC.

Le langage de programmation choisi influera fortement sur la suite des opérations : un langage utilisable sur un processeur embarqué sans trop de modifications facilitera l'étape de portage, tandis que d'autres langages permettront un prototypage logiciel plus rapide, une simulation plus facile, ou encore des mesures plus précises. Si un ou plusieurs microcontrôleurs embarqués font partie des cibles éventuelles, un programme en C est généralement une solution idéale, du moment que l'on s'abstient d'utiliser des bibliothèques particulières indisponibles pour ces microcontrôleurs. Dans le cas du SoC de vision, le langage C est choisi pour sa rapidité d'exécution et sa portabilité : il est utilisable directement avec la totalité des microcontrôleurs embarqués envisagés dans le projet (NIOS II, MicroBlaze, LEON3, PowerPC, Cortex A8/A9).

Le développement en C du prototype se fait tout d'abord en utilisant pour les résultats des traitements, le type de données utilisé par l'équipe Neurocybernétique, à savoir le **float**, qui permet une grande précision des résultats, en comparaison aux entiers et nombres à virgule fixe. Ce type de données n'étant pas adapté à tous les processeurs embarqués, et encore moins au déploiement matériel sous forme d'IP, il pourra être raffiné ultérieurement en faveur de types de données en virgule fixe, voire entières. Une fois l'exécutable généré, le programme est caractérisé pour chaque fonction : on mesure leurs durées d'exécution moyennes ainsi que le nombre de fois où elles seront appelées

par le programme.

Le programme est écrit sous forme de fonctions représentant chacune un bloc de traitement de l'application de vision présentée en figure 1.2. Il y a une fonction pour les opérateurs : gradient, filtrage Gaussien de coefficient $\sigma = 1$, filtrage Gaussien de coefficient $\sigma = \sqrt{2}$, DoG, sous-échantillonnage de facteur 2, recherche de points d'intérêt, tri de points d'intérêts, et enfin transformation log-polaire des voisinages des points d'intérêt. Les fonctions reçoivent en argument les adresses de leurs entrée(s) et sortie, il est ainsi facile d'utiliser la même fonction sur différents résultats intermédiaires.

Les mesures de durées d'exécution sur PC sont réalisées à l'aide de l'outil *gprof*, qui mesure pour chaque fonction d'un programme, le nombre d'appels, la durée totale d'exécution, et la durée moyenne d'un appel à cette fonction. Pour ces mesures, les fonctions récurrentes (toutes sauf le gradient) ont été dupliquées et nommées de façon à correspondre à chaque bloc individuel de l'application (voir figure 1.2). Il y aura donc par exemple 6 fonctions DoG pour les mesures, toutes les 6 ayant un nom permettant de les identifier.

Une série de 5964 images (vidéo de 3m58s à 25 images par seconde) est utilisée en entrée du programme afin d'augmenter la qualité de l'information fournie par les mesures. Ces images sont extraites d'une vidéo capturée par le robot durant une navigation en intérieur, avec une version précédente de l'application de vision. Une particularité intéressante d'un tel échantillon de test est qu'il permet d'étudier l'évolution temporelle des mesures au fil de la navigation du robot, dans un environnement visuel représentatif d'un cas réel.

Un déploiement en virgule fixe (type de données : short) a été réalisé, afin de noter l'influence du type de données sur ce type de processeur GPP. En effet, le choix du float comme type de données pour stocker les images intermédiaires est à éviter pour certains des processeurs embarqués envisagés : sur un processeur Soft-Core, la présence d'une unité de traitement en virgule flottante augmente les ressources utilisées sur le FPGA. On mesure les différences de durées d'exécution entre les deux versions sur PC

pour comparaison. Les durées d'exécution relevées à l'aide de gprof sont présentées en table 3.1.

Les mesures ont été réalisées sur un ordinateur portable aux caractéristiques suivantes : processeur Intel T2300 @1.66GHz, 2Go de RAM DDR2, Arch Linux sans interface graphique, noyau linux 3.0.

On remarque qu'il y a une différence assez faible entre les durées d'exécution des deux versions de l'application. Si le choix du float ou du short pour un ordinateur portable importe peu, il est important de mesurer les différences entre les deux versions sur les autres plate-formes. Les différences architecturales entre le PC et l'éventail des plate-formes embarquées envisagées sont importantes. En effet, si un processeur grand public comme celui du PC portable traite à une vitesse similaire les nombres à virgules flottantes et les nombres à virgule fixe, un déploiement matériel à fort parallélisme en virgule flottante sera bien plus lourd qu'un déploiement équivalent en virgule fixe.

Dans le cas de l'utilisation de processeur(s) embarqué(s) sur le SoC, il est toutefois intéressant d'interpréter les mesures effectuées sur le PC. Même si la durée d'exécution n'est pas représentative des performances que l'on pourra obtenir sur un processeur embarqué, on pourra commencer par éliminer les fonctions qui sont déjà trop coûteuses en temps d'exécution. Déployer ces fonctions sur un des processeur embarqué envisagés, beaucoup moins performants, entraînerait avec certitude un non-respect des contraintes temps-réel sur le SoC.

Le tableau 3.1 permet déjà d'identifier quelles parties de l'application devront nécessairement être accélérées : sans surprise, les filtrages Gaussiens sont de loin les plus représentatifs. Il est à noter que l'algorithme choisi pour ce déploiement est l'algorithme 2 (section 3.1.2.1), qui est déjà une version accélérée de l'algorithme 1. Le tableau 3.2 permet par ailleurs de prendre conscience de l'accélération de ce choix d'algorithme.

On remarque une forte accélération des filtrages Gaussiens, ainsi que de l'application dans sa globalité : l'application passe d'une cadence de traitement de moins de 7 images par seconde à une cadence de 25 images par secondes. Le facteur d'accélération de l'application est ainsi d'environ 3,6, les filtrages Gaussiens en eux-même étant accélérés d'un facteur 4,7.

Fonctions	Durée moyenne virgule fixe	Durée moyenne virgule flottante
Gradient	**1.08 ms**	**1.09 ms**
Gaussienne $\sigma = 1$, HF 1	**6.88 ms**	**7.08 ms**
Gaussienne $\sigma = 1$, HF 2	**7.26 ms**	**7.02 ms**
Gaussienne $\sigma = \sqrt{2}$, HF	**8.85 ms**	**8.93 ms**
Gaussienne $\sigma = 1$, MF	**1.57 ms**	**1.67 ms**
Gaussienne $\sigma = \sqrt{2}$, MF	**2.14 ms**	**2.10 ms**
Gaussienne $\sigma = 1$, BF	0.43 ms	0.40 ms
Gaussienne $\sigma = \sqrt{2}$, BF	0.58 ms	0.52 ms
Sous-échantillonnage HF $\rightarrow$ MF	0.17 ms	0.16 ms
Sous-échantillonnage MF $\rightarrow$ BF	0.03 ms	0.03 ms
Différence de Gaussiennes, HF 1	0.43 ms	0.44 ms
Différence de Gaussiennes, HF 2	0.40 ms	0.43 ms
Différence de Gaussiennes, MF 1	0.08 ms	0.08 ms
Différence de Gaussiennes, MF 2	0.11 ms	0.08 ms
Différence de Gaussiennes, BF 1	0.04 ms	0.02 ms
Différence de Gaussiennes, BF 2	0.02 ms	0.02 ms
Recherche de points d'intérêt, HF 1	**3.21 ms**	**3.29 ms**
Recherche de points d'intérêt, HF 2	**3.83 ms**	**3.96 ms**
Recherche de points d'intérêt, MF 1	0.50 ms	0.52 ms
Recherche de points d'intérêt, MF 2	0.55 ms	0.62 ms
Recherche de points d'intérêt, BF 1	0.11 ms	0.12 ms
Recherche de points d'intérêt, BF 2	0.14 ms	0.14 ms
Extraction de voisinages, HF 1	0.29 ms	0.40 ms
Extraction de voisinages, HF 2	0.33 ms	0.36 ms
Extraction de voisinages, MF 1	0.12 ms	0.13 ms
Extraction de voisinages, MF 2	0.11 ms	0.12 ms
Extraction de voisinages, BF 1	0.07 ms	0.08 ms
Extraction de voisinages, BF 2	0.08 ms	0.07 ms
Application complète	**39.42 ms**	**39.88 ms**
Recherches + extractions	**9.34 ms**	**9.81 ms**
Recherches	**8.34 ms**	**8.65 ms**
Extractions	**1.00 ms**	**1.16 ms**

Table 3.1 – **Durées d'exécution sur un ordinateur portable** - Durées moyennes par image, mesurées sur 5964 images échantillon de 192×144 pixels

Fonctions	Convolution 1 passe 2D	Convolution 2 passes 1D
Gaussienne $\sigma = 1$, HF 1	29.17 ms	6.88 ms
Gaussienne $\sigma = 1$, HF 2	30.45 ms	7.26 ms
Gaussienne $\sigma = \sqrt{2}$, HF	48.63 ms	8.85 ms
Gaussienne $\sigma = 1$, MF	7.21 ms	1.57 ms
Gaussienne $\sigma = \sqrt{2}$, MF	12.07 ms	2.14 ms
Gaussienne $\sigma = 1$, BF	1.75 ms	0.43 ms
Gaussienne $\sigma = \sqrt{2}$, BF	2.94 ms	0.58 ms
Filtrages Gaussiens	132.22 ms	27.71 ms
Application complète	143.88 ms	39.42 ms

TABLE 3.2 – **Durées d'exécution : filtrages Gaussiens 1D et 2D** - Durées moyennes par image, mesurées sur 5964 images échantillon de 192×144 pixels

Ces performances sont à nuancer, ces mesures ayant été réalisées sur une vidéo de test de résolution 192×144. Il serait regrettable de devoir se limiter à une si faible résolution quand la plupart des caméras récentes ont pour résolution minimale 640×480 pixels. Le robot tirerait parti d'informations visuelles plus précises avec un flux d'images de 640×480 pixels à 25 images par seconde. Pour cette résolution, on estime que le débit chuterait en dessous de 2,5 images par seconde, ce qui serait inacceptable.

Les filtrages Gaussiens restant longs à exécuter sur PC portable, il est raisonnable de penser qu'il ne sera pas possible d'exécuter cette application à une résolution plus élevée dans le respect des contraintes temps-réel de l'application, encore moins sur un processeur embarqué. Le déploiement sur PC portable étant lui-même difficilement justifiable sur le robot, surtout en terme d'encombrement, il faudra trouver une solution permettant l'accélération des traitements les plus longs.

Les éléments les plus rapides, à savoir les sous-échantillonnages et les différences de Gaussiennes, méritent une réflexion plus approfondie avant de figer le choix d'une architecture du SoC. On devra notamment se pencher sur l'aspect de communication des données dans le SoC : il est courant de se retrouver face à une congestion de bus quand on cherche à déployer en logiciel des traitements d'images travaillant à des résolutions conséquentes, et si le cas présent reste dans des résolutions modestes, le nombre de traitements effectués sur chaque image est assez grand : du gradient aux DoG, il y aura 16 lectures et 16 écritures d'images. À cela se rajoutent les lectures et écritures des

recherches et des tris des points d'intérêts, et de la mise en forme des voisinages de ces derniers.

3.2.2 Durées d'exécution sur processeur embarqué

Au vu des résultats du déploiement sur PC, une partie de l'application pourra être portée sur un processeur embarqué, tandis que certains traitements devront être fortement accélérés. Il est aussi envisageable d'héberger une partie des traitements logiciels sur le PC portable du robot, selon la durée d'exécution sur PC, si le déploiement sur FPGA s'avère problématique.

Comme on a pu le voir dans la section précédente, l'existence de certains traitements dont la rapidité d'exécution est satisfaisante, permet d'envisager l'évaluation d'un déploiement sur processeur embarqué (conception de la plate-forme + portage du code). Cela dit, même un ordinateur classique simple cœur ne suffit pas à exécuter la totalité de l'application dans le respect des contraintes temporelles. Il devient alors intéressant d'étudier les solutions multiprocesseur et les accélérateurs matériels connectés au processeur embarqué.

Une plate-forme de tests est choisie pour mesurer les durées d'exécution sur un processeur embarqué. L'éventail des processeurs embarqués envisagés est assez large, du MicroBlaze au Cortex A9, si bien qu'il est difficile de trouver une plate-forme de mesures embarquée dont les performances soient comparables aux autres. Des mesures sur un processeur soft-core nous permettraient d'explorer les architectures MPSoC, plus faciles à mettre en œuvre à l'aide de processeurs soft-core. Des mesures réalisées sur un Cortex A8 permettent quant à elles l'évaluation de processeurs embarqués puissants.

Un téléphone portable sous Linux avec un processeur Cortex A8 à 1GHz (Nokia N900, sous Maemo Linux, circuit TI OMAP 3430) a permis d'obtenir ces mesures (à l'aide de l'outil OProfile), sur le même échantillon d'images que la version PC. Le portage du code sur cette version est assez direct, la bibliothèque logicielle utilisée pour gérer les images étant la même sur PC et sur le téléphone : OpenCV. Les mesures présentées excluant les lectures et écritures de fichiers (images PNG), la différence éventuelle des versions d'OpenCV sur PC et sur N900 ne serait pas retranscrite dans ces résultats, cette bibliothèque n'étant utilisée que pour la gestion des fichiers images.

Il est toutefois important de remarquer la différence entre un Cortex A8 sur un système dédié, disposant potentiellement de mémoire DDR3, exécutant uniquement le programme de vision et de communication avec le réseau de neurones, et un processeur de téléphone portable, exécutant un OS et des tâches de fond, ne disposant que de mémoire mobile DDR. La plate-forme de tests utilisée ici permet d'avoir un aperçu de la puissance d'un tel processeur, en comparaison d'un processeur soft-core type NIOS, mais on peut s'attendre à de meilleures performances sur un système dédié.

Les résultats de mesure obtenus sur cette plate-forme sont présentés en table 3.3. On remarque que la durée d'exécution augmente de façon significative par rapport à une exécution sur PC, ce qui renforce le besoin d'accélération matérielle. De plus, même si l'on suppose que l'on accélère le gradient et la pyramide Gaussienne, les recherches de points d'intérêt et les extractions de leurs voisinages restent longues à exécuter (rappel : pour traiter 25 images par seconde, l'application complète doit s'exécuter en 40 ms au plus). Ces résultats montrent que l'utilisation d'un simple Cortex A8 est insuffisante pour cette partie de l'application, dès que l'on souhaite utiliser des résolutions plus réalistes que 192×144. Par exemple, même deux Cortex A8 avec un code parfaitement parallélisé (charge de travail équilibrée sur les deux processeurs) ne permettraient pas au SoC de traiter 25 images par seconde à une résolution de 320×240, ce qui est pourtant une résolution modeste.

Même la solution envisagée plus tôt, d'utiliser un cortex A9 double cœur tel qu'on peut en trouver dans le Zynq de Xilinx, semble maintenant insuffisante. Pour exécuter en logiciel les recherches, tris, et extractions, il faudra des processeurs bien plus puissants, ce qui n'est pas envisageable actuellement au sein du SoC embarqué.

Les recherches de points d'intérêt étudiant la totalité d'une DoG, leur durée d'exécution dépend directement de la résolution de la DoG étudiée. Il est indispensable de déployer ces recherches sous forme d'IP, si l'on souhaite utiliser des résolutions d'image réalistes. Les recherches de points d'intérêt devront par conséquent être déployées en matériel.

Les 11,52 ms des "extractions" mesurées sur le Cortex A8 correspondent à la durée d'extraction et de mise en forme log-polaire des voisinages en anneau des N points d'intérêt des six DoG (ici, N=20). Cette durée n'est pas directement dépendante de la résolution de l'image, mais des dimensions des voisinages des points à extraire (rayon R, dimensions de l'image log-polaire ρ_{max} et θ_{max}), et leur nombre par demi-échelle

Fonctions	Durée moyenne Cortex A8 1GHz	Durée moyenne sur PC (Rappel)
Gradient	**11,106 ms**	**1,09 ms**
Gaussienne $\sigma = 1$, HF 1	**37,763 ms**	**7,08 ms**
Gaussienne $\sigma = 1$, HF 2	**37,471 ms**	**7,02 ms**
Gaussienne $\sigma = \sqrt{2}$, HF	**46,366 ms**	**8,93 ms**
Gaussienne $\sigma = 1$, MF	**8,868 ms**	**1,67 ms**
Gaussienne $\sigma = \sqrt{2}$, MF	**10,714 ms**	**2,10 ms**
Gaussienne $\sigma = 1$, BF	2,039 ms	0,40 ms
Gaussienne $\sigma = \sqrt{2}$, BF	2,516 ms	0,52 ms
Sous-échantillonnage HF $\rightarrow$ MF	1,131 ms	0,16 ms
Sous-échantillonnage MF $\rightarrow$ BF	0,195 ms	0,03 ms
Différence de Gaussiennes, HF 1	3,889 ms	0,44 ms
Différence de Gaussiennes, HF 2	3,841 ms	0,43 ms
Différence de Gaussiennes, MF 1	0,587 ms	0,08 ms
Différence de Gaussiennes, MF 2	0,561 ms	0,08 ms
Différence de Gaussiennes, BF 1	0,126 ms	0,02 ms
Différence de Gaussiennes, BF 2	0,118 ms	0,02 ms
Recherche de points d'intérêt, HF 1	**10,066 ms**	**3,29 ms**
Recherche de points d'intérêt, HF 2	**10,512 ms**	**3,96 ms**
Recherche de points d'intérêt, MF 1	2,684 ms	0,52 ms
Recherche de points d'intérêt, MF 2	2,627 ms	0,62 ms
Recherche de points d'intérêt, BF 1	0,804 ms	0,12 ms
Recherche de points d'intérêt, BF 2	0,586 ms	0,14 ms
Extraction de voisinages, HF 1	4,232 ms	0,40 ms
Extraction de voisinages, HF 2	4,165 ms	0,36 ms
Extraction de voisinages, MF 1	1,094 ms	0,13 ms
Extraction de voisinages, MF 2	0,895 ms	0,12 ms
Extraction de voisinages, BF 1	0,588 ms	0,08 ms
Extraction de voisinages, BF 2	0,542 ms	0,07 ms
Application complète	**215,21 ms**	**39,88 ms**
Recherches/tris/extractions	**38,80 ms**	**9,81 ms**
Extractions	**11,52 ms**	**1,16 ms**

Table 3.3 – **Durées d'exécution sur un processeur embarqué** - Durées moyennes par image, mesurées sur 5964 images échantillon de 192×144 pixels, sur un ARM Cortex A8 à 1GHz

(N). Les extractions de voisinages sont par conséquent les traitements les plus adaptés à un déploiement logiciel. Il ne reste plus qu'à choisir le type d'unité de calcul qui sera responsable de leur exécution : processeur embarqué, soft-core ou hard core, ou encore le PC qui utilise le SoC ?

En fonction des besoins du robot, la résolution de l'image sera modifiée, ayant un impact sur les IP de la partie matérielle seulement. Les dimensions des anneaux de pixels correspondant aux voisinages, ainsi que les dimensions des imagettes log-polaires, influeront directement sur le code logiciel embarqué et sa durée d'exécution, en plus d'influer sur les IP de recherche de points d'intérêt.

3.2.3 Parallélisation logicielle embarquée

Dans le cas où la caméra continuerait de fournir des images à une résolution de 192×144 pixels, la parallélisation logicielle sur des processeurs soft-core reste une solution attrayante : le déploiement d'IP peut s'avérer plus problématique que la parallélisation logicielle dans le cas de certains algorithme. Il n'est pas toujours pratique de restreindre le choix des puces aux seuls FPGA dotés de processeurs puissants (Zynq et Cortex A9), pour des raisons de disponibilité de cartes de développement par exemple. On se limite dans le cas présent à des processeurs moins puissants que celui du Zynq, que l'on pourra tester sur des cartes actuellement disponibles. Les durées d'exécution mesurées sur le Cortex A8 ne seraient pas reproductibles sur un processeur soft-core actuel de type MicroBlaze. C'est la raison pour laquelle un déploiement sur FPGA avec une partie logicielle ne peut se faire sur un seul processeur Soft-Core : la parallélisation des traitements est indispensable. L'utilisation de processeurs soft-core présente l'avantage d'être très flexible, vis-à-vis du nombre de processeurs que l'on peut déployer sur la puce.

Afin d'évaluer les vitesses théoriques de divers schémas de parallélisation, un outil de modélisation multiOS a été utilisé. Cet outil, le framework développé par Emmanuel Huck au cours du projet OveRSoC (HMV07), a été utilisé pour explorer une architecture multiprocesseur.

Le framework, écrit en SystemC, permet de modéliser un système multiprocesseur, multiOS (chaque processeur ayant un OS indépendant), à partir d'un partitionnement manuel et de l'injection des durées d'exécution des différentes parties du programme.

Dans le cas de l'application de vision, un partitionnement naturel s'impose : les traitements des 6 DoG (recherche, tri de points d'intérêt et mise en forme de leurs voisinages) forment six flots de données indépendants. On peut donc partitionner l'application selon ces 6 éléments indépendants, ce qui rend le partitionnement bien plus simple. Un partitionnement plus fin serait possible, mais ajouterait de la complexité aux algorithmes et nécessiterait des communications inter-processeur. Si la conception du SoC n'en est pas rendue impossible, il est préférable d'étudier en priorité les déploiements les plus simples.

Les durées d'exécution qui sont injectées dans le modèle sont issues de mesures sur un processeur soft-core : le NIOS II d'Altera. Le comportement temporel simulé par le framework correspond donc à un système composé de plusieurs processeurs de ce type. Si ce type d'architecture multiprocesseur ne correspond pas forcément à l'architecture exacte qui sera utilisée dans le SoC définitif, elle permet toutefois d'étudier la pertinence d'une parallélisation sur des processeurs soft-core.

Comme on a pu le voir, la pyramide Gaussienne devra dans tous les cas être accélérée à l'aide d'IP matérielles. Cette partie de l'application n'est donc pas étudiée dans le modèle d'architecture logicielle. La partie recherche/tri/mise en forme, en revanche, sera évaluée. C'est cette partie de l'application que l'on souhaite paralléliser, étant donné qu'elle reste trop lourde pour un seul processeur soft-core, et qu'un processeur puissant comme le cortex A8 ou A9, capable de respecter les contraintes temporelles, n'est pas forcément envisageable. L'application semble a priori assez légère pour se paralléliser sur un nombre raisonnable de processeurs embarqués.

La figure 3.2 représente la durée d'exécution de cette partie de l'application en fonction du nombre de processeurs déployés sur le système. Ces résultats sont obtenus par le framework à l'aide du partitionnement manuel naturel (par flots de données). On voit qu'à partir de 2 processeurs, la durée d'exécution totale (donnée par la courbe la plus haute) sur le processeur le plus chargé (processeur 2) cesse de décroître. La raison est que l'étude d'une des deux DoG de haute fréquence (seule sortie de DoG étudiée par le processeur 2) prend plus de temps que l'étude de toutes les autres DoG. Deux conclusions émergent de cette modélisation : le partitionnement logiciel naturel n'est pas efficace au delà de deux processeurs. Un partitionnement logiciel sur NIOS II, permettant d'atteindre le respect des contraintes temporelles, nécessiterait plus de 90 processeurs (3,6s/90 = 40ms). Ce chiffre démontre que les processeurs NIOS II, et par

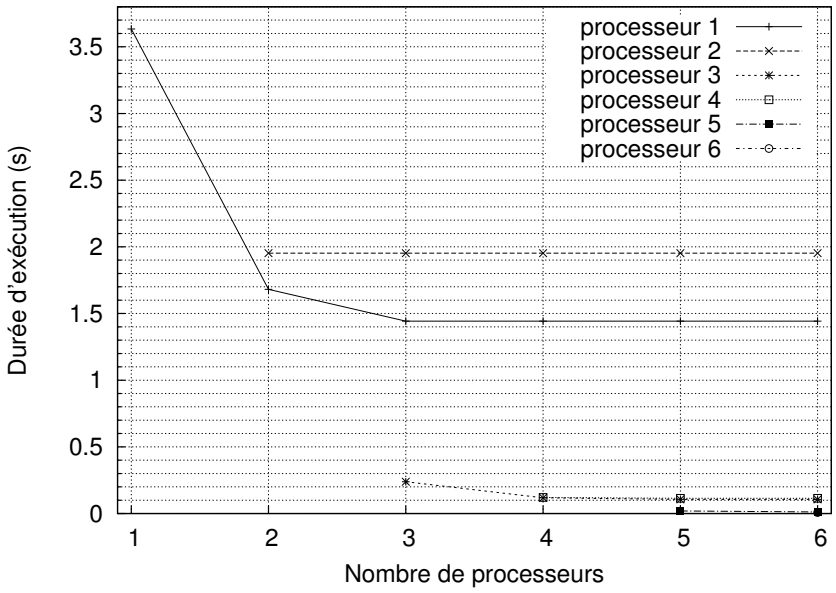


FIGURE 3.2 – **Parallélisation sur processeurs embarqués : recherches et extractions des points d'intérêt** - Exploration architecturale multiprocesseur NIOS à 100MHz, partitionnement naturel

extension des soft-core actuels, ne permettent pas l'exécution des recherches et extractions de voisinages de points d'intérêt. Si on étudie la parallélisation des extractions de voisinages seules (figure 3.3), on peut voir que la contrainte de 25 images par seconde n'est toujours pas respectée.

Il est alors envisagé d'exécuter les extractions de voisinages des points d'intérêt sur le PC embarqué du robot, ces traitements étant très rapides sur un PC portable. Les extractions des voisinages de 20 points d'intérêt dans chacune des 6 DoG, quelle que soit leur résolution, dure 2,53ms pour R=[20,10,5], ρ_{max}=36 et θ_{max}=5, sur un PC portable doté d'un processeur Intel T2300 @1.66GHz, 2Go de RAM DDR2, Arch Linux sans interface graphique, noyau linux 3.0. Le programme logiciel ayant fourni ce résultat de 2,53ms n'est pas parallélisé sur les deux cœurs du processeur. Même sur ce type de PC assez peu récent, les ressources de calcul ne sont ainsi utilisées qu'à 3,1% (2,53ms pour un seul cœur, toutes les 40ms).

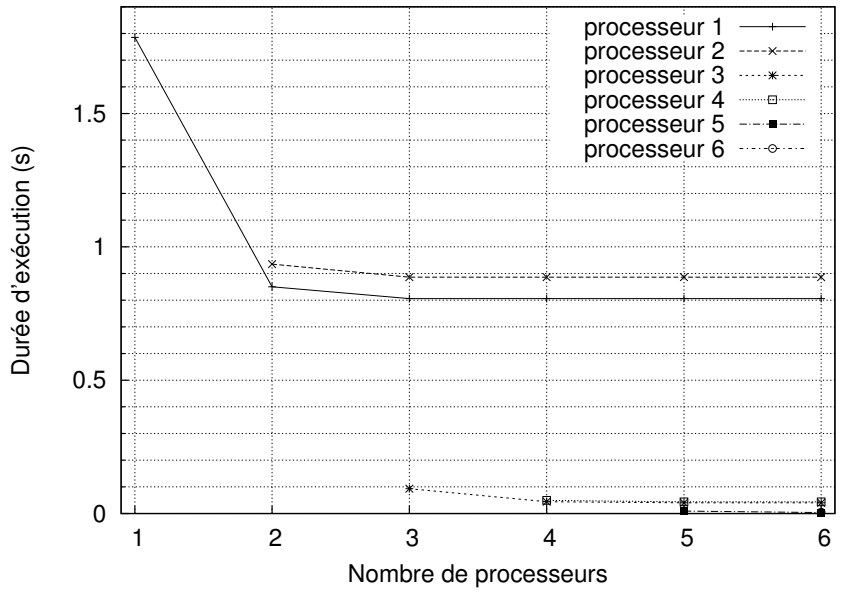


Figure 3.3 – **Parallélisation sur processeurs embarqués : extractions de voisinages des points d'intérêt** - Exploration architecturale multiprocesseur NIOS à 100MHz, partitionnement naturel

3.2.4 Charge des canaux de communication

Dans la conception d'un SoC utilisant des bus de communication partagés, le débit de données généré par l'application est essentiel à prendre en compte. Une connaissance insuffisante de la charge des canaux de communication peut en effet dissimuler une congestion, ce qui peut fortement ralentir l'exécution de l'application. Les contraintes temporelles du SoC risqueraient dans ce cas de ne pas être respectées.

Pour concevoir l'architecture du SoC, il convient de dimensionner les canaux de communication, voire de changer le partitionnement logiciel/matériel de l'application, en accord avec les débits de données entrant et sortant de chaque bloc fonctionnel, et la charge supportée par les canaux envisagés.

Le déploiement sous forme d'IP matérielles de parties consécutives des traitements permet de réduire la charge de ces canaux : les données concernées transiteront d'une IP à l'autre par des interconnexions dédiées. Plus grand sera le nombre de blocs fonctionnels connectés entre eux directement, moins les canaux de communication partagés

seront utilisés, réduisant ainsi les risques de congestion.

Le débit de données total entre les blocs fonctionnels sur le SoC dépend de plusieurs jeux de paramètres. Pour les transmissions d'images complètes, soit de la caméra aux sorties des DoG, la taille des images et le nombre d'images étudiées par seconde influence proportionnellement la charge des canaux de communication. Ces deux paramètres sont réglables et leur réduction permettrait d'alléger la charge des canaux de communication, mais cette solution nuit évidemment à la qualité des informations générées par le SoC. Pour les recherches de points d'intérêt et l'extraction de leurs voisinages, les paramètres N, R, ρ_{max}, θ_{max}, ainsi que le nombre d'images traitées par seconde, sont responsables de la charge des canaux.

Idéalement, on souhaite pouvoir étudier des images de grandes tailles à une cadence de 25 images par seconde, tout en fournissant au réseau de neurones des imagettes log-polaire de taille adéquate, correspondant à des zones circulaires de taille pertinente autour d'un nombre pertinent de points d'intérêt. Un SoC permettant un fonctionnement optimal du robot doit pouvoir supporter les débits générés par ces paramètres.

La table 3.4 référence les quantités de données lues et écrites à chaque image de la caméra, par les blocs fonctionnels de l'application, ainsi que les débits correspondants pour une cadence de 25 images par seconde. On suppose que les données sont stockées sur 16 bits, à part les données lues par le bloc gradient (la caméra fournit une image en niveau de gris, 8 bits par pixel), et les sorties en coordonnées log-polaires, à 32 bits par pixel. Les paramètres influant sur ces débits ont été fixés : N=20, R=20 en haute fréquence, R=10 en moyenne fréquence, et R=5 en basse fréquence, ρ_{max}=5, θ_{max}=36.

Si l'on part du postulat que la caméra est connectée directement au SoC, les données responsables de la charge du réseau Ethernet sont les voisinages des points d'intérêt, qui sont envoyés au réseau de neurones du robot. Pour les paramètres indiqués, et quelle que soit la taille de l'image, le débit résultant sur le réseau Ethernet serait d'environ 720 ko/s, ce qui est tout à fait raisonnable. Si les images des DoG devaient être envoyées au robot, le débit nécessaire serait d'environ 10 Mo/s

On peut voir que les traitements de l'application ont une bande passante totale de 72 Mo/s pour une image de 320×240. La charge est quasiment proportionnelle au nombre de pixels, les recherches et extractions ne générant qu'une petite partie de la charge.

Blocs Fonctionnels	**Données lues (ko)**	**Données écrites (ko)**	**Débit I/O à 25 fps (ko/s)**
Gradient	76,8	153,6	5760
Gaussienne HF (×3)	153,6 (×3)	153,6 (×3)	7680 (×3)
Gaussienne MF (×2)	38,4 (×2)	38,4 (×2)	1920 (×2)
Gaussienne BF (×2)	9,6 (×2)	9,6 (×2)	480 (×2)
Sous-échantillonnage HF → MF	153,6	38,4	4800
Sous-échantillonnage MF → BF	38,4	9,6	1200
DoG HF (×2)	307,2 (×2)	153,6 (×2)	11520 (×2)
DoG MF (×2)	76,8 (×2)	38,4 (×2)	2880 (×2)
DoG BF (×2)	19,2 (×2)	9,6 (×2)	720 (×2)
Recherche HF (×2)	153,6 (×2)	1,2 (×2)	3870 (×2)
Recherche MF (×2)	38,4 (×2)	1,2 (×2)	990 (×2)
Recherche BF (×2)	9,6 (×2)	1,2 (×2)	270 (×2)
Extraction HF (×2)	51,5 (×2)	14,4 (×2)	1647 (×2)
Extraction MF (×2)	13,8 (×2)	14,4 (×2)	704 (×2)
Extraction BF (×2)	4,3 (×2)	14,4 (×2)	469 (×2)
Application complète	2,25 Mo	1,26 Mo	87,7 Mo/s
Recherches + extractions	542,3 ko	93,6 ko	15,9 Mo/s
Extractions	139,1 ko	86,4 ko	5,6 Mo/s

Table 3.4 – **Débits mémoire entre les blocs fonctionnels** - Image caméra de 320×240, 25 images par seconde - N=20, R=[20,10,5], ρ_{max}=5, θ_{max}=36, pixels sur 16 bits, sauf en entrée (8 bits) et en sortie d'extraction (32 bits)

Le choix de l'organisation du SoC est alors plus facile à faire, du type d'opérateurs à l'architecture des canaux de communication.

3.3 Conclusion

Au vu des résultats obtenus en 3.2.2 et en 3.2.3, on remarque que le déploiement logiciel embarqué des recherches et des extractions nécessite une forte parallélisation.

Il apparaît nécessaire d'accélérer matériellement au moins les recherches de points d'intérêt, en plus de tous les traitements dont le portage matériel est déjà prévu. Ces traitements se révèlent relativement complexes à déployer sous forme d'IP, mais leur durée d'exécution logicielle embarquée force le choix d'une autre solution. Les extractions de voisinages au format log-polaire conservent des durées d'exécution logicielles acceptables sur Cortex A8, indépendamment de la taille de l'image étudiée. Il est donc prévu de les déployer en logiciel, si possible sur un processeur assez puissant embarqué dans le circuit FPGA.

Dans le cas d'un déploiement à base de processeurs soft-core, bien moins puissants, une forte parallélisation des processeurs risque d'être nécessaire. Une solution exécutant la totalité des traitements en matériel reste envisageable en dernier recours, si les partitionnements logiciel/matériel ne permettent pas d'atteindre les objectifs du projet.

Chapitre 4

Architecture des traitements matériels (IP)

4.1 Vue d'ensemble

L'organisation de notre architecture, déployée sur FPGA, est représentée en figure 4.1. Elle se compose de plusieurs IP connectées entre elles, reliées à la caméra et au système logiciel par des interfaces dédiées.

Cette partie matérielle génère, à partir de l'image de la caméra, plusieurs résultats qui sont lus par le système logiciel embarqué :

- six Différences de Gaussiennes représentant six octaves de fréquence spatiale
- les listes triées de points d'intérêt trouvés dans ces six DoG

Un signal indique au système logiciel que les résultats sont disponibles, afin qu'ils les lise. Ainsi, il lit les coordonnées des points d'intérêt, et récupère dans les DoG les anneaux de pixels dont il a besoin pour générer les caractéristiques locales. Ces imagettes log-polaires sont alors envoyées au réseau de neurone artificiel par le biais du module Ethernet de la carte.

Le flux de pixels en provenance de l'interface de la caméra est transféré en entrée de la première IP : l'intensité de gradient. Un signal Enable permet d'activer le fonctionnement de cette IP dès qu'un nouveau pixel est disponible en entrée à chaque nouveau cycle d'horloge. Les coordonnées des pixels permettent aux IP de maîtriser les effets de

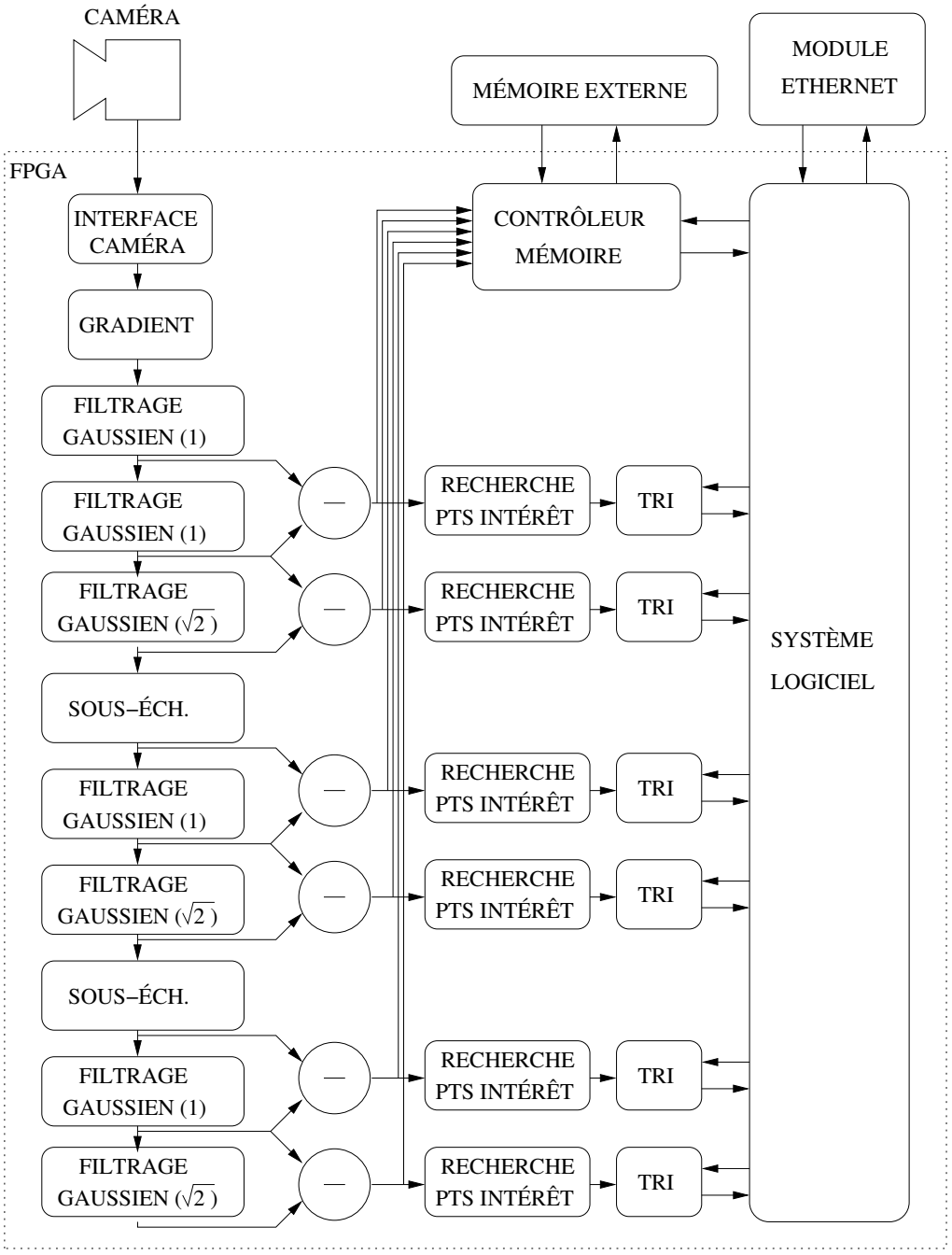


Figure 4.1 – **Vue d'ensemble des IP** - Organisation des unités de traitement matérielles dans le FPGA

bord. L'interface entre la caméra et l'IP de gradient génère ces signaux supplémentaires en fonction des réglages de la caméra et de ses signaux de sortie.

Les DoG sont stockées dans une mémoire externe au FPGA, un contrôleur mémoire dédié permet de stocker les six DoG, en laissant un accès en lecture pour le système logiciel.

Les pixels, leurs coordonnées et les signaux Enable forment l'ensemble des données communiquées d'une IP à l'autre, du gradient jusqu'aux sorties des DoG. Les recherches et tris de points d'intérêt ont des fonctionnements différents : l'IP de recherche fournira, en plus des pixels et de leurs coordonnées, un signal indiquant si le pixel est un point d'intérêt. Les IP de tri ne liront les pixels et leurs coordonnées que quand ce signal sera actif, afin de ne trier que les points d'intérêt.

La partie logicielle verra les IP de tri comme autant de mémoires RAM, contenant pour chaque octave les coordonnées des points d'intérêt.

4.2 Squelette des IP

La plupart des IP ayant pour entrées et sorties des flux de pixels de même type, il est utile de mettre en place une interface standard commune à ces IP. Cette standardisation permet une connexion transparente d'une IP à une autre, et la modularité de l'architecture globale du système matériel.

Cette interface se base sur les signaux suivants :

- la valeur de pixel,
- ses coordonnées X et Y,
- un signal Enable

Ainsi, chacune de ces IP traite ainsi un flux de pixels dont les coordonnées sont connues à tout moment, et un signal Enable permet de limiter le fonctionnement de l'IP que lorsqu'un pixel valide est fourni.

Les IP générant ce même type de données en sortie gèrent en interne leur latence intrinsèque, pour assurer la synchronisation des valeurs et des coordonnées des pixels de sortie. Un signal enable est généré afin de permettre à l'IP cliente de récupérer les

pixels seulement quand ils sont valides. L'interruption du flux de pixels en entrée sera alors répercuté de la première IP aux suivantes par effet boule de neige.

Le squelette externe d'une IP ayant cette interface standard en entrée et en sortie est présenté en figure 4.2.

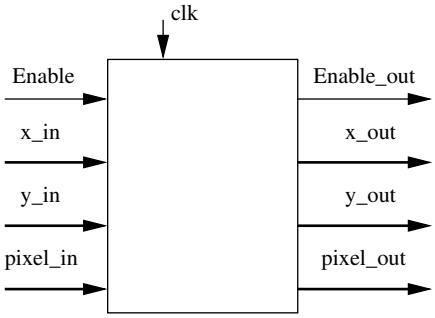


Figure 4.2 – **Interface des IP en flot de pixels** - Interface standard du système de vision

Les différences de Gaussiennes sont un cas particulier, les entrées de l'IP étant deux flux de pixels non-synchronisés. La section 4.5 montre comment la synchronisation des deux flux est gérée.

L'IP de recherche de points d'intérêt utilise cette interface en entrée, mais en sortie, le signal Enable_out est remplacé par un signal "Trier". Ce signal indique que le pixel en sortie est un point d'intérêt potentiel, et doit être inséré dans la liste triée de points d'intérêt (si sa valeur lui permet de rester dans la liste).

Les IP de tri de points d'intérêt fournissent un seul signal de façon active : une fois que l'image est traitée dans son intégralité, le signal "Terminé" indique à l'interface du système logiciel que la demi-échelle correspondante est traitée. Le système logiciel peut alors travailler sur chacune des six demi-échelles au fur et à mesure qu'elles finissent d'être étudiées par les IP. Les résultats des IP de tris, sont lus à la manière de mémoires RAM, contenant les coordonnées des points d'intérêt par ordre de valeurs de points d'intérêt.

4.3 Intensité de gradient

L'algorithme choisi pour calculer l'intensité de gradient de l'image d'entrée correspond à une version simplifiée de l'opérateur de Sobel. La différence fondamentale avec l'opérateur de Sobel est la racine carrée, que l'on remplace par une simple moyenne pour gagner en place sur le FPGA pour un moindre impact sur l'efficacité du système. Dans le cas où l'on voudrait éviter cette approximation, le déploiement d'un opérateur de racine carrée pourrait être envisagé. Différentes IP présentes dans la littérature permettent le calcul d'une racine carrée en une seul itération (CL00), ce qui permettrait de conserver une homogénéité du comportement temporel des IP du système de vision. En figure 4.3, on peut voir l'architecture proposée pour l'IP d'intensité de gradient.

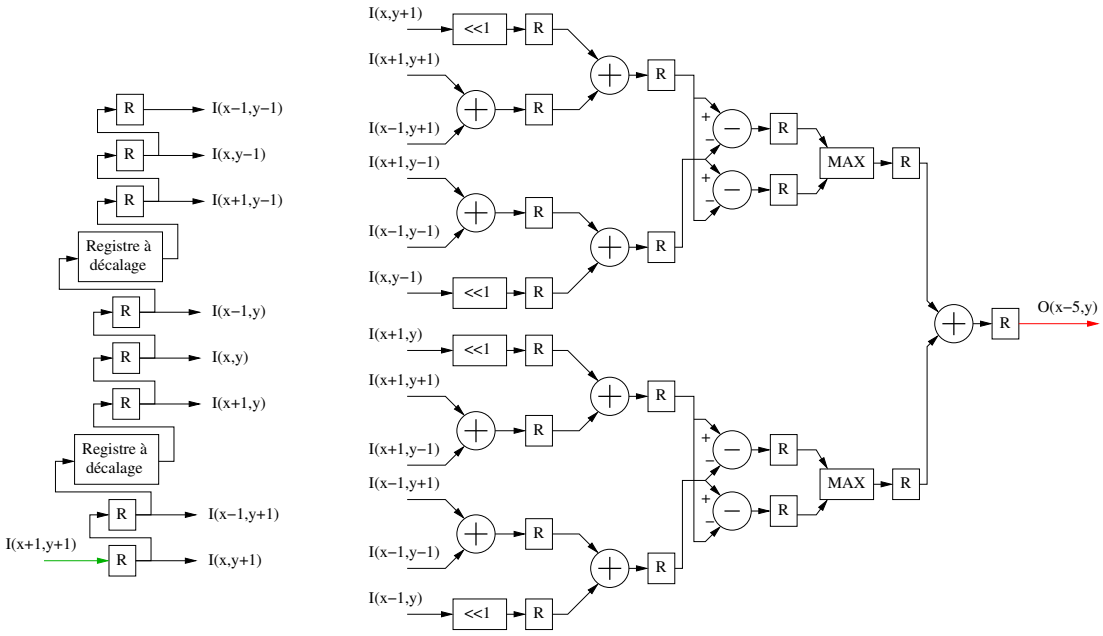


FIGURE 4.3 – **IP d'intensité de gradient** - À gauche, gestion des pixels d'entrée, à droite, calcul de l'intensité du gradient

La synchronisation des pixels en entrée de l'opérateur de l'IP est un point important. Une mémorisation des pixels d'entrée de l'IP, à base de registres à décalage et de simples registres, permet de fournir simultanément les huit pixels nécessaires au calcul de chaque pixel de sortie.

Des étages de registres permettent de disposer le calcul en pipeline, afin d'augmenter la bande passante, au prix d'un peu de latence supplémentaire. La latence de l'IP est ici de $w_{image} + 6$ cycles d'horloge, la mise en mémoire de deux lignes et un pixel étant nécessaire pour effectuer le calcul, et les étages de pipeline du calcul induisant un retard de seulement 5 cycles d'horloge.

La gestion des effets de bords dans ce bloc fonctionnel permet d'éviter la corruption d'une bande d'un pixel de large autour de l'image. Le mécanisme utilisé ici n'est pas montré dans la figure, faute de place. Il consiste tout simplement à remplacer par la valeur "0" tout pixel d'entrée du bloc de calcul qui serait hors de l'image. La détection des bords est effectuée à partir des coordonnées du pixel d'entrée, et des constantes w_{image} et h_{image}. Ce mécanisme de gestion d'effets de bords a pour conséquence d'atténuer la valeur du gradient pour les pixels disposés sur le bord de l'image, sans affecter les autres pixels, qui ne subissent pas d'effet de bord.

4.4 Filtrage Gaussien

Le filtrage Gaussien est réalisé par une convolution de l'image d'entrée avec un noyau de coefficients 2D. Étant donné que le filtrage Gaussien 2D est séparable en deux convolutions 1D, des déploiements en deux passes 1D sont étudiés et comparés aux équivalents en une passe 2D. On peut ainsi connaître l'impact du choix d'un filtrage séparable, et le surcoût qu'impliquerait un changement pour un filtrage non-séparable.

Les IP de convolution 2D présentes dans la littérature sont présentées en section 4.4.1. Il est à noter que la plupart de ces déploiements ne semblent pas particulièrement s'attacher à une optimisation quelconque de l'IP.

Les algorithmes décrits en section 3.1.2.1 sont tous fidèles à la version logicielle du filtrage Gaussien, au détail près du type de données (virgule fixe). Les différentes architectures proposées ici jouent sur l'organisation des opérations de ces algorithmes, afin de paralléliser les calculs au maximum.

Les types de données utilisés ici sont génériques, et aisément réglables avant synthèse des IP. Dans les tests qui ont été faits, les pixels d'entrée et de sortie sont stockés sur

des nombres à virgule fixe non-signés de 16 bits. Les produits et les sommes sont quant à eux stockés sur 32 bits.

4.4.1 État de l'art : convolution Gaussienne 2D

Si l'intensité de gradient, les DoG et sous-échantillonnages sont relativement simples à déployer, ou suffisamment peu communs sous forme d'IP, la convolution 2D représente un point plus délicat. Dans le cas présent, on souhaite :

- une bonne gestion des effets de bords, afin de minimiser l'erreur dans les résultats obtenus.
- une cadence d'exécution à la hauteur du reste du système : dans le cas présent, un nouveau pixel sera traité à chaque cycle d'horloge.
- une consommation raisonnable des ressources du FPGA ciblé.

Dans (BPS98), une architecture rapide est proposée pour déployer une convolution 2D sur un flot de pixels. Le calcul de la convolution se fait au fur et à mesure que les pixels entrent dans le système (calcul en pipeline), à une cadence d'un pixel par coup d'horloge. Cette architecture, proposée en 1998, est déployée sur les FPGA de cette époque, disposant de peu de mémoire. L'auteur propose de paralléliser la convolution sur plusieurs FPGA pour pallier à ce problème. Encore aujourd'hui, la parallélisation sur plusieurs FPGA reste envisageable dans le cas d'une fenêtre de convolution de très grande taille.

Dans (Nel00), une IP de convolution 2D 3×3 est présentée, accompagnée du code VHDL correspondant. L'approche traditionnelle est utilisée : les pixels en entrée passent par des registres et des FIFO pour que les entrées des multiplieurs correspondent aux pixels recouverts par la fenêtre de convolution. Les produits sont ensuite sommés dans un arbre d'additionneurs. La division par la somme des coefficients est approchée à la puissance de 2 la plus proche, l'erreur étant jugée acceptable par l'auteur. Son approche permet une gestion basique des effets de bords : dès que la fenêtre de convolution dépasse du bord de l'image, le résultat de convolution est rendu nul.

L'étude de la convolution 2D a aussi fait l'objet d'optimisations énergétiques (Per03). L'architecture proposée par Perri se base sur des multiplieurs SIMD, sur un arbre d'ad-

ditionneurs SIMD et un module de saturation SIMD (ce dernier servant à formater la somme des produits sur 16 bits non-signés). Cette architecture est adaptable à de nouveaux paramètres en cours d'exécution, que ce soit la largeur des bus de données des pixels ou des coefficients du noyau de convolution.

Une méthode de déploiement matériel de la convolution 2D, proposée par B. Cope (Cop06), apparaît comme très proche de ce que l'on souhaite obtenir. Cette architecture utilise la flexibilité des FPGA pour déployer des pipelines parallèles pour le calcul des convolutions. L'auteur compare cette architecture à des déploiements logiciels sur PC et sur GPU. Si les déploiements sur certains GPU se montrent bien plus efficaces que l'architecture FPGA proposée pour déployer des convolutions pour des petites fenêtres, l'avantage revient au FPGA dès que la fenêtre de convolution utilisée est de 4×4 pixels. Alors, un déploiement sur Spartan3 prend de vitesse le GPU 6800 Ultra qui travaillait 5.6 fois plus vite que lui pour une fenêtre de 2×2, et 2.1 fois plus vite pour une fenêtre de 3×3. Les tailles de fenêtres concernant notre projet étant de 7×7 et 9×9, le facteur d'accélération correspondant est respectivement de 4.1 et de 8.1 en faveur du déploiement FPGA.

Plus récemment, Fons et Al. (FFC10) présentaient une architecture de convolution 2D reconfigurable dynamiquement. Les dimensions du noyau, les largeurs de bus et les étages de pipeline sont ici reconfigurables en cours d'exécution, en exploitant la technologie de reconfiguration dynamique des Virtex 4 de Xilinx. Les auteurs présentent les FPGA comme étant toujours plus efficaces que les DSP récents pour la convolution 2D, ce qui ne peut que renforcer les conclusions de Cope. Une fois de plus, la somme des produits est réalisée à l'aide d'un arbre d'additionneurs.

La capacité de fournir un pixel de l'image de sortie à chaque coup d'horloge est proposée dans chacun de ces documents. Cet aspect est attrayant pour une plate-forme dont la résolution d'images d'entrée n'est pas fixée définitivement. En effet, si une image de faible résolution ne représente que peu de pixels à traiter, on préférera garder la possibilité de traiter des images de grandes dimensions tout en gardant la même cadence de fonctionnement. La limitation sera ici la fréquence de fonctionnement maximale de l'architecture sur le FPGA. À l'inverse, l'utilisation d'une caméra de faible résolution

permettra le déploiement à plus faible cadence, ce qui permet généralement de réduire la puissance électrique consommée.

Dans le système robotique (BMC08) présenté en 2.1.1, les auteurs présentent un déploiement matériel de détection de caractéristiques à base de DoG. Leur déploiement des filtrages Gaussiens ne s'effectue pas en 2D directement, mais en deux passes 1D. L'utilisation de deux passes 1D implique le stockage d'au moins une partie du résultat intermédiaire dans une mémoire, ce qui peut se révéler délicat à mettre en œuvre dans un état d'esprit d'économie de mémoire. Les résultats de ce déploiement sont présentés, l'opérateur étant capable de traiter un filtrage Gaussien de 30 images de 320×240 pixels par seconde.

4.4.2 Noyau de convolution et division

Les noyaux des convolutions Gaussiennes utilisés dans l'application sont définis sous forme de tableaux de constantes dans le code VHDL. Les coefficients sont déclarés sur 16 bits chacun, étant donné que les pixels sont stockés sur 16 bits et que la somme des produits sera sur 32 bits. Afin d'économiser des ressources utilisées par les multiplications, l'outil de synthèse réduira automatiquement la taille des bus de ces constantes à des valeurs adéquates (un coefficient de valeur 11 n'a besoin d'être stocké que sur 4 bits par exemple).

Pour toutes les architectures proposées, un élément reste parfaitement identique : la division finale de la somme des produits. Une division est une opération sensible quand il s'agit de la déployer sur une architecture électronique. Au contraire, une division par une puissance de 2 est une opération triviale : à partir d'une valeur, un décalage à droite de n permet d'obtenir la division par 2^n. Si l'on suppose que le diviseur est constant, l'exclusion des n bits de poids faible de la valeur d'entrée permet de faire cette division sans la moindre latence. Dans le cas contraire, il est nécessaire d'utiliser un opérateur de division coûteux en ressources matérielles. Un remplissage aura lieu si besoin par câblage direct (dans le cas présent, on garde les 16 bits de poids fort d'une valeur de 32 bits, aucun remplissage n'est nécessaire).

Les coefficients du noyau Gaussien seront calculés de façon à ce que leur somme soit égale à une puissance de 2 (ou le plus proche possible). On peut ainsi normaliser la sortie de convolution par une simple division par cette puissance de 2, ce qui réduit l'utilisation des ressources du circuit. Nelson, par exemple, se base sur ce principe (Nel00), sans adapter les valeurs des coefficients : si la somme des coefficients vaut 9, il propose de diviser par 8. Dans le cas présenté, cette approche nécessite un bit supplémentaire en sortie pour gérer les éventuels débordements, ou réduit la précision du résultat de moitié si l'apparition de ce bit supplémentaire ampute le résultat de son bit de poids faible. L'obtention de valeurs maximales du gradient (tous les bits à '1') étant mathématiquement impossible avec l'opérateur de Sobel (maximum théorique composé de 3 bits de poids faible à '0'), et une image maximisant le résultat de l'opérateur étant quasi-impossible à capter sur une caméra hors conditions de laboratoire (optiques et capteurs imparfaits, bruit électronique), une marge d'erreur est acceptable dans les valeurs de coefficients utilisés. On tendra alors vers la valeur la plus juste des coefficients, pour se rapprocher au mieux des résultats obtenus en logiciel avec des coefficients de type "float" ou "double".

Dans le cas présent, on suppose la somme des coefficients stockée sur 24 bits et les pixels sur 17 bits, la somme des coefficients doit se rapprocher de 2^{24}. Le tableau 4.1 répertorie deux types d'erreurs obtenues pour différentes largeurs de noyaux de convolution. La première, l'erreur par rapport à 2^{24}, indique quel est le rapport entre cette valeur idéale pour la somme des coefficients (1000000 en base 16). Cette erreur est due à la baisse de précision que l'on subit en passant d'un "double" à un entier proche de 2^{24}. La deuxième erreur est mesurée par rapport à la valeur de la somme des coefficients pour un noyau de dimensions infinies (logiciellement approché avec des noyaux de 2001×2001 coefficients). Cette dernière erreur indique donc l'erreur de précision due à la taille du noyau, combinée à la première erreur.

Au delà d'une certaine taille de noyau, les coefficients sont tous nuls, dépasser cette taille limite n'apporte rien à la précision du résultat. Pour $\sigma = 1$, un noyau de 9×9 suffit à contenir tous les coefficients pour une somme proche de 2^{24}. Pour $\sigma = \sqrt{2}$, cette taille limite est de 15×15 coefficients. Les précisions maximales de $-6.15e^{-8}$ pour $\sigma = 1$ et de $-7.84e^{-9}$ pour $\sigma = \sqrt{2}$ sont listées ici pour des coefficients codés sur 24 bits au plus. Pour aller au delà de cette précision, il faut utiliser des bus de données plus larges

K	σ^2	somme des coefficients	erreur par rapport à 2^{24}	erreur par rapport à la Gaussienne réelle
3	1	FFFFFF	$-5.96e^{-10}$	$-2.21e^{-3}$
5	1	FFFFFA	$-3.58e^{-9}$	$-1.82e^{-4}$
7	1	FFFFFF	$-5.96e^{-10}$	$-5.41e^{-6}$
9	1	FFFFFD	$-1.79e^{-9}$	$-6.15e^{-8}$
3	2	1000000	0	$-4.79e^{-3}$
5	2	FFFFFE	$-1.19e^{-9}$	$-1.37e^{-3}$
7	2	FFFFFC	$-2.38e^{-9}$	$-2.29e^{-4}$
9	2	FFFFFF	$-5.96e^{-10}$	$-2.32e^{-5}$
11	2	100000E	$8.34e^{-9}$	$-1.44e^{-6}$
13	2	FFFFF4	$-7.15e^{-9}$	$-6.24e^{-8}$
15	2	FFFFF5	$-6.56e^{-9}$	$-7.84e^{-9}$

TABLE 4.1 – **Précision des noyaux Gaussiens** - noyaux de convolution de K^2 coefficients sur 24 bits : erreur de la somme par rapport à 2^{24}, et erreur de précision de la convolution

pour stocker les coefficients, l'agrandissement des noyaux n'ayant aucun effet sans cette mesure supplémentaire.

4.4.3 Convolution 2D

Plusieurs architectures matérielles ont été évaluées pour la convolution 2D.

La première solution est la méthode traditionnellement présente dans la littérature (section 4.4.1), dont l'architecture est présentée en figure 4.4.

Cet opérateur calcule à chaque coup d'horloge le pixel de sortie en fonction des pixels d'entrée qui lui correspondent, et de leurs coefficients respectifs. Les pixels de l'image d'entrée passent par un jeu de registres à décalage pour arriver en entrée des multiplieurs simultanément. Les multiplications se font ainsi en parallèle pour un même pixel de l'image de sortie. Les produits sont ensuite additionnés entre eux pour produire la valeur du pixel de sortie.

La latence L de cette IP peut se calculer, en cycles d'horloge, par la formule $L_{IP} = (H_{noyau}/2) * W_{image} + (W_{noyau}/2) + L_{opérateur}$ (divisions entières). Pour un noyau de convolution de 9×9 pixels, $L_{opérateur}$ sera de 8 coups d'horloge : 7 étages de

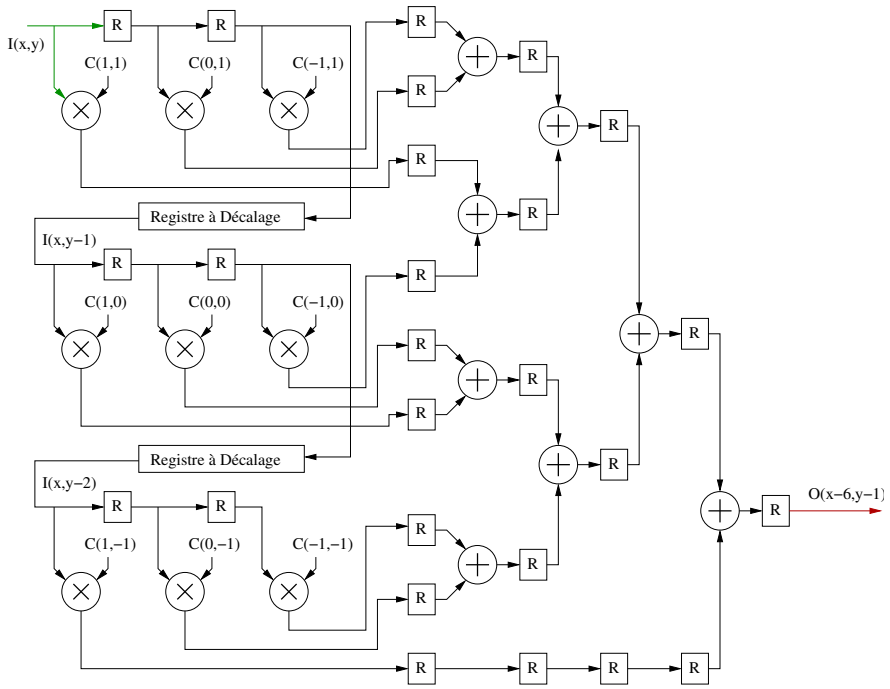


Figure 4.4 – **Convolution 2D matérielle : Architecture traditionnelle** - Exemple sur noyau 3x3

pipeline permettant de calculer la somme des 81 produits, et ces derniers représentent un étage supplémentaire de pipeline. Le nombre d'étages dans le pipeline d'addition est facile à trouver. Pour calculer la somme de $2^7 = 128$ valeurs deux à deux, il y aura 7 étages de pipeline. Avec 6 étages, on ne peut calculer la somme que de $2^6 = 64$ valeurs. Pour additionner nos 81 produits, on a donc 7 étages de pipeline (en plus de l'étage de sortie de multiplication). Pour un noyau de 7×7 coefficients, les 49 valeurs à additionner nécessitent 6 étages de pipeline. Pour une image 320x240 sur laquelle on utilise un noyau de coefficients de taille 9x9, la latence de l'IP sera de 4x320 + 4 + 8 =1292 coups d'horloge, en supposant que le flot de pixel envoie un nouveau pixel à chaque coup d'horloge. Sur ces 1292 coups d'horloge, 1284 seront dus aux dimensions du noyau de coefficients. Pour calculer le pixel de sortie (produit de convolution) aux coordonnées [x,y], tous les pixels de l'image d'entrée devront être lus par l'IP. D'où le décalage de 4 pixels en x et en y. Seuls 8 coups d'horloge seront dus à la topologie de l'opérateur.

La deuxième solution, utilise le principe des MACC pour cette convolution 2D. Deux architectures basées sur ce principe sont présentées en figure 4.5.

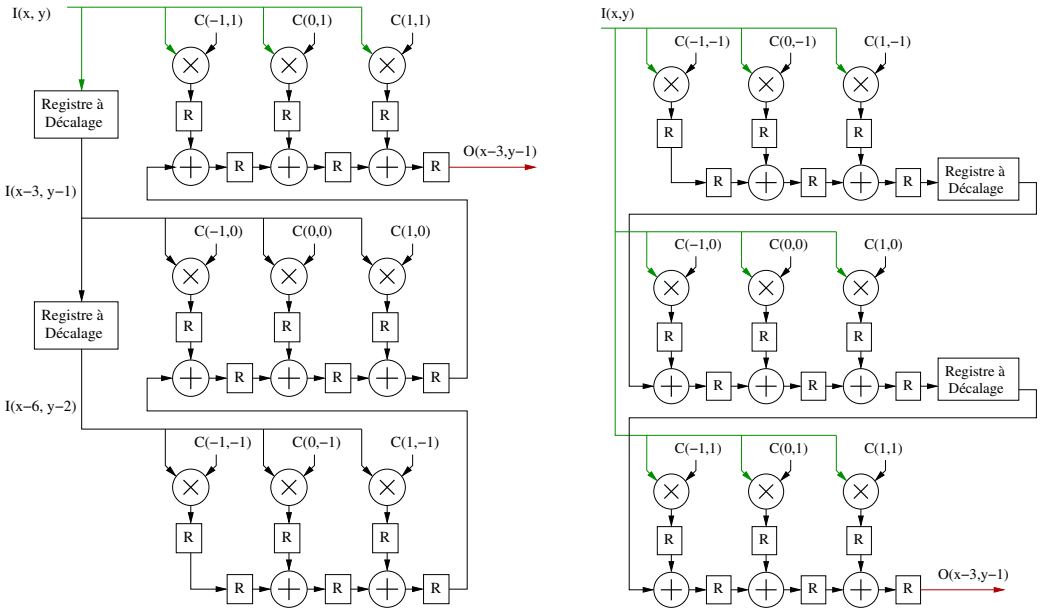


FIGURE 4.5 – **Convolution 2D matérielle : Architectures type MACC** - Exemples sur noyau 3x3. Mise en registre des pixels d'entrée ou de l'accumulation. Opérateurs MACC représentés par leurs multiplieurs et additionneurs internes

Cette fois, la somme des produits sera calculée selon le principe des MACC. La somme des produits est calculée par accumulations successives. Dans l'architecture de gauche, le dernier pixel sera multiplié simultanément par tous les coefficients de la dernière ligne du noyau. L'architecture MACC permet d'additionner les résultats obtenus aux sommes de produits correspondantes. Des registres à décalage permettent de multiplier avec les autres lignes de coefficients du noyau, les pixels adéquats. Cette architecture permet de stocker en registres à décalage les pixels d'entrée, qui ont souvent un bus moins large que les résultats de MACC.

L'architecture de droite fonctionne sur le même principe, à la différence que l'utilisation de registres à décalage se fera sur les résultats de MACC. Le pixel d'entrée de l'IP est multiplié par tous les coefficients du noyau, et les résultats sont accumulés de façon à obtenir à chaque coup d'horloge un pixel en sortie.

Le pixel de sortie de ces deux architectures correspond simplement à la sortie de la dernière MACC.

La latence L de ces IP se calculera par la formule $L_{IP} = H_{noyau} * W_{image} + W_{noyau} + L_{opérateur}$, comme pour l'autre solution. La différence réside dans la valeur de $L_{opérateur}$, qui voit sa valeur baisser à 2, quelle que soit la taille du noyau de convolution : un registre en sortie de multiplication, un autre en sortie d'accumulation.

Les trois solutions proposées utilisent le même nombre de multiplieurs et d'additionneurs, seule la topologie de l'opérateur diffère.

La première est plus intuitive, cependant les solutions à base de MACC sont plus pratiques à déployer sous forme d'IP générique : on souhaite pouvoir utiliser le même code pour différentes tailles de noyau et de bus de données. Les solutions à base de MACC permettent aussi de réduire légèrement la latence des calculs, et surtout de s'abstraire de la variation de latence introduite par l'arbre d'additionneurs. En effet, la somme des produits est calculée progressivement, les additionneurs étant disposés non plus en arbre mais en ligne. La taille du masque de convolution, qui impacte directement sur la latence de la première architecture, n'aura pas d'effet sur la latence des deux autres. À partir du moment où l'opérateur aura reçu le dernier pixel nécessaire au calcul d'une fenêtre, le pixel de sortie correspondant au centre de la fenêtre sera calculée en

deux coups d'horloge. Cette latence est due aux deux registres de la dernière MACC (sortie de multiplieur et sortie d'additionneur).

De plus, vis-à-vis du code VHDL, la généricité des dimensions du noyau Gaussien est rendue plus aisée, les additions étant intégrées à l'architecture en tableau 2D du reste des opérations. Pour la première solution, la conception générique d'un arbre d'additionneurs n'est pas un problème trivial, ces opérations formant une architecture irrégulière dès que le nombre d'additionneurs n'est pas une puissance de 2.

Pour les résultats des produits et des sommes de produits, on pourra utiliser des données plus larges que pour les pixels d'entrée et de sortie, afin d'améliorer la précision de la convolution. Cette décision entraînerait l'avantage des deux premières solutions proposées, car la mémoire consommée par la troisième augmenterait proportionnellement avec la largeur des données des MACC. Dans le cas où ces valeurs seraient stockées sur des données de même taille, les ressources mémoire nécessaires aux deux solutions seraient équivalentes, et la troisième solution garde un avantage conséquent, présenté dans la section suivante.

4.4.4 Coefficients redondants dans les noyaux de convolution

Dans beaucoup de cas nécessitant une convolution 2D, et particulièrement dans le cas présent, plusieurs coefficients ont la même valeur au sein d'un noyau de convolution. Dans le cas d'un noyau Gaussien, une très forte symétrie est visible : les coefficients sont répétés 8 fois, sauf sur les diagonales, la verticale, et l'horizontale qui se croisent au centre, dont les coefficients sont répétés 4 fois. Le centre, lui, est la seule valeur unique du noyau. Dans un tel cas, il est possible de réduire fortement le nombre de multiplieurs dans l'IP, en utilisant un même multiplieur pour calculer le produit d'un coefficient avec la somme des pixels concernés. Le multiplieur aura un bus de données plus large en entrée (une somme de pixels nécessitant un bus plus large qu'un pixel seul), c'est là le défaut de la mise en commun de multiplieurs. Cette solution est notamment proposée par French (Fre04) pour la convolution 2D.

On peut le remarquer en figure 4.6, où l'on passe de 9 à 3 multiplieurs en regroupant les coefficients de même valeur, et en multipliant la somme des pixels concernés par ces coefficients.

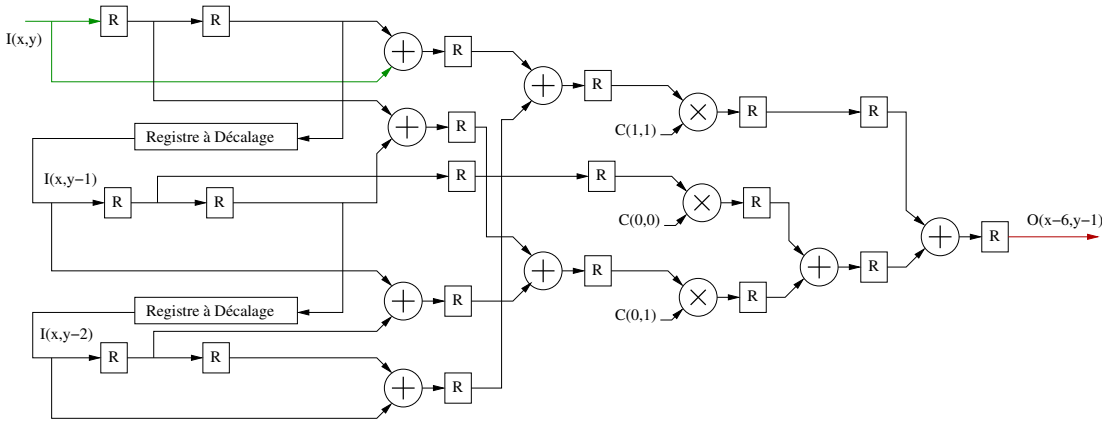


FIGURE 4.6 – **Convolution 2D matérielle : Coefficients redondants - architecture traditionnelle** - Exemple sur noyau Gaussien 3x3

Cette solution garde les défauts de l'architecture traditionnelle, à savoir la difficulté de conception générique, et la latence plus forte et dépendante de la taille du noyau.

Concernant les architectures à base de MACC, en figure 4.7, on peut voir que la solution de droite permet une réduction plus efficace du nombre de multiplieurs.

La solution de gauche permet de passer de 9 à 6 multiplieurs, les coefficients pouvant être regroupés par ligne, mais pas d'une ligne à l'autre dans cette architecture. La solution de droite permet la même réduction que l'architecture traditionnelle, au détriment de la mémoire, dans le cas où les bus de données des MACC seraient plus larges que ceux des pixels.

Un compromis entre les trois éléments (latence et généricité, utilisation mémoire, nombre de multiplieurs) doit donc être choisi.

Il reste une optimisation essentielle dans le cas de noyaux de convolution Gaussiens : la séparation en deux convolutions 1D.

4.4.5 Séparation en deux convolutions 1D

Le filtrage Gaussien par convolution 2D étant séparable en deux passes de convolution 1D, des IP correspondantes ont été développées, afin de les comparer aux convolu-

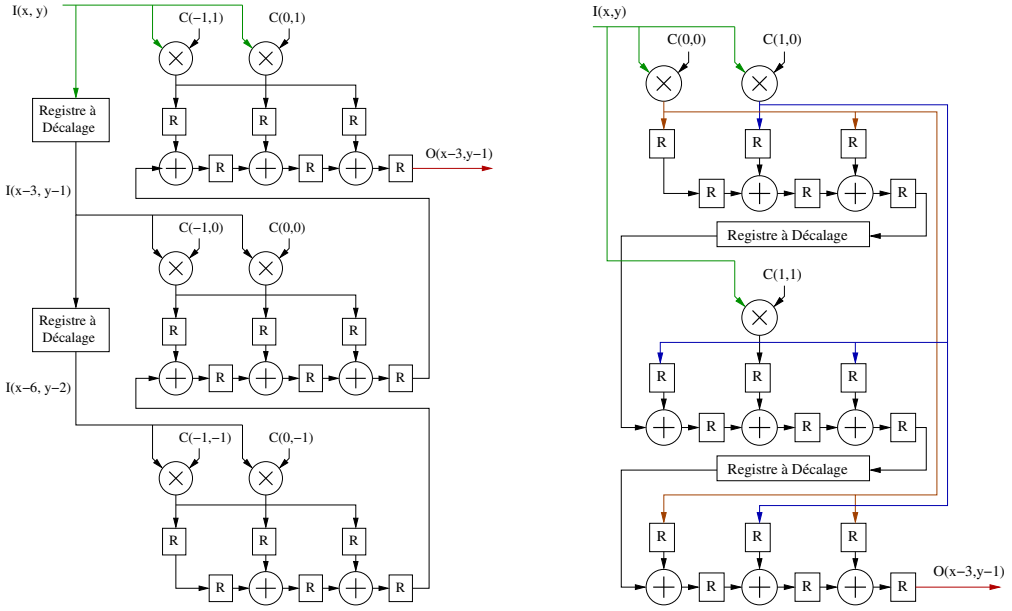


FIGURE 4.7 – **Convolution 2D matérielle : Coefficients redondants - architectures MACC** - Exemple sur noyau Gaussien 3x3

tions 2D. La figure 4.8 présente l'architecture de ces IP.

Dans le cas général, on remarque que le nombre d'opérateurs est diminué par rapport aux architectures 2D. Cependant, dans le cas du noyau de 3×3 coefficients, un plus grand nombre de multiplieurs est nécessaire (4 au lieu de 3 dans le cas de noyaux de coefficients symétriques). Au delà de noyaux de 3×3 coefficients, les IP utilisant la séparation en deux passes 1D sont avantageuses ou similaires en tout point. L'application de vision utilisant des noyaux de tailles 7×7 et 9×9, cette architecture est avantageuse en tous points pour le SoC de vision.

À part l'utilisation d'un plus grand nombre de multiplieurs pour les petits noyaux de convolution, cette IP demeure plus intéressante que les solutions basées sur une seule passe 2D, quelle que soit la taille du noyau. Le choix d'un filtrage en deux passes 1D ou en une passe 2D dépendra, pour les petits noyaux, des ressources disponibles dans le FPGA au moment de la synthèse de l'architecture complète.

4.4.6 Gestion des effets de bords

Des effets de bord sont générés par toutes les IP de convolution décrites précédemment. La figure 4.9 permet de visualiser la cause de ces effets de bords : les pixels en

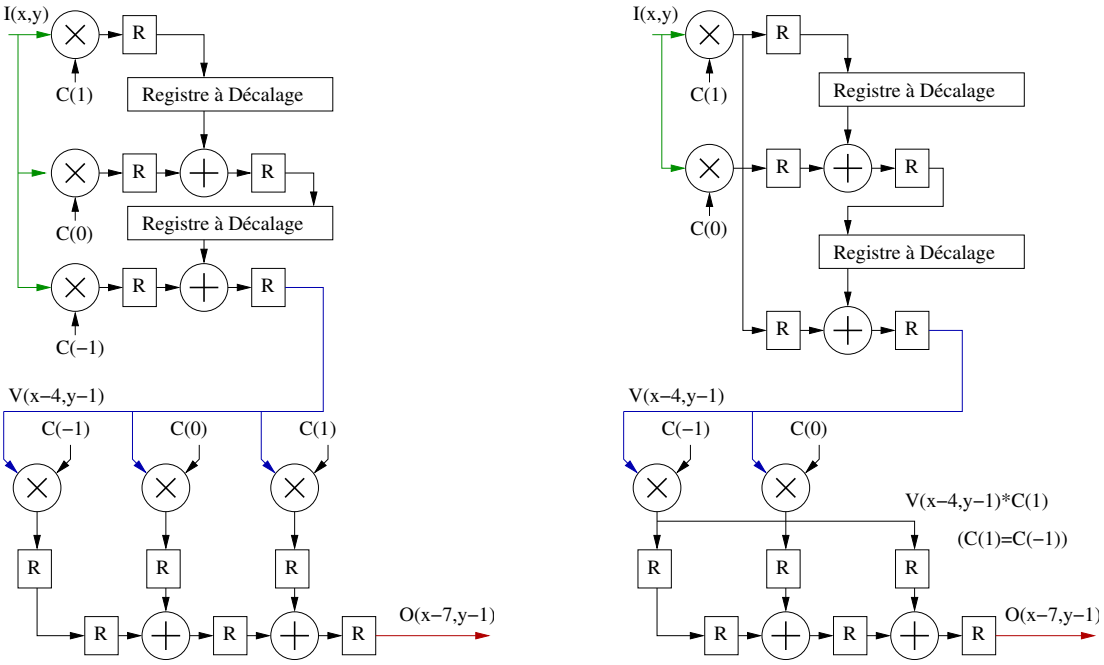


FIGURE 4.8 – **Convolution 2D matérielle : deux passes 1D** - Coefficients symétriques à droite. Le signal V correspond à la première passe, soit le filtrage vertical

noir sont hors de l'image. Lors du filtrage Gaussien, l'opérateur lit une fenêtre de $m \times n$ pixels dans l'image d'entrée (m et n étant les dimensions du noyau de coefficients). Si la plupart du temps, tous ces pixels seront valides, lorsque l'opérateur s'approche des bords de l'image, cette fenêtre dépasse plus ou moins de l'image d'entrée. En se rapprochant d'un bord de l'image, avec un noyau de 9×9 coefficients, on aura d'abord 9, puis 18, 27 et enfin 36 pixels invalides sur 81. Dans un coin de l'image, on arrivera dans les mêmes conditions à 56 pixels invalides sur 81. Les signaux des pixels pouvant être remplacés par des valeurs aléatoires (sur les bords haut et bas de l'image), ou des pixels de l'autre bord de l'image (sur les bords gauche et droit), la valeur de sortie de l'opérateur peut être fortement corrompue. Il est donc indispensable de prendre en compte ces effets de bord.

Deux solutions se présentent : une solution mathématiquement sans erreur, et une solution permettant de réduire la corruption des pixels de sortie proches du bord. La solution sans erreur nécessite de rogner la taille de l'image de sortie par rapport à

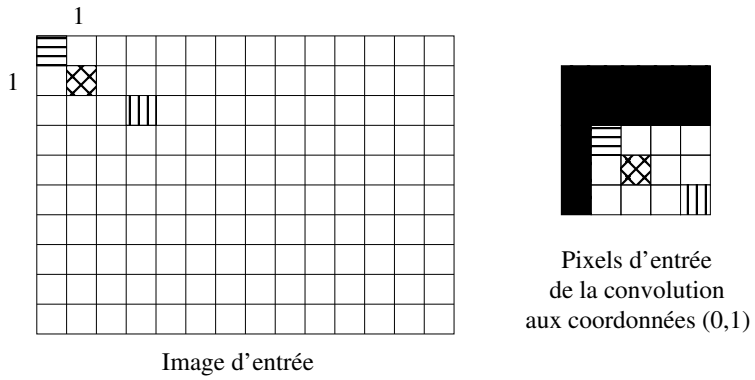


FIGURE 4.9 – **Effets de bord de l'algorithme de convolution 2D** - Les pixels hors de l'image dans la fenêtre de convolution sont ici en noir. Exemple pour un noyau de convolution de 5x5 pixels

l'image d'entrée. Sur une image de 320×240 pixels, une convolution avec un noyau de 9×9 coefficients générera une image de taille 312×232 pixels : on aura rogné l'image de 4 pixels sur chaque bord. Cette solution est attrayante dans le cas d'images de grande résolution en entrée, mais le passage de l'image d'entrée par la cascade de filtres Gaussiens fait que cette solution réduit la perception périphérique du robot dans les basses fréquences spatiales. Au bout des trois premiers filtrages Gaussiens, l'image d'entrée sera passée de 320×240 à 300×220. À la fin du dernier filtrage, l'image de plus basse fréquences spatiales fera 64×44 pixels au lieu de 80×60 pixels, soit une perte de surface de plus de 40%. Ces bandes de basses fréquences étant les plus importantes pour la navigation et la détection d'obstacle, il semble difficile d'envisager un tel défaut dans les algorithmes.

Au lieu de cette solution coûteuse, on préférera réduire la corruption des pixels calculés près des bords. Cette réduction se fera en modifiant les valeurs des pixels invalides, ce qui impactera directement sur la valeur du pixel de sortie. Une maximisation des pixels d'entrée invalides n'est bien sûr pas une idée valable : l'image de sortie serait plus claire sur les bords. Une minimisation de ces pixels entraînerait au contraire un assombrissement des bords. Donner aux pixels invalides la valeur moyenne d'un pixel (127 ou 128 pour un pixel codé sur 8 bits, par exemple), permet de couvrir les images claires et les images sombres : l'impact sur les images très sombres ou très claires sera limité par rapport aux deux solutions présentées.

Cela dit, dans le cas de l'application de vision, il convient de rappeler l'utilisation qui est faite de ces filtrages. En entrée de la cascade de filtres se trouve l'intensité du gradient de l'image de la caméra, généralement sombre, sauf environnement visuel très chargé d'arètes et de points saillants. En sortie, une détection de maxima locaux, permet d'identifier des points les plus saillants de l'image. Un bord trop clair créera artificiellement des faux points saillants au bord de l'image, empêchant la détection de vrais points d'intérêt trop proches. Un bord sombre aura pour effet d'atténuer les valeurs des points du bord. La deuxième solution est préférable, car elle permet de détecter plus facilement des points d'intérêt réels dans l'image.

On choisira par conséquent d'annuler la valeur des pixels invalides.

Les coordonnées du pixel d'entrée étant connues par les IP, il est relativement aisé de détecter les pixels invalides. Les multiplieurs étant rangées par leurs coordonnées respectives au noyau, il est aisé d'annuler le produit quand le pixel d'entrée d'un multiplieur est reconnu invalide. Un multiplexeur suffit à passer à l'additionneur correspondant une valeur nulle plutôt que la sortie du multiplieur.

Cette solution réduit l'effet de bord de l'algorithme, mais le rayon d'inhibition de l'opérateur de recherches de points d'intérêt reste moins efficace que dans les zones mathématiquement justes. Un point saillant sur le bord sera perçu comme moins saillant en sortie de filtrage que s'il se trouvait au centre de l'image. Un point légèrement moins saillant, un peu plus éloigné du bord, pourra alors être détecté comme point d'intérêt alors qu'il est dans le rayon d'inhibition du premier point.

Cette erreur, due au manque d'informations sur l'environnement visuel au delà des bords de l'image, se retrouve dans tous nos déploiements de l'application.

L'erreur maximale générée par cet algorithme est représentée en figure 4.10 : on peut voir les effets de bords sur les étapes successives de filtrage Gaussien. L'image étudiée fait 320×240 pixels, et représente une intensité de gradient de valeur maximale sur toute l'image (quasiment impossible à obtenir en situation réelle). Pour une intensité de gradient de valeur maximale sur toute l'image, l'annulation des pixels au delà des bords de l'image assombrit les bords de l'image résultat. Cette figure permet de visualiser le résultat des filtrages successifs et les conséquences du cumul de ces effets de bords. En

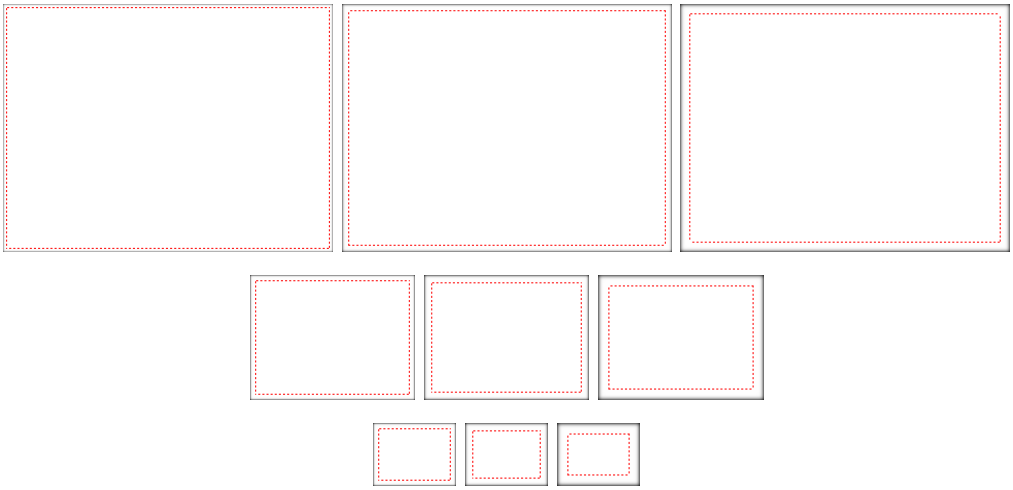


FIGURE 4.10 – **Effets de bord cumulatifs théoriques des convolutions** - Pyramide Gaussienne appliquée à une image blanche de 320×240 pixels. En pointillés rouges, les bords du résultat de convolution parfaite. Au bord des images en noir, l'erreur maximale due aux effets de bord

pointillés rouges, on peut voir les bordures des images générées par des convolutions mathématiquement parfaites. On peut voir que dans les basses fréquences, la méthode approchée permet d'étudier des images beaucoup plus grandes. On voit aussi que la convolution mathématiquement parfaite exclut des pixels dont la valeur estimée a une erreur très faible.

4.5 Différence de Gaussiennes

La différence de Gaussiennes n'étant que la soustraction de flots de pixels deux à deux dans notre cas, l'unité de calcul se résume à un simple opérateur de soustraction. Les pixels de sortie de cette IP sont signés, codés en complément à deux. La difficulté de cette IP réside dans la synchronisation des deux flux de pixels d'entrée. En effet, ils correspondent à l'entrée et la sortie d'une même IP de filtrage Gaussien, ainsi la latence de l'IP de filtrage sépare les deux flux dans le temps. On peut voir l'IP en figure 4.11.

Un registre à décalage permet de retarder le premier flux de pixels d'entrée, jusqu'à ce que les coordonnées du premier flux retardé et du deuxième flux soient identiques. Les coordonnées des pixels des deux flux permettent à la simulation de s'assurer de leur

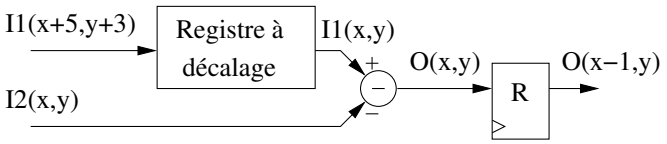


FIGURE 4.11 – **IP de Différence de Gaussiennes** - Soustraction de deux flots de pixels. Un registre à décalage est utilisé pour synchroniser les deux flots

synchronisation, la latence fixe générée par ce registre à décalage pouvant ainsi être réglée de façon sûre. Ainsi, le calcul du pixel de sortie à ces mêmes coordonnées peut être réalisé. Un registre en sortie de l'opérateur de soustraction génère une latence d'un cycle d'horloge (vis-à-vis du deuxième flux de pixels).

4.6 Sous-échantillonnage

Comme pour les IP de gradient et de filtrage Gaussien, un élément important de cette IP est la gestion des données d'entrée de l'opérateur. Pour quatre pixels en entrée de l'opérateur, un seul pixel sera généré en sortie. Cependant, ces quatre pixels ne sont jamais consécutifs dans le flux de pixels d'entrée de l'IP. La gestion des entrées de l'opérateur nécessite par conséquent une attention particulière.

Le sous-échantillonnage est calculé en moyennant des blocs de 2×2 pixels non-recouvrants. Ce mécanisme est décrit en figure 4.12 À partir d'un flot continu de pixels, une FIFO de $W/2$ éléments (W étant la largeur de l'image) permet de stocker les sommes partielles correspondant à la première ligne, puis aux autres lignes paires. Les sommes sont stockées sur un bit de plus que les pixels afin de ne pas réduire la précision du résultat. Cette FIFO sera lue durant l'arrivée dans l'IP des pixels de la deuxième ligne, puis des autres lignes impaires, afin de poursuivre les calculs pour chaque bloc. Ainsi, la FIFO se remplit et se vide toutes les deux lignes de l'image d'entrée. Les blocs de 2×2 pixels sont donc divisé en sous-blocs horizontaux de 2 pixels par ligne. Ces blocs étant non-recouvrant, l'IP utilisera le bit de poids faible des coordonnées X et Y (parité de la colonne/ligne) afin de savoir quel élément du bloc 2×2 est en entrée de l'IP. Une fois la somme des quatre pixels d'un même bloc calculée, il n'y a plus qu'à supprimer les deux bits de poids faible pour obtenir le calcul de la moyenne, un pixel de sortie ayant la même largeur de bus qu'un pixel d'entrée.

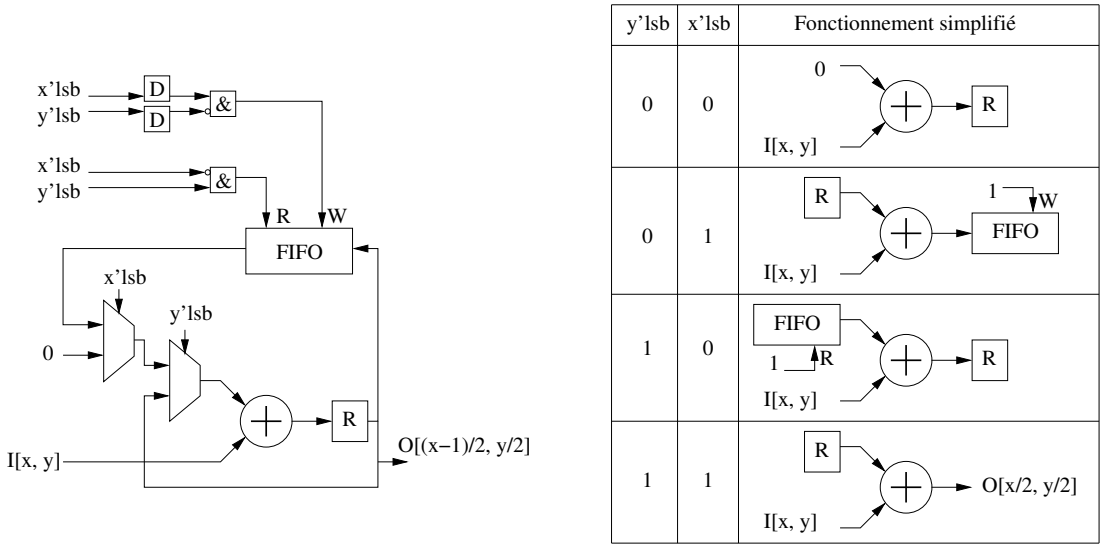


FIGURE 4.12 – **IP de sous-échantillonnage** - Sous-échantillonnage à partir d'un flot de pixels consécutifs. Les parités (bits de poids faible) de x et de y définissent le stade du calcul de l'IP

4.7 Recherche de points d'intérêt

La recherche de point d'intérêt correspond à une détection de maxima locaux dans les images générées par les DoG. Pour chaque pixel, on cherche dans une zone circulaire de rayon R si un autre pixel est supérieur au pixel étudié, afin de déterminer s'il est maximum dans cette zone. L'opérateur se déplace donc sur l'image par fenêtre glissante, de façon similaire à un opérateur de convolution traditionnel (figure 4.4), au détail près que la fenêtre est circulaire. Une fenêtre de pixels de l'image d'entrée est traitée à chaque coup d'horloge. Tous les signaux intermédiaires sont mémorisés dans des structures adaptées, afin que l'IP puisse traiter en parallèle tous les pixels qu'elle recouvre.

L'identification du pixel au centre de la fenêtre comme point d'intérêt est déclenchée par la validation de quatre tests, comme on peut le voir en figure 4.13 :

– La valeur de chacun des pixels de la zone circulaire est comparée à la valeur du pixel central. Si le pixel central est supérieur ou égal à tous les autres pixels de la fenêtre, il correspond donc à un maximum local. Il peut donc être détecté comme point d'intérêt. Voir figure 4.14.

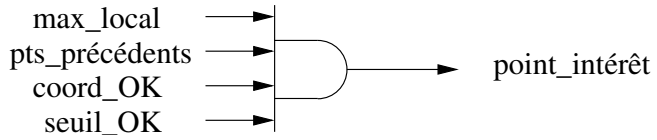


Figure 4.13 – **IP de détection de points d'intérêt : élément décisionnel** - les quatre tests en entrée de cette porte logique "ET" sont décrits dans les prochaines figures

- La non-détection comme point d'intérêt des pixels précédemment testés qui sont recouverts par la fenêtre circulaire de rayon R. Les pixels du demi-disque haut de la zone, et du rayon à gauche du pixel testé, ont déjà été testés par le détecteur. La mémorisation de R lignes de ces résultats permet de savoir si un de ces pixels a déjà été détecté comme point d'intérêt. Si un ou plus des pixels précédents dans cette zone circulaire est un point d'intérêt, le pixel central ne peut pas être un point d'intérêt. Ce mécanisme garantit une distance minimale entre deux points d'intérêt. Voir figure 4.15.
- Le seuil de bruit γ permet de filtrer les maxima locaux de trop faible valeur. Si le pixel central est supérieur ou égal à ce seuil, alors il pourra être détecté comme point d'intérêt. Ainsi, un maximum local dans une zone trop monotone de l'image de la caméra ne pourra pas être étudié par le réseau de neurones, ce qui lui évite de se baser sur des données trop peu pertinentes. Voir figure 4.16.
- Comme dans l'algorithme présenté en 3.1.3, les pixels au delà du bord de l'image sont considérés comme points d'intérêt potentiels. Il faut donc s'assurer que le pixel testé est assez loin du bord pour que tous les pixels dans la fenêtre circulaire soient valides. Ainsi, ce test est effectué uniquement sur les coordonnées du pixel central de la fenêtre. Si la fenêtre ne dépasse pas de l'image, alors le pixel central de la fenêtre peut être détecté comme point d'intérêt. En effet, un pixel trop près du bord a un voisinage partiellement inconnu (hors de l'image), il n'est donc pas possible de s'assurer qu'il est bien maximum local. Voir figure 4.17.

Les résultats de ces quatre tests se présentent sous forme de bits, à '1' si le test est passé. Une porte logique "et" permet ainsi de valider le pixel au centre de la fenêtre comme point d'intérêt. La valeur et les coordonnées de ce pixel sont transmises à l'IP suivante, responsable du tri des points d'intérêt.

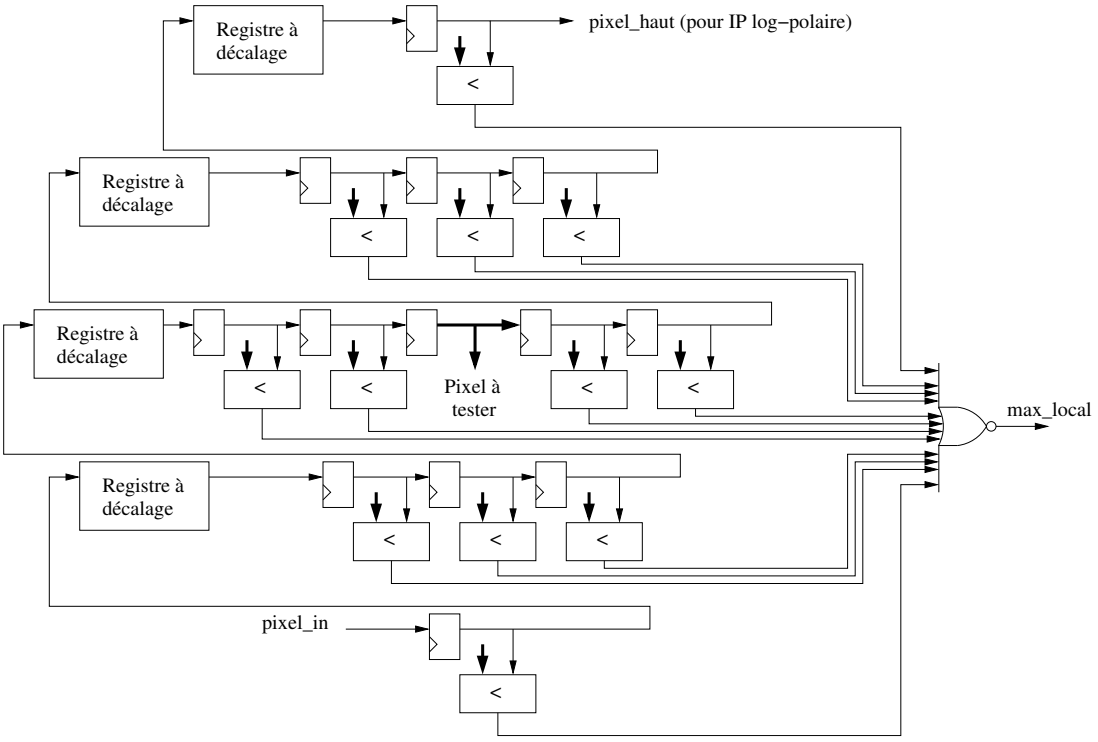


FIGURE 4.14 – **IP de détection de points d'intérêt : détection de maximum local** - exemple pour un rayon de recherche R=2. Les comparateurs testent si les pixels du disque sont supérieurs ou égaux au pixel à tester

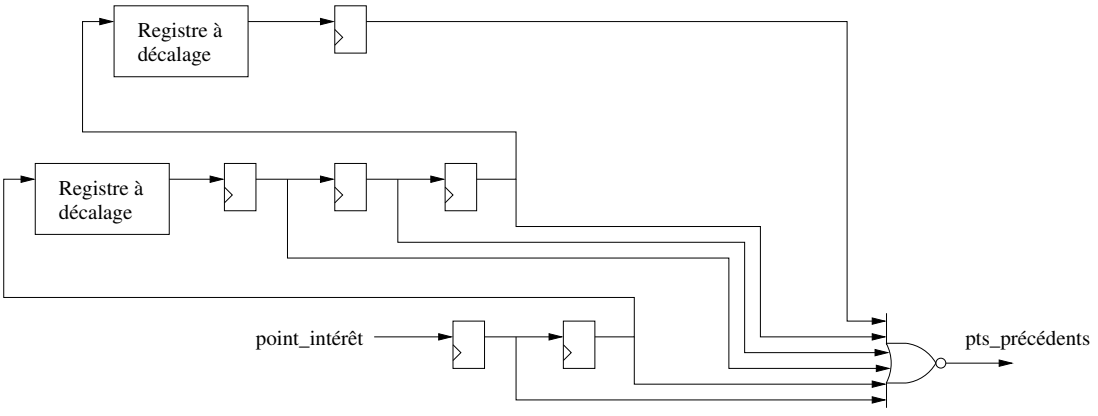


FIGURE 4.15 – **IP de détection de points d'intérêt : inhibition en cas de point d'intérêt précédent trop proche** - exemple pour un rayon de recherche R=2. Ce test se base sur la mémorisation du résultat de l'IP de recherche de points d'intérêt

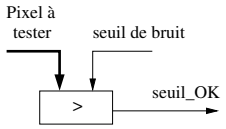


FIGURE 4.16 – **IP de détection de points d'intérêt : seuil de bruit** -

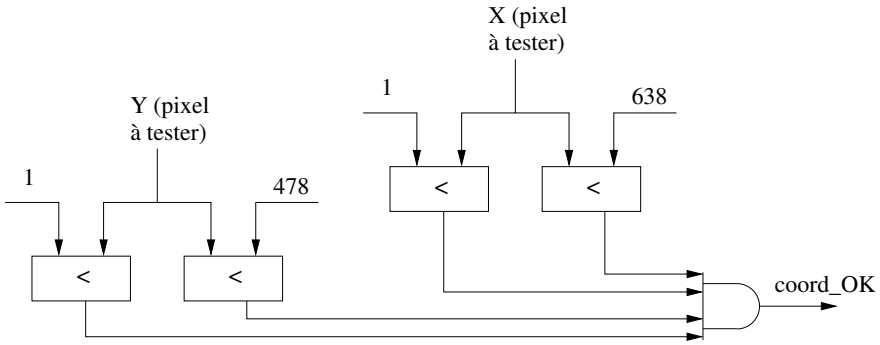


FIGURE 4.17 – **IP de détection de points d'intérêt : inhibition sur les bords de l'image** - exemple pour un rayon de recherche R=2, et une image de 640×480 pixels

Ces quatre tests représentent le cœur fonctionnel de cette IP. On remarque en figure 4.14 que les comparaisons des pixels du disque avec le pixel central se font de façon concurrente. Cette partie de l'IP nécessitera donc d'avoir environ $\pi \times R^2$ comparateurs de deux valeurs de pixels, et $2 \times R$ registres à décalage de dimensions allant de $W - 2$ à $W - 2 \times R$ pour stocker les lignes de pixels. Le deuxième test nécessite de stocker une information de 1 bit (point d'intérêt ou non) pour R lignes de pixels, soit R mémoires de tailles de l'ordre de W bits. Le troisième test n'a besoin que d'un comparateur de deux valeurs de pixels (l'une étant le seuil). Enfin, quatre comparateurs de coordonnées (11 bits pour du 1920×1080 pixels) et une LUT à quatre entrées (pour la fonction logique "et") suffisente pour s'assurer que le pixel central n'est trop près d'aucun des quatre bords.

La grande puissance de cette IP, qui fonctionne en temps-réel sur un flux de pixel, se paie par un coût élevé en opérateurs matériels. Par exemple, pour un rayon de recherche de $R = 20$ et des pixels sur 16 bits, le premier test coûtera environ 1256 comparateurs 16 bits, ce qui nécessite une bonne partie d'un FPGA de taille moyenne actuel. La valeur

de R décroît au fur et à mesure que l'on réduit la résolution, il faudra toutefois deux IP de cette taille pour la haute résolution, les autres résolutions ayant un impact divisé par quatre à chaque sous-échantillonnage. Ce coût en comparateurs augmente en fonction de R de façon quadratique, ce paramètre est donc à régler avec attention.

Dans le cas d'un rayon de recherche assez large, le nombre d'entrées des portes logiques en figures 4.14 et 4.15 peut devenir un frein à la cadence de fonctionnement de l'IP. Afin d'éviter ce désagrément, on regroupe les portes logiques par lignes, une porte supplémentaire permet de regrouper les résultats de ces portes après leur mise en registre.

4.8 Tri de points d'intérêt

Les points d'intérêt sont fournis à l'IP de tri par l'IP de recherche, sous la forme de leurs coordonnées et de la valeur des pixels correspondant. L'IP de tri classe ces ensembles de données comme des blocs indissociables correspondant chacun à un point d'intérêt. La cadence d'entrée des points d'intérêt dans l'IP représente le point le plus délicat de son développement : a priori, à chaque front d'horloge, un nouveau point d'intérêt peut entrer dans l'IP. Il est donc préférable d'avoir une IP capable d'effectuer le tri en un coup d'horloge. Dans cette optique, on dispose d'un avantage de taille : un seul élément à trier peut rentrer dans la liste à chaque coup d'horloge.

La figure 4.18 décrit le fonctionnement de l'IP : la valeur du pixel du nouveau points d'intérêt est comparée à celles de tous les éléments de la liste triée. Chaque point d'intérêt est stocké dans un registre (concaténation de sa valeur et de ses coordonnées). Si un nouveau point d'intérêt entre dans l'IP, il ira se placer immédiatement dans la case adéquate (juste en dessous des points d'intérêt de valeur supérieure ou égale). Tous les pixels de valeur inférieure se décaleront d'un cran vers le bas dans la liste triée. Le mécanisme est simple et modulaire : pour chaque rang de la liste triée, la valeur du pixel du nouveau point d'intérêt est comparée à :

1. la valeur du pixel du point d'intérêt de chaque rang. Si le nouveau pixel a une valeur strictement supérieure à celle du pixel en place, le point d'intérêt en place

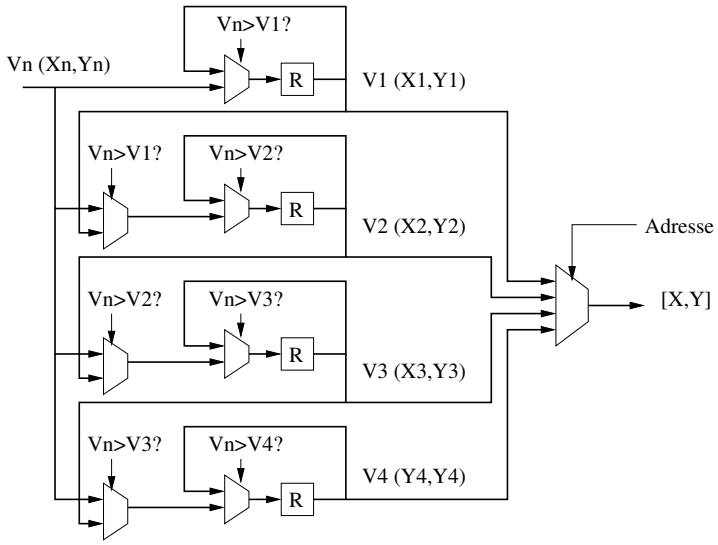


Figure 4.18 – **Tri de points d'intérêt** - Insertion du point d'intérêt de valeur Vn et de coordonnées (Xn,Yn) dans un tableau de 4 points triés

sera remplacé, et sera envoyé au rang de tri directement inférieur (si la taille de la liste le permet ; sinon il sera tout simplement abandonné).

2. la valeur du pixel du point d'intérêt de rang directement supérieur (sauf le premier, n'ayant aucun rang supérieur). Si le nouveau pixel a une valeur strictement supérieure au pixel de rang supérieur, cela signifie que le point d'intérêt de rang supérieur est remplacé, et va prendre la place du point d'intérêt en place.

Un multiplexeur permet ainsi pour chaque rang (sauf le premier) de sélectionner le point d'intérêt entrant ou le point d'intérêt de rang supérieur, en comparant le pixel d'entrée au pixel de rang supérieur. Un autre multiplexeur permet de sélectionner pour chaque rang la valeur en place ou la sortie du multiplexeur précédent (ou le point d'intérêt entrant pour le premier rang), selon que le point d'intérêt devra être remplacé ou non (pixel d'entrée supérieur au pixel en place).

Le fonctionnement en sortie est semblable à une mémoire : une adresse indique l'indice du point dont on souhaite récupérer les coordonnées. Un multiplexeur permet de recopier les coordonnées correspondantes en sortie de l'IP. La valeur des pixels triés n'étant pas utile à la suite des traitements, la sortie de l'IP ne contient que les coordonnées.

4.9 Mise en forme des caractéristiques log-polaires

Les caractéristiques générées par le système de vision permettent au robot d'appréhender son environnement visuel. Le système doit générer pour chaque image fournie par la caméra, une caractéristique pour chaque point d'intérêt.

L'IP de mise en forme des caractéristiques reçoit des informations de l'IP de recherche de points d'intérêt, et de l'IP de tri :

- L'IP de recherche de points d'intérêts lui fournit les pixels qui seront utilisés pour générer les caractéristiques. C'est le signal "pixel_haut" en figure 4.14.
- L'IP de tri lui indique chaque nouveau point d'intérêt détecté dans l'image, son rang dans la liste triée, et ses coordonnées dans l'image.

Un découpage en sous-IP permet la gestion indépendante des caractéristiques, par index dans la liste. Une sous-IP de mise en forme log-polaire sera responsable de la génération de la caractéristique correspondant au point d'intérêt lié à son index. L'IP de tri pouvant remplacer un point d'intérêt par un autre lorsque la liste triée est pleine, une sous-IP doit, à la fin de l'image, avoir généré la caractéristique correspondant au point d'intérêt définitif.

Le schéma en figure 4.19 permet de voir l'organisation de l'IP de mise en forme des caractéristiques, utilisant d'un coté des sorties des IP de recherche et de tri, et de l'autre, permettant une lecture des caractéristiques générées en adressant chaque sous-IP comme une mémoire, une deuxième adresse permettant de sélectionner la sous-IP à lire.

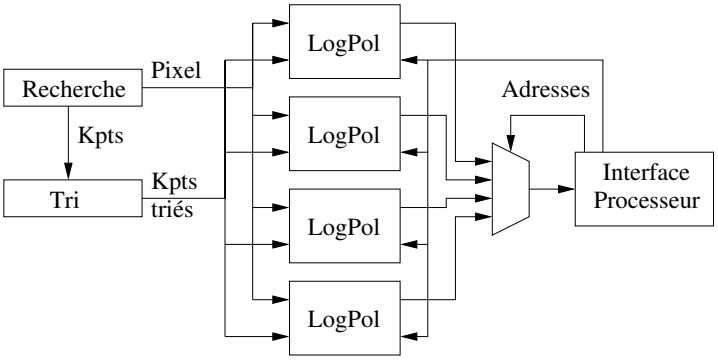


FIGURE 4.19 – **Génération des caractéristiques log-polaires : schéma global** - pour une liste de 4 points d'intérêt

L'interface processeur dépend de la plate-forme sur laquelle le système de vision sera déployé. Dans les tests qui ont été effectués, une interface Avalon a permis à un processeur NIOS II de lire les caractéristiques log-polaires pour les envoyer au réseau de neurones via Ethernet.

Le fonctionnement de chaque sous-IP "logpol" en figure 4.19 est présenté en figure 4.20.

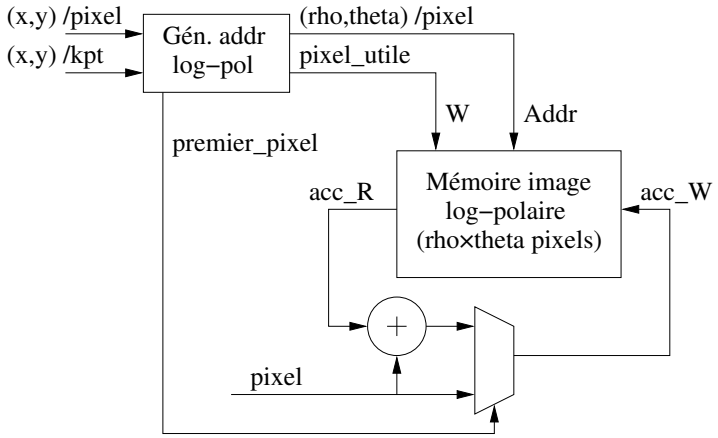


Figure 4.20 – **Génération des caractéristiques log-polaires : pour chaque point d'intérêt** - contenu de chaque bloc "logpol" de la figure 4.19

Les coordonnées de deux pixels sont tout d'abord étudiées : celles du point d'intérêt pour lequel on génère la caractéristique, et celles du pixel d'entrée de l'IP (sortie de l'IP de recherche de points d'intérêt). Le bloc de génération d'adresse permet, à partir de ces deux jeux de coordonnées, de calculer les coordonnées log-polaires du pixel d'entrée, et de tester si ce pixel est dans le voisinage du point d'intérêt, et ainsi activer le cœur de calcul de l'IP. Un signal supplémentaire est généré par ce bloc : il s'agit, pour chaque jeu de coordonnées (ρ, θ), d'identifier le premier pixel de l'image d'entrée à être utilisé. En effet, pour un pixel log-polaire, on calculera la moyenne des pixels correspondant dans l'image d'entrée. La première étape étant de calculer la somme des pixels, on utilise un additionneur, dont on relie la sortie à l'entrée d'une mémoire double port. Un multiplexeur permet de sélectionner la valeur qui sera mise en mémoire, à savoir la valeur du pixel d'entrée si le signal "premier_pixel" est actif, sinon la sortie de l'additionneur.

La lecture des caractéristiques se faisant de façon séquentielle par l'interface processeur, l'opérateur de division permettant le calcul de la moyenne est unique pour chaque IP de mise en forme des caractéristiques, comme on peut le voir en figure 4.19. Le calcul de cette division peut aussi être exécuté logiciellement, sur le processeur embarqué du système. Cette solution permet de réduire la consommation de ressources de cette IP, et de gagner en précision du calcul des caractéristiques en virgule flottante, le réseau de neurones utilisant cette représentation pour ses données.

Un dernier aspect important de la sous-IP de transformation log-polaire est la gestion de la mémoire : cette IP utilise une topologie de type ping-pong, comme on peut le voir en figure 4.21. Ainsi, la lecture des caractéristiques de l'image précédente par le processeur se fait en parallèle de l'écriture des caractéristiques de l'image présente.

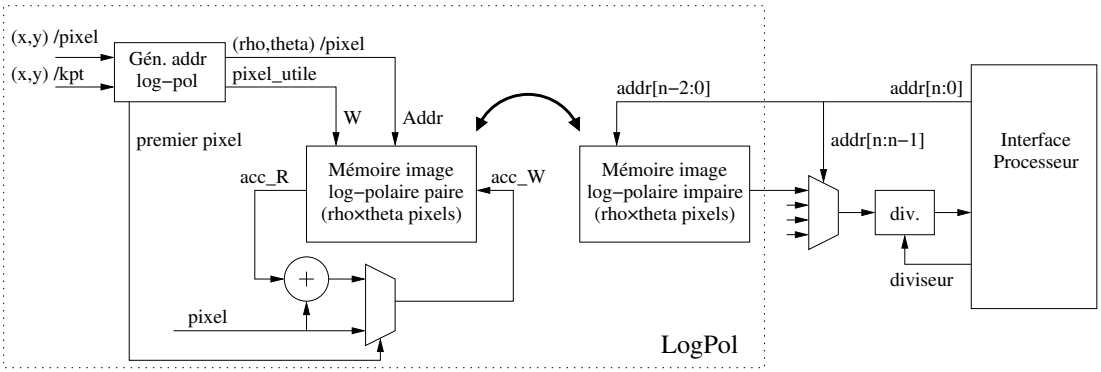


FIGURE 4.21 – **Génération des caractéristiques log-polaires : topologie ping-pong** - la flèche en gras fait apparaître l'utilisation de la topologie ping-pong.

4.10 Conclusion

Ce chapitre a permis de présenter les différents composants matériels réalisés au cours de ce projet, qui permettent d'exécuter la totalité de la chaîne algorithmique du système de vision. Toutefois, la connexion de cet ensemble d'IP à une caméra et au réseau de neurones n'a pas été présenté, car ces parties du système sont dépendantes de l'environnement des IP proposées. La caméra est, à l'heure de la publication de ce document, inadaptée aux besoins du robot, et une solution alternative est recherchée. Concernant la connexion au réseau de neurones, la plate-forme de démonstration

réalisée par Laurent Fiack utilise un processeur NIOS-II pour envoyer par Ethernet les caractéristiques générées par le système. La carte de développement FPGA utilisée pour cette démonstration utilise un FPGA à bas coût, pour valider une version allégée du système complet. Ainsi, la connexion au réseau de neurone n'est elle non plus pas fixée définitivement.

Le choix d'une plate-forme exécutant la totalité de la chaîne algorithmique sous forme d'IP ou déportant certains calculs sur un processeur embarqué reste à faire, en fonction des résultats obtenus au chapitre précédent, et de l'utilisation des ressources FPGA des différentes IP. Il est ainsi nécessaire d'étudier la répartition des ressources FPGA des IP présentées afin de mieux choisir la topologie du système de vision.

Chapitre 5

Déploiement

5.1 Plate-forme de traitements

5.1.1 Partitionnement logiciel/matériel

Le chapitre 3 conclut sur la nécessité de déployer la plupart des traitements sous forme d'accélérateurs matériels, les extractions des voisinages des points d'intérêt restant les seuls traitements logiciels. Si les IP matérielles sont traitées dans le chapitre précédent, la partie logicielle du système reste un aspect problématique du SoC. Pour que le système respecte l'intention de départ de déployer l'application sur un SoC, un système logiciel doit être embarqué au SoC, qu'il soit composé d'un processeur hard-core ou de plusieurs processeurs soft-core. Le premier cas nécessite un circuit particulier doté d'un processeur hard-core suffisamment puissant, ainsi que d'une zone FPGA permettant d'héberger toutes les IP nécessaires. Le circuit Zynq de Xilinx est un bon exemple. Dans le deuxième cas, un FPGA traditionnel peut être utilisé, mais la taille de ce dernier devra être particulièrement élevée, afin de pouvoir héberger, en plus des IP, la partie logicielle soft-core.

Une solution plus attrayante serait de reporter ces traitements logiciels sur le PC embarqué du robot. Si cette solution représente un pas en arrière vis-à-vis de l'autonomie du système de vision, l'utilisation du PC embarqué du robot peut se révéler peu coûteuse pour ce dernier, ce qui rendrait la solution envisageable. Cette solution permettrait d'exécuter l'application dans les contraintes temps-réel fixées, tout en permettant l'utilisation d'une gamme plus large de circuits FPGA.

Les extractions de voisinages des points d'intérêt sont assez complexes à déployer sous forme d'IP homogènes à la solution matérielle proposée, et se trouvent en fin de chaîne des traitements. Ces traitements sont donc parfaitement adaptés à un déploiement sur le PC embarqué du robot, si ce dernier peut supporter la charge de calcul supplémentaire. Un obstacle de taille est le canal de communication qui relie le FPGA au PC : il doit pouvoir supporter l'augmentation de débit résultante (voir section 3.2.4).

Si on regarde dans les mesures faites en 3.2.1, on peut voir que sur un seul cœur de processeur, les extractions des voisinages des points d'intérêt, pour les paramètres donnés, prennent en tout un peu plus de 1 ms. En sachant que les traitements se feront à une cadence de 25 images par seconde, on peut estimer qu'un PC un peu moins puissant nécessitera 50 ms toutes les secondes sur un seul cœur de processeur. Sur un PC doté d'un processeur simple cœur, cela équivaut à une charge de processeur de 2%.

Il semble raisonnable de penser qu'une telle surcharge du PC portable du robot soit admissible du point de vue des traitements normaux de ce PC.

Pour résumer, trois solutions se présentent :

- le choix d'un SoC disposant d'un processeur puissant : le Zynq de Xilinx est un bon exemple actuel, doté d'un processeur plus puissant que le Cortex A8 utilisé pour les mesures. On choisira alors d'exécuter les extractions de voisinages des points d'intérêt sur ce processeur, le reste des traitements étant accéléré par des IP matérielles. Cette solution permet de déployer l'application sur un seul SoC, tout en réduisant la charge du canal de communication du PC embarqué. On peut voir en figure 5.1 l'agencement de cette plate-forme, telle qu'elle serait utilisée pour le SoC de vision.
- un FPGA traditionnel sans processeur hard-core intégré. Les traitements logiciels décrits dans la solution précédente pourront être déportés sur le PC du robot, un processeur soft-core prenant en charge les communications avec le PC. À l'inverse, ils pourront être déployés sur un système multiprocesseur soft-core déployé dans le FPGA, ce qui nécessiterait un FPGA de grande taille.
- le déploiement de tout le système sous forme d'IP matérielles. Les IP d'extraction et de mise en forme des voisinages des points d'intérêt doivent alors être finali-

sées et mises en place dans le système final. Les performances de la plate-forme résultante seront reproductibles de façon plus sûre.

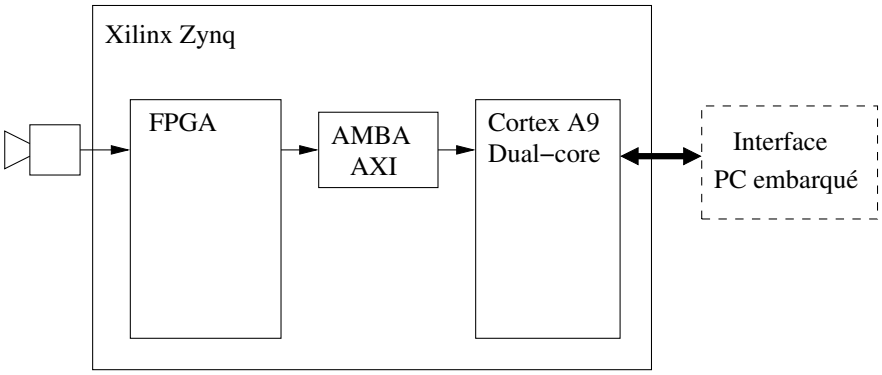


FIGURE 5.1 – **Architecture du SoC sur un composant Zynq de Xilinx** - Les éléments internes du Zynq ne sont pas représentés à l'exception du processeur, de la matrice FPGA, et de l'interconnexion AMBA AXI

La solution la plus intéressante à court terme reste la première, les circuits envisagés étant les plus adaptés aux architectures hybrides logiciel/matériel, telles qu'elles sont décrites dans ce document. À long terme, la dernière solution semble plus séduisante : l'équipe Neurocybernétique envisage de déployer des réseaux de neurones sur FPGA, ce qui permettrait d'unifier le système bio-inspiré et peut-être de gagner en puissance de calcul. Les imagettes log-polaires enregistrées dans des mémoires indépendantes permettront une lecture parallèle de celles-ci par le réseau de neurones.

5.1.2 Interface réseau

L'interface standard sur le robot est l'Ethernet. Un switch Gigabit Ethernet permet la connexion des différents éléments du robot, qu'ils soient capteur ou actionneurs, au réseau de neurones. Le système de vision n'échappe pas à cette règle, pour un souci d'intégration aux outils de développement du réseau de neurones. Le FPGA de vision est relié au réseau de neurones par un module Ethernet, présent sur la carte de développement.

Dans le cas présent, un prototype sur carte Altera (Fia12) utilise un processeur soft-core NIOS II pour récupérer une image de DoG et l'envoie par Ethernet à un ordinateur

à une cadence de 9 images par seconde, de 320×240 pixels de 16 bits, le débit équivalent est alors de 1,3 Mo/s. Pour comparaison, les 6 jeux de 20 imagettes log-polaires 5×36 pixels 32 bits représentent plus de 2 Mo/s pour une cadence de 25 images par seconde. Pour le moment, cette interface est encore en développement, ces performances ne sont représentatives que d'une version précoce de cette partie du système.

5.2 Validation de l'architecture

Les IP proposées sont décrites en VHDL, et validées par simulation fonctionnelle. L'outil de simulation utilisé est ISim de Xilinx. Le principe d'un tel outil est d'utiliser des stimuli en entrée des IP, et de fournir les résultats fonctionnels générés en sortie.

Afin d'obtenir une validation pertinente, il est utile d'obtenir un comportement précis au bit près (bit-accurate) des IP par rapport à un modèle logiciel de ces IP. Cette section présente le modèle logiciel qui est fait des IP, ainsi que les outils mis en place pour comparer pixel par pixel les résultats fournis par les IP en simulation.

5.2.1 Modèle logiciel des IP

La validation des IP et le réglage de leurs paramètres étant des étapes fastidieuses de la conception, un modèle algorithmique en C se révèle particulièrement utile. Une modélisation logicielle des algorithmes et des types de données des IP permet d'accélérer considérablement ces étapes, en obtenant des résultats fonctionnels plus rapidement qu'en simulation HDL. Ce modèle, codé en langage C dans le cas présent, permet d'évaluer plus rapidement les images résultant des traitements des IP, en modulant divers paramètres de l'architecture et de l'application (largeur des bus des pixels/coefficients, données en virgule fixe/flottante, seuil de détection d'une IP, etc). Un modèle SystemC aurait permis de modéliser plus finement les IP, et reste envisageable, le modèle C restant un point de départ logique de celui-là. Des fichiers contenant les résultats numériques des différents traitements sont aussi générés, afin de les comparer aux résultats de simulation du code HDL des IP.

Les premières étapes de développement de ce modèle consistent à valider le comportement des IP telles qu'on les a décrites : les erreurs éventuelles de description (largeurs

de bus inadaptées, valeurs de coefficients, etc) peuvent êtres détectées avant même d'effectuer la simulation du code VHDL. Les résultats de la version logicielle de l'application peuvent être comparés avec les résultats du modèle logiciel des IP afin de connaître la précision de ces IP. Une fois que le modèle logiciel des IP est considéré fonctionnel, les fichiers de résultats sont conservés pour être comparés aux résultats de simulation du code HDL.

Les paramètres des IP peuvent être choisis à l'aide de ce modèle : à l'aide des fichiers de résultats numériques, on peut effectuer des comparaisons rapides de l'influence des paramètres sur le comportement ou la précision de l'IP. Ces paramètres seront choisis par l'utilisateur en fonction des besoins en précision (largeur des bus des pixels et des coefficients), du fonctionnement souhaité (rayons de recherches, seuils de bruit, etc.), ainsi que des contraintes pratiques (dimensions de l'image). Les paramètres évoluant selon les besoins de l'utilisateur (robot mobile en missions de navigation, robot fixe appliqué à la détection d'expressions faciales, etc.), on comprend bien l'utilité d'IP paramétrables et génériques. L'utilisateur final n'étant pas nécessairement un utilisateur averti de la conception d'IP, un fichier de configuration contenant les divers paramètres permettra de déployer un SoC adapté aux besoins sans avoir à modifier le code des IP.

Les communications entre les IP ayant un fonctionnement simple et standardisé dans l'architecture proposée (un pixel par cycle d'horloge en entrée et en sortie, modulé par un signal enable), la modélisation temporelle n'est pas indispensable pour obtenir des résultats fonctionnels valides. La simulation HDL permettra de valider par la suite les comportements temporels des IP, notamment pour les registres à décalages ou FIFO internes aux IP.

Le système d'IP complet, ainsi que les traitements logiciels qui en dépendent, sont modélisés, ce qui permet d'étudier le fonctionnement global du système. Le SoC n'étant pas déployé dans une version complète, on peut ainsi connaître le résultat des transformations log-polaires des données générées par les IP. Ce modèle permet ainsi l'intégration des flots de développement logiciel et matériel au sein d'une même étape de validation. Les communications entre les IP du FPGA et l'unité de traitement logicielle étant toujours en cours de réalisation, il est important de pouvoir valider le système dans sa globalité. Les communications IP/traitements logiciels seront alors considérées

transparentes.

Par ailleurs, différentes architectures d'IP et différents algorithmes peuvent être étudiés à l'aide de ce modèle. Dans la figure 5.2, on compare les images générées par un opérateur de Sobel et un opérateur de Prewitt pour le calcul de l'intensité du gradient. Dans les deux cas, le résultat du filtre est stocké sur des pixels 16 bits non-signés, l'image d'entrée étant codée sur 8 bits par pixel, en niveaux de gris.

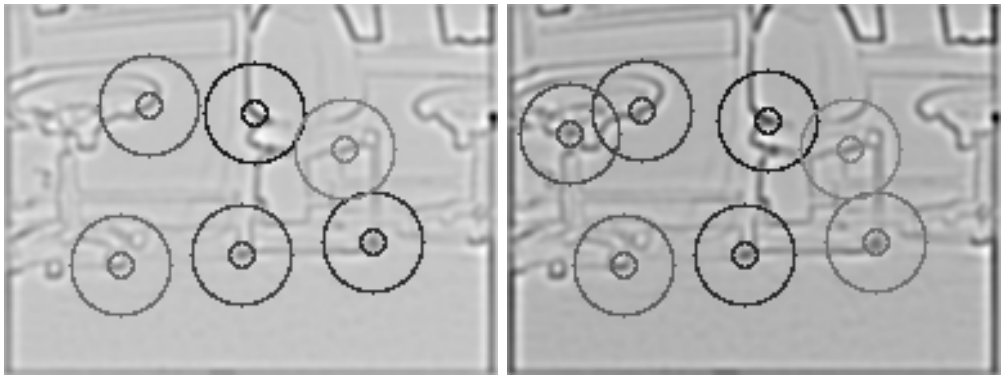


Figure 5.2 – **Test de deux algorithmes sur le modèle** - Intensité de gradient : DoG obtenue avec un modèle utilisant l'algorithme de Prewitt à gauche, l'algorithme de Sobel à droite

Les limites principales de ce modèle sont les suivantes :

- La traduction du modèle vers la description matérielle reste délicate. Ce modèle permet de valider le fonctionnement des IP, mais n'est pas un outil de traduction de code. Des outils de traduction automatique tels que Handel-C ou Catapult-C pourraient être utilisés si le besoin se présentait (MS09).
- L'utilisation des ressources du FPGA n'est pas modélisée : la synthèse des IP est nécessaire pour obtenir ces informations. Si le modèle permet d'évaluer rapidement les résultats d'une IP, la variation de l'utilisation des ressources en fonction de paramètres du modèle n'est pas couverte. Comme pour les ressources utilisées, la cadence de fonctionnement maximale ne sera disponible qu'après placement et routage de l'architecture sur le FPGA.

– Le comportement temporel de l'architecture n'est pas modélisé. La synchronisation des flux de pixels n'est pas vérifiée par le modèle : les lignes à retard ne sont pas modélisées, et une erreur dans la valeur d'un retard devra être détectée à la simulation HDL.

Cela dit, la validation des algorithmes, le choix de paramètres optimaux, ainsi que le gain de temps lors de la correction du code HDL de l'IP, justifient largement le modèle logiciel. Le choix des algorithmes sera fait tout d'abord grossièrement, à l'aide des images générées par le modèle, puis par comparaison numérique avec les résultats de l'algorithme logiciel d'origine.

Pour le choix des paramètres, le modèle est lancé à de nombreuses reprises pour différentes valeurs des paramètres, les différents résultats sont comparés par la suite. Les mêmes mesures à partir du code HDL prendraient beaucoup plus de temps.

Pour la correction du code HDL, les valeurs des calculs intermédiaires sont calculées et rapportées par le modèle. Lors de la simulation HDL de l'IP, l'identification des valeurs intermédiaires erronées permet de localiser les parties du code HDL qui doivent être corrigées.

Le choix des largeurs des bus des pixels permet d'accéder à un compromis entre la consommation en ressources FPGA, et la précision des résultats du SoC. Il est important d'évaluer l'impact des paramètres sur l'utilisation des ressources des IP, afin de mieux appréhender ce compromis. La section 5.3 s'applique à cette tâche, pour les IP les plus imposantes. Le fonctionnement efficace du réseau de neurones dans les diverses tâches qui lui sont données dépend de la précision des résultats. L'utilisateur final pourra adapter les paramètres des IP en fonction de la taille des FPGA dont il dispose, et des besoins de précision de son application robotique propre.

Sur la figure 5.3, on peut voir les résultats visuels d'une même DoG, pour différentes tailles de bus (coefficients du noyau Gaussien et pixels des images intermédiaires). Si la précision des niveaux de gris n'est pas visible au delà de 8 bits par pixel (256 niveaux de gris uniquement sur ce document), on peut tout de même remarquer une dégradation de l'image liée à la baisse de dynamique des pixels de l'image (résultats plus visibles sur la ligne du bas). Parallèlement, on remarque que la somme des coefficients du noyau Gaussien a bien moins d'importance sur la qualité du résultat : la ligne du bas correspond

à une somme de coefficients stockée sur 5 bits, et fournit de bien meilleurs résultats que la colonne de droite correspondant à des pixels sur 8 bits. Une raison majeure de ceci est que sur une image d'entrée quelconque, l'image d'intensité de gradient sera très éloignée du maximum théorique, et les valeurs des pixels en aval de la chaîne de traitements ne seront répartis que dans la partie basse de la dynamique des pixels. Les bits non-utilisés dans cette majorité des cas doit toutefois être gardée afin de garantir un fonctionnement correct dans le cas où l'image d'entrée permet à un moment de maximiser le gradient sur une zone relativement large.

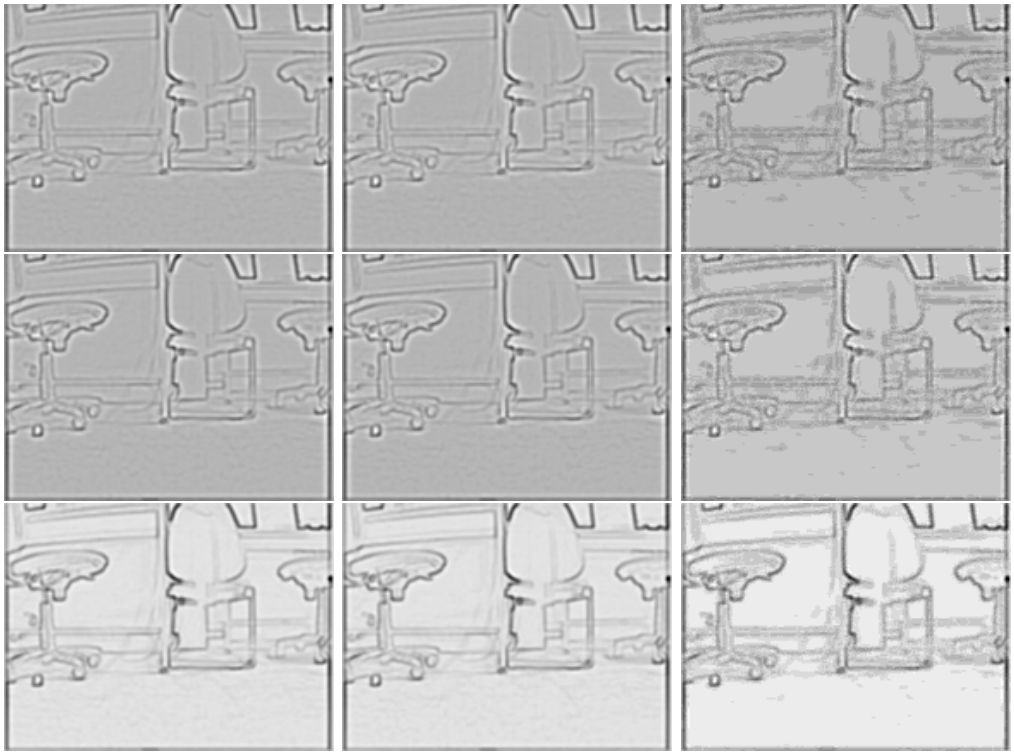


Figure 5.3 – **Effet du type de données sur les résultats** - résultats obtenus à l'aide du modèle d'architecture. De gauche à droite, pixels sur 16, 12, 8 bits. De haut en bas, somme des coefficients= 2^{16}, 2^{10}, 2^5

Un premier choix des largeurs de bus des pixels et des coefficients peut alors se faire, en fonction des besoins de précision du robot. L'impact de ces paramètres sur l'utilisation du FPGA peut restreindre ce choix, en fonction de la taille du FPGA et des autres IP. La section 5.3 s'applique à étudier les ressources utilisées en fonction des paramètres de l'application et de l'architecture des IP.

5.2.2 Simulation

La simulation des IP permet de valider le fonctionnement du code HDL des IP (VHDL dans le cas présent). On note deux types de simulations : la simulation fonctionnelle, qui ne tient pas compte des temps de propagation dans le circuit, et qui permet d'effectuer une première validation des IP. La simulation temporelle, quant à elle, se fait après placement/routage du design sur le FPGA ciblé, et prend en compte l'aspect temporel des signaux du FPGA. Cette deuxième option est utile quand des erreurs sont détectées sur carte, mais pas lors de la simulation fonctionnelle.

La simulation du code HDL est effectuée afin d'assurer un fonctionnement identique à celui du modèle logiciel des IP. La simulation fonctionnelle est effectuée dans le cas présent avec l'outil ISim de Xilinx.

La librairie VHDL `std.textio` permet l'utilisation de fichiers texte en entrée et en sortie de la simulation. Des fichiers de génération de stimuli (testbench) sont donc écrits pour tester les IP en lisant des images au format texte, tout en écrivant les résultats de simulation dans d'autres fichiers.

La lecture d'image au format texte représente bien sûr une étape supplémentaire. Un programme écrit en C, utilisant la librairie OpenCV, permet de lire des images, et de les transcrire au format texte. Un deuxième programme, permet d'effectuer la conversion en sens inverse, afin d'obtenir les images correspondant aux résultats générées par les IP lors de la simulation. Cette étape de vérification visuelle permet notamment de détecter certains types de problèmes de synchronisation de données dans l'IP.

Les fichiers générés par le modèle et par la simulation sont comparés à l'aide de l'outil UNIX diff. Un comportement précis au bit près étant souhaité, les fichiers provenant de la simulation et de l'éxécution du modèle logiciel doivent être identiques. Une différence de valeurs correspond à une incohérence entre le modèle et la description matérielle de l'IP. Il faut dans ce cas se pencher sur le fonctionnement interne de l'IP pour en comprendre la cause, et ainsi adapter le code VHDL (ou le modèle, dans le cas d'une erreur de modélisation). Si les signaux d'entrée et de sortie de l'IP sont insuffisants pour comprendre les erreurs, la comparaison de fichiers peut se faire sur les signaux internes de l'IP. Il faut alors modifier le code de l'IP pour pouvoir lire les signaux depuis le fichier

de testbench, et générer les fichiers des valeurs intermédiaires dans le modèle logiciel.

Il est important de ne pas sous-estimer les erreurs de modélisation logicielle. Par exemple, lors du passage d'un nombre réel décimal à un nombre entier, un mécanisme d'arrondi doit être mis en place dans le modèle logiciel, pour correspondre au mécanisme mis en place automatiquement par la bibliothèque VHDL `ieee.math_real`.

5.3 Réglage des paramètres

Afin d'obtenir un SoC efficace, sur un circuit de taille raisonnable, un réglage des paramètres de l'architecture et de l'application est nécessaire. Les mesures permettant de juger de ce compromis sont les résultats de synthèse (utilisation des ressources FPGA), et les résultats du modèle logiciel vu précédemment (précision des résultats).

Le choix des paramètres en vue de la précision des résultats est du ressort de l'utilisateur final, à savoir le roboticien sur le point d'embarquer la plate-forme de vision sur son robot. Des séries de synthèses ont été faites pour les IP nécessitant le plus de ressources FPGA, les résultats sont analysés ici en fonction des paramètres afin de guider le choix de leurs valeurs. Les choix qui sont faits ici pour le système de vision découlent d'une vision purement architecturale, en gardant toutefois à l'esprit qu'une plus grande précision est préférable du point de vue du robot.

5.3.1 Ressources utilisées par IP

Les différentes IP sont synthétisées pour un jeu de paramètres choisi après le réglage des IP les plus complexes, afin d'être plus proche de l'utilisation finale conseillée du SoC. On utilisera ainsi 17 bits de données par pixel, des fenêtres de convolution de 7 coefficients codés sur 24 bits pour le filtrage Gaussien, des rayons de recherche de 20, 10, et 5 pixels pour les recherches de points d'intérêt dans les DoG. Les tris de points d'intérêt se limiteront à N=20 points d'intérêt trouvés, dans le cas où l'image en comporterait davantage. Le seuil de détection des recherches de points d'intérêt n'influe pas sur les ressources nécessaires aux IP, pour information sa valeur est fixée à 2.

IP (nombre dans le SoC)	Registres	LUT	BRAM 36k	DPS48E1
Gradient	160	281	1	0
Gaussienne(1) HF (×2)	815 (×2)	800 (×2)	3 (×2)	13 (×2)
Gaussienne($\sqrt{2}$) HF	825	802	3	13
Sous-éch. HF⇒MF	147	132	1	0
Gaussienne(1) MF	799	782	3	13
Gaussienne($\sqrt{2}$) MF	809	784	3	13
Sous-éch. MF⇒BF	141	127	1	0
Gaussienne(1) BF	783	762	3	13
Gaussienne($\sqrt{2}$) BF	793	764	3	13
DoG (×6)	21 (×6)	80 (×6)	0	0
Ligne à retard (×6)	11 (×6)	24 (×6)	1 (×6)	0
Recherche R=20 HF (×2)	22174 (×2)	23813 (×2)	30 (×2)	0
Recherche R=10 MF (×2)	5579 (×2)	6233 (×2)	15 (×2)	0
Recherche R=5 BF (×2)	1423 (×2)	1729 (×2)	8 (×2)	0
Tri de points d'intérêt (×6)	1172 (×6)	2008 (×6)	0	0
Architecture totale	69018	74402	119	91

TABLE 5.1 – **Résultats de synthèse sur Virtex 6** - circuit XC6VLX550T - pixels sur 17 bits, noyaux de convolution de 7 coefficients sur 24 bits, résolutions HF : 640×480, MF : 320×240, BF : 160×120

On remarque dans la table 5.1 que l'utilisation du FPGA est très différente d'une IP à une autre. Les recherches de points d'intérêt utilisent ici 106 blocs BRAM sur les 119 de l'architecture complète (soit 89%). Les mêmes IP utilisent 58352 registres et 63550 LUT, soit dans les deux cas 85% de l'architecture complète. Les blocs DSP48E1 sont quant à eux exclusivement utilisés par les IP de filtrage Gaussien, à hauteur de 91 blocs, ce qui est assez peu pour la famille Virtex 6 (le plus petit dispose de 288 blocs DSP48E1).

5.3.2 Filtrage Gaussien

L'IP de filtrage Gaussien se paramètre selon plusieurs valeurs, qui influencent l'utilisation du FPGA :

- le coefficient σ, valant 1 ou $\sqrt{2}$ dans le SoC
- la largeur du bus de données des pixels d'entrée et de sortie
- la largeur du bus de données des résultats intermédiaires de MACC, donc des coefficients
- les dimensions du noyau de coefficients Gaussiens

Les figures suivantes permettent d'entrevoir l'utilisation des ressources FPGA selon l'évolution de ces paramètres. L'utilisation des ressources du FPGA (on considère surtout les registres, les LUT, les blocs mémoire BRAM et les blocs de calcul DSP48E1) évolue selon tous ces paramètres, il est donc difficile de proposer une seule figure pour chacune de ces ressources. Deux jeux de figures sont présentés, le premier fige le paramètre σ et la largeur du noyau de convolution, permettant de voir l'influence des largeurs de bus des pixels et des coefficients. Le deuxième jeu de figures fige les largeurs de bus, et permet ainsi d'appréhender l'impact des deux autres paramètres.

La figure 5.4 présente les résultats de synthèse obtenus pour un noyau de convolution de 7 coefficients de large et pour un paramètre $\sigma = \sqrt{2}$. Les résultats correspondants à des largeurs de noyaux de convolution de 3, 5, 7 et 9 coefficients sont présentes en annexes, en plus grandes tailles pour une meilleure lisibilité. Cette figure montre que la largeur de bus des coefficients n'influe pas sur l'utilisation des blocs mémoire BRAM, qu'un passage de 18 à 19 bits par pixel augmente d'un coup (les 23 largeurs de bus de coefficients donnant toutes les mêmes courbes, seules les deux extrêmes sont présentées). L'impact des largeurs de bus sur le nombre de registres et de LUT nécessaires à l'IP est relativement limité en deçà de certaines valeurs. À partir de 18 bits de données

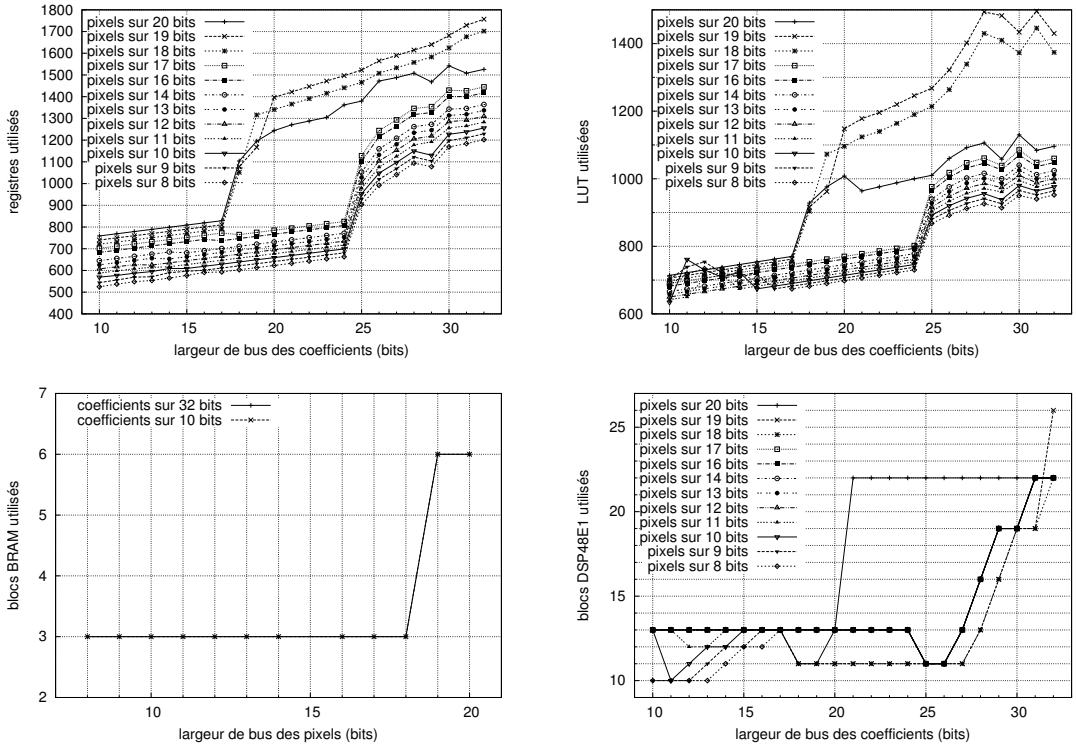


FIGURE 5.4 – **IP de filtrage Gaussien : impact des paramètres 1** - Utilisation d'un FPGA de type Virtex 6 pour différentes largeurs de bus des pixels et des coefficients, pour une image de 640×480 pixels, $\sigma = \sqrt{2}$, et un noyau de convolution de largeur 7

pour les pixels et pour les coefficients de convolution, par contre, la consommation de registres et LUT semble franchir un palier. Ce palier semble être atteint à partir de 25 bits de données par coefficient de convolution, pour les architectures utilisant moins de 18 bits pour stocker les pixels. Le nombre de blocs DSP48E1 semble à peu près stable en dessous de 20 bits par pixel, jusqu'à 26 bits par coefficient de convolution. Au delà, le nombre de ces blocs augmente fortement.

Au vu de ces résultats de synthèse, des pixels sur 17 bits et des coefficients sur 24 bits semblent apporter un compromis intéressant sur les ressources utilisées et la précision des calculs. Cela dit, il faut garder à l'esprit que les filtrages Gaussiens font partie des premiers traitements, et sont cascadés. La précision de leurs résultats est alors particulièrement importante. Du fait des résultats de la table 5.1, doubler les blocs BRAM utilisés par les filtrages Gaussiens semble raisonnable,

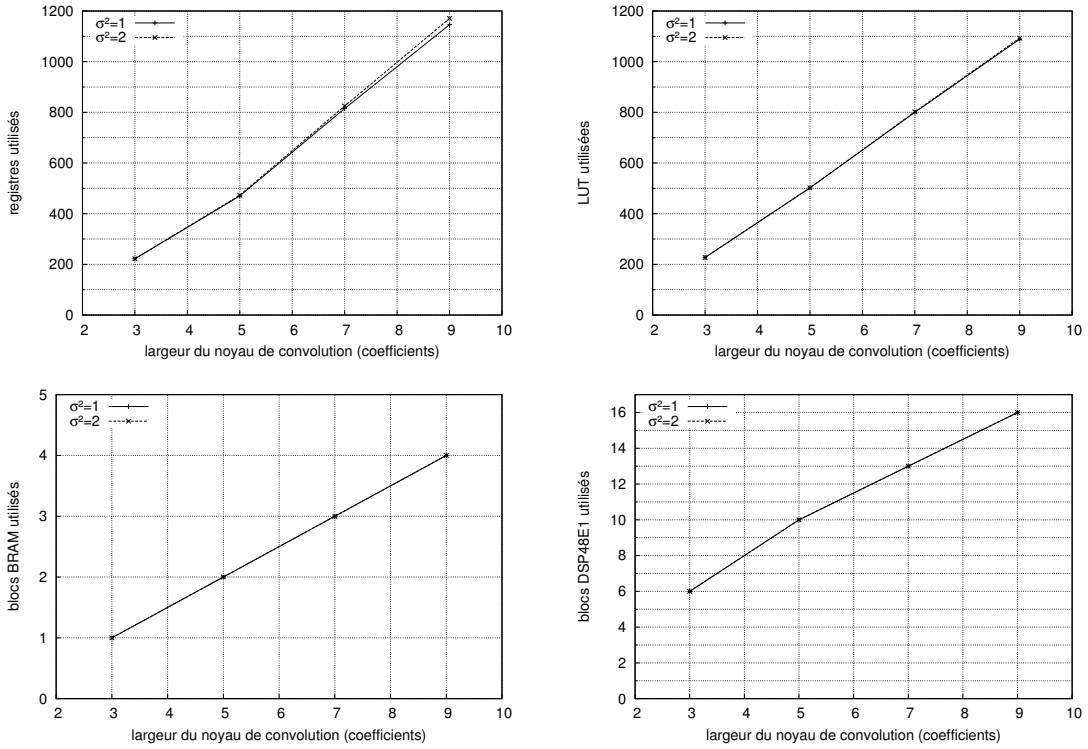


Figure 5.5 – **IP de filtrage Gaussien : impact des paramètres 2** - Utilisation d'un FPGA de type Virtex 6 pour différentes valeurs de σ et largeurs de noyaux de convolution (image de 640×480 pixels, pixels sur 17 bits et coefficients sur 24 bits)

On voit dans la figure 5.5 que le coefficient σ influe très peu sur l'utilisation des ressources, cependant la largeur du noyau de coefficients de convolution augmente linéairement la consommation pour tous les types de ressources présentés. Le choix de l'utilisateur pour la largeur du noyau de convolution dépendra directement du besoin de précision du filtrage Gaussien, ce paramètre ayant un impact fort sur la quantité de ressources nécessaires à l'IP. Le modèle logiciel des IP peut être utilisé pour générer des courbes d'erreur par rapport à l'utilisation de flottants, concernant les sorties d'une IP Gaussienne.

La figure 5.6 permet de voir l'évolution de l'estimation post-synthèse de la fréquence de fonctionnement de l'IP de filtrage Gaussien. La fréquence de fonctionnement estimée suit l'utilisation des blocs DSP48E1 : pour les solutions nécessitant plus de ces blocs, la fréquence de fonctionnement est très fortement réduite. Il est à noter qu'une fréquence de 100 MHz permet, par exemple, de traiter plus de 50 images de 1600×1200 pixels par seconde. Il n'est donc pour l'instant pas indispensable de choisir les paramètres des IP en fonction de la fréquence de fonctionnement.

5.3.3 Recherche de points d'intérêt

L'IP de recherche de points d'intérêt représente une forte consommation au sein du SoC. En effet, chaque pixel dans un rayon R autour d'un pixel de référence sera comparé à ce dernier, à chaque coup d'horloge. Ainsi, pour un rayon de recherche $R = 20$ pixels, on peut estimer un nombre de $\pi \times 20^2 = 1256$ comparateurs, dont la taille dépendra de la largeur de bus des pixels.

Les paramètres influents sur le coût de déploiement de cette IP sont la largeur des bus d'entrée et le rayon de recherche R. La figure 5.7 présente l'utilisation d'un Virtex 6 de Xilinx en registres, LUT et blocs de mémoire BRAM, respectivement. Aucun bloc DSP48E1 n'est utilisé par cette IP.

On remarque que l'utilisation des ressources évolue de façon régulière en fonction des paramètres. Pour les registres et les LUT, l'utilisation varie linéairement avec la largeur de bus des pixels, et avec le nombre de comparateurs déployés (donc une évolution quadratique par rapport au rayon de recherche R). La quantité de blocs mémoire BRAM évolue linéairement en fonction de R, mais par pallier en fonction de la largeur de bus des pixels : le nombre de BRAM reste stable en dessous de 19 bits et au dessus de 18 bits, mais le passage de 18 à 19 augmente fortement la consommation de blocs

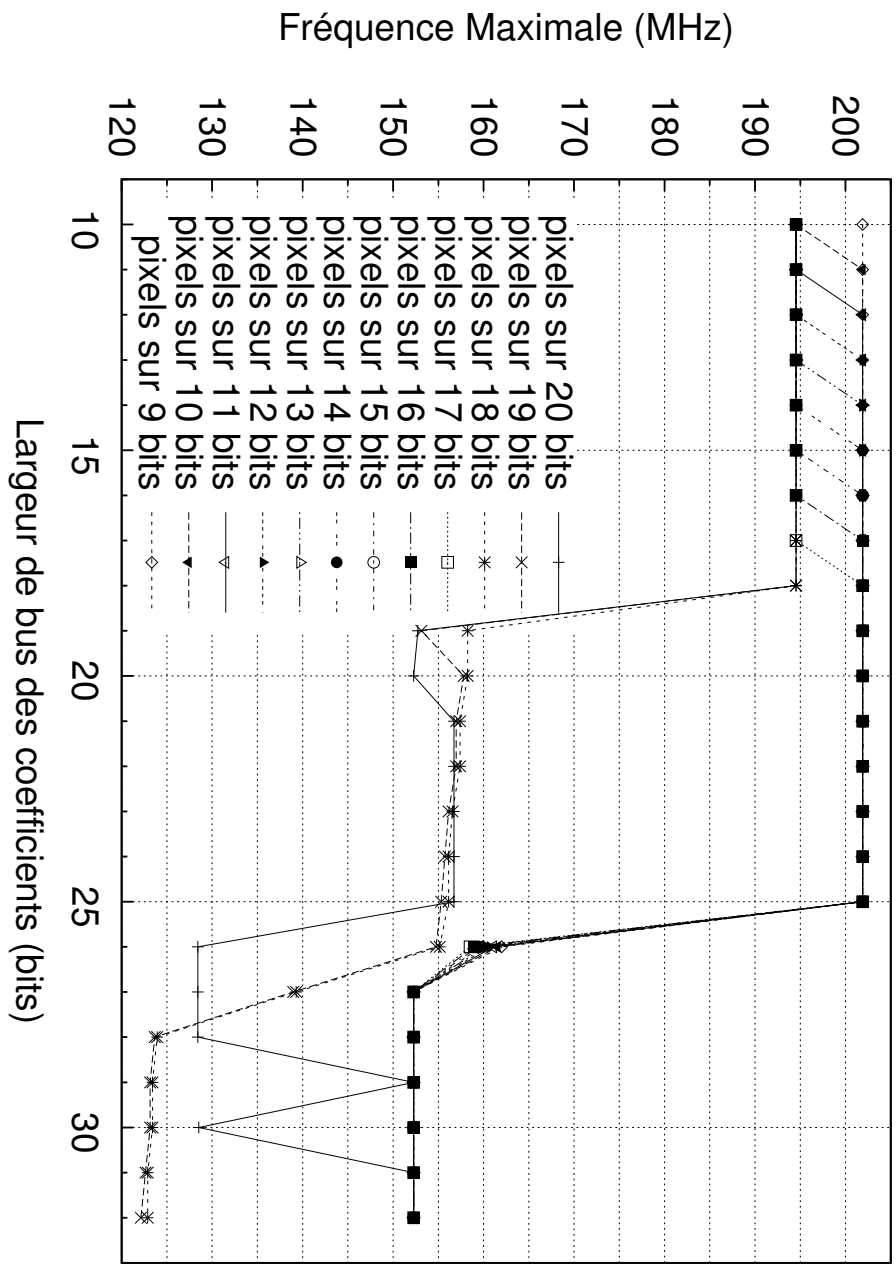


Figure 5.6 – **IP de filtrage Gaussien : impact des paramètres 3** - Fréquence maximale d'un FPGA de type Virtex 6 pour différentes largeurs de bus des pixels et des coefficients, pour une image de 640×480 pixels, $\sigma = \sqrt{2}$, et un noyau de convolution de largeur 7

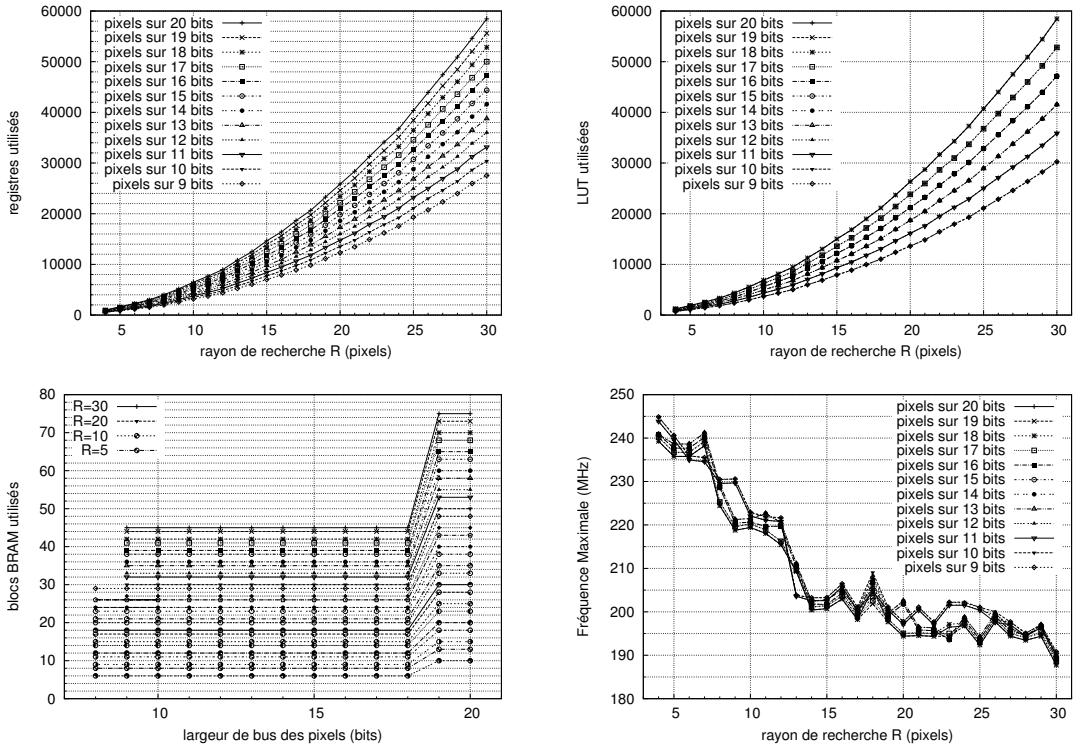


FIGURE 5.7 – **IP de recherche de points d'intérêt : impact des paramètres** - Utilisation d'un FPGA de type Virtex 6 et fréquence maximale de fonctionnement, pour différentes valeurs de R et de la largeur de bus des pixels (image de 640×480 pixels)

mémoire BRAM. Ce palier est dû à la technologie Virtex 6 : les blocs mémoire BRAM de cette famille de FPGA sont utilisables comme deux mémoires de 1k mots de 18 bits, ou de 512 mots de 36 bits. La largeur d'image étudiée ici étant de 640 pixels et le rayon R étant au maximum de 20 pixels, la profondeur des mémoires utilisées pour stocker les lignes de pixels sera au minimum de 600 mots, il faut donc utiliser des blocs BRAM de profondeur 1k, ainsi le passage d'un mot de 18 bits à 19 bits implique l'utilisation de BRAM supplémentaires. L'étude d'une image de plus haute résolution (au dessus de 1024 pixels de large) nécessitera plus de mémoire BRAM pour des pixels de plus de 9 bits, le stockage d'une ligne de pixels dépassant 1k pixels.

La fréquence maximale de fonctionnement en fin de synthèse semble évoluer assez peu à partir d'un rayon de recherche de 14 pixels. Un rayon de recherche moindre permet d'augmenter sensiblement cette fréquence, mais peut porter préjudice à l'efficacité du robot. La largeur de bus des pixels semble très peu influencer cette fréquence maximale.

Le point le plus important à garder à l'esprit en réglant les paramètres en fonction des besoins du robot est le palier d'utilisation des BRAM au delà de 18 bits par pixel. Il est préférable de rester à 18 bits maximum par pixels, sauf si l'utilisation de mémoires BRAM importe peu. Pour rappel, les IP de recherches de points d'intérêt sont les premiers consommateurs de blocs mémoire BRAM dans le SoC. Le reste de l'utilisation FPGA varie de façon stable en fonction des paramètres, il faut alors trouver un compromis entre la précision des résultats fournis par l'IP et les ressources qui lui sont nécessaires.

5.3.4 Extraction de voisinages de points d'intérêt

L'effet des paramètres sur les IP existantes a été étudié, ces effets sont aussi évalués pour les traitements logiciels. La transformation log-polaire des voisinages des points d'intérêts, située en fin de chaîne des traitements, est actuellement conçue pour s'exécuter en logiciel. Le déploiement final de ces traitements est prévu sous forme d'IP à terme, ce qui fait que ces IP représentent un coût inconnu pour l'instant. Si par conséquent, l'impact de ces paramètres sur l'utilisation totale du FPGA n'est pas encore connu, leurs valeurs influent fortement sur l'efficacité du réseau de neurones. Une imagette log-polaire de plus petite taille sera lue plus rapidement, mais représentera une

quantité moindre d'information. L'utilisateur final sera chargé de trouver le juste milieu entre la rapidité du réseau de neurones et la pertinence des imagettes log-polaires. En figure 5.8, on voit l'impact sur les répartitions des pixels de l'imagette log-polaire sur les pixels de l'image cartésienne : chaque zone grise continue dans l'anneau représente un pixel de l'imagette log-polaire.

La valeur de θ_{max} proposée au début de ce projet, 36 (un choix permettant de représenter chaque incrément de θ comme un angle de 10°), a été écarté en faveur de multiples de 8. Les multiples de 8 permettent d'obtenir une cartographie log-polaire régulière sur l'image cartésienne. Une meilleure robustesse à la rotation des imagettes log-polaires découle naturellement de ce choix. On peut voir que dans l'image de gauche, certains pixels de l'imagette log-polaire seront redondants (vers le centre de l'anneau), tandis que dans l'image de droite, chaque pixel log-polaire sera affecté à son propre lot de pixels. Le désavantage de la solution de droite est une plus faible résolution de l'imagette log-polaire, la redondance de l'image de droite étant compensée par une plus grande précision générale.

L'utilisateur final décidera de fixer ces valeurs ρ_{max} et θ_{max} en fonction des besoins du réseau de neurones. Ces paramètres pourront être réglés expérimentalement, à travers une batterie de tests intensifs, afin de viser un fonctionnement optimal du robot.

5.4 Choix de la plate-forme FPGA

Le choix de la carte FPGA se fait selon plusieurs critères. Tout d'abord, la quantité de ressources logiques nécessaire à l'élaboration des IP restreint le choix des FPGA que l'on pourra utiliser. Les ressources logicielles sont particulièrement importantes aussi, si l'on ne peut pas se permettre d'exécuter les traitements logiciels sur le PC embarqué du robot. Enfin, une carte sera utilisée temporairement au cours du développement du SoC pour la disponibilité de ses accessoires, tels qu'une caméra sur une carte fille que l'on pourra connecter rapidement aux IP afin de valider le fonctionnement sur puce.

Dans la pratique, on préférera pour la plate-forme définitive un circuit de type Zynq, qui dispose en plus d'une zone FPGA, d'un processeur Cortex A9 (double cœur, 800MHz). Un processeur de ce type permet d'exécuter, dans le respect des contraintes temps-réel, les traitements logiciels de l'application de vision (extraction des voisinages

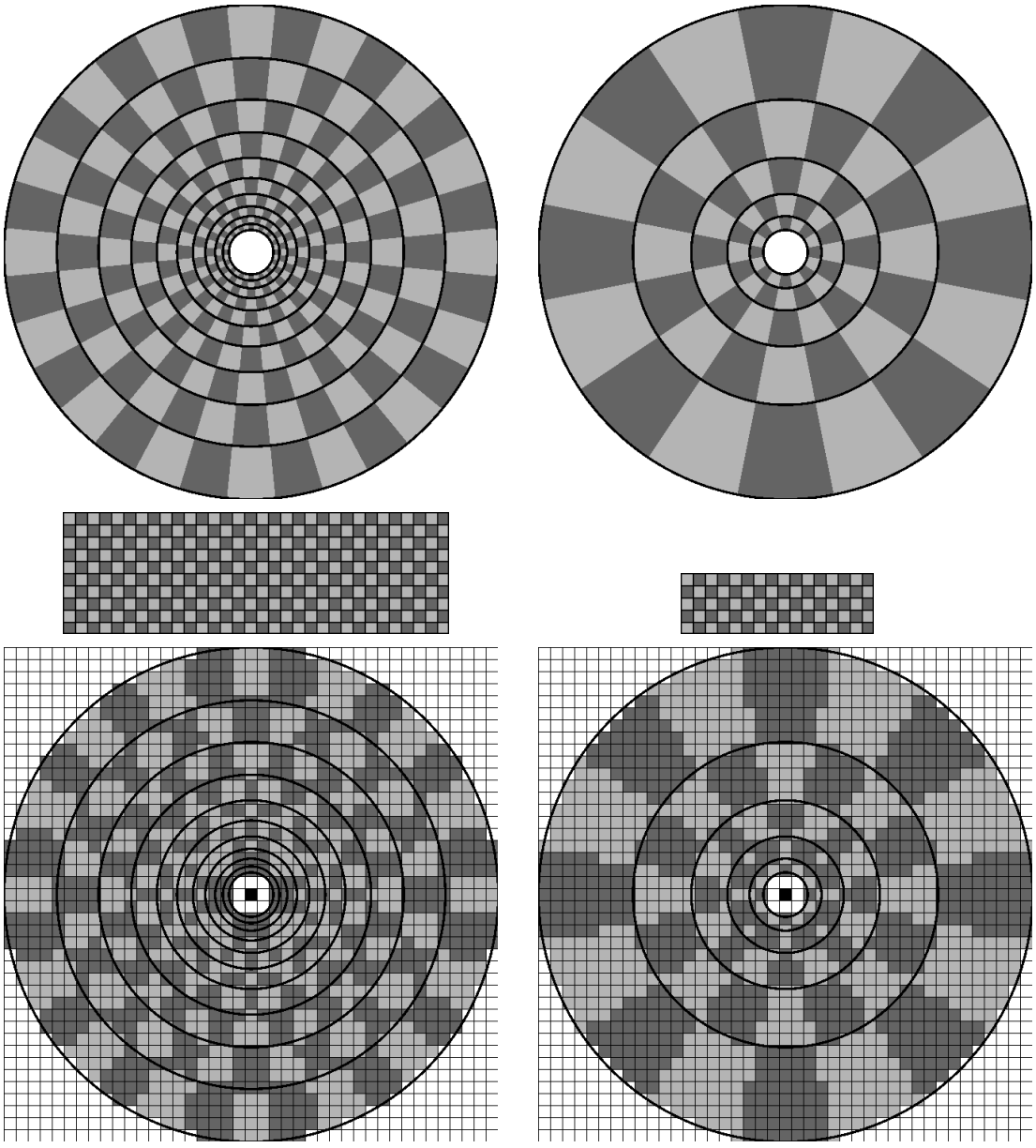


Figure 5.8 – **Paramètres de transformation log-polaire** - En haut, l'empreinte d'une transformation log-polaire d'un anneau de rayons 1,8 pixels (intérieur) et 20 pixels (extérieur). Au centre, les imagettes log-polaires, et en bas, les pixels de l'anneau d'entrée associés à chaque pixel log-polaire. Paramètre commun : R=20, paramètres $[\rho_{max}, \theta_{max}]$: à gauche, $[10, 32]$, à droite, $[5, 16]$

des points d'intérêt, dont mise en forme log-polaire et normalisation). Des cartes de développement pour ce circuit sont disponibles à l'heure de la publication de ce document.

En attendant que les cartes de développement Zynq soient disponibles, les Virtex 6 de Xilinx peuvent être utilisés, en utilisant un PC pour effectuer les traitements logiciels. Pour les tests préliminaires des IP sur carte de développement, une carte ALTERA permet d'utiliser un module caméra enfichable directement sur la carte. Cette carte disposant d'un FPGA trop petit pour l'architecture matérielle proposée, elle sera utilisée pour la validation d'IP individuelle, mais pas l'hébergement de la totalité du SoC.

Les ressources utilisées par le FPGA sont un point essentiel du choix du FPGA. Les IP présentées précédemment sont synthétisées pour un FPGA Kintex 7, cette technologie étant utilisée pour la partie FPGA des circuits Zynq de Xilinx (les circuits Zynq n'étant pas proposés à la synthèse dans la version utilisée de Xilinx ISE). On synthétisera les IP sur le FPGA le plus proche du plus gros Zynq (Z-7045) : le Kintex 7 XC7K355T. Le tableau 5.2 présente les ressources disponibles sur les quatre circuits Zynq.

Circuit	Cellules logiques	BRAM (kb/blocs 36kb)	DPS48E1
Kintex7 XC7K355T	356160	25740/715	1440
Zynq Z-7045	350000	19620/545	900
Zynq Z-7030	125000	9540/265	400
Zynq Z-7020	85000	5040/140	220
Zynq Z-7010	28000	2160/60	80

TABLE 5.2 – **Comparaison des ressources disponibles dans les FPGA Zynq et Kintex 7** - On utilisera le Kintex 7 XC7K355T pour synthétiser les IP, la synthèse sur Zynq n'étant pas disponible au moment des mesures

Les IP sont synthétisées sur un circuit XC7K355T afin d'évaluer l'utilisation typique de la matrice FPGA des circuits Zynq. Les résultats de la synthèse de l'ensemble de la chaîne d'IP pour le XC7K355T sont présentés dans le tableau 5.3.

L'utilisation du Kintex 7 est très proche des résultats de synthèses obtenus sur un Virtex 6. Cette utilisation de ressources du FPGA appelle un circuit Zynq de type

FPGA utilisé	Registres	LUT	BRAM 36k	DPS48E1
Virtex 6	69018	74402	119	91
Kintex 7	67046	74235	119	91

Table 5.3 – **Résultats de synthèse sur deux types de FPGA** - les circuits utilisés pour la synthèse sont le Virtex 6 XC6VLX550T et le Kintex 7 XC7K355T. Pixels sur 17 bits, coefficients de convolution sur 24 bits, R=[20,10,5], image de 640 × 480 pixels

Z-7020, Z-7030 ou Z-7045, le circuit Z-7010 étant sous-dimensionné pour toutes les ressources présentées.

La fréquence maximale estimée en fin de synthèse, pour la partie à base d'IP, se situe aux environs de 180MHz, pour ce même jeu de paramètres. Ainsi, le flux d'images de 640 × 480 pixels en niveaux de gris peut être étudié à une cadence de plus de 500 images par seconde, les limites se situeront donc en pratique non pas dans les IP, mais dans la caméra, son interface avec les IP, et l'interface des IP avec le réseau de neurones. La résolution de l'image d'entrée est elle aussi à surveiller, une résolution full-HD (1920 × 1080 pixels) en niveaux de gris réduisant la cadence à 80 images par seconde. De plus, cette fréquence n'est qu'une estimation en fin de synthèse, il reste à obtenir la fréquence de fonctionnement maximale après placement et routage de ces IP, qui peut alors être plus faible. Ces pronostics ne sont valables que pour les IP de traitement, les communications avec la caméra et avec le processeur du SoC n'étant pas représentées dans les résultats de synthèse.

Conclusion

Le but de cette thèse est l'intégration du système de vision bio-inspiré multirésolution sur des plate-formes robotiques autonomes. L'accélération des calculs, ainsi que la réduction du volume et de la consommation électrique des unités de calcul sont les problèmes principaux de ces travaux.

Plusieurs solutions logicielles ont été étudiées, afin d'identifier les traitements nécessitant une forte accélération. Un découpage logiciel/matériel est proposé pour atteindre au plus vite un circuit fonctionnel intégrable aux robots.

La technologie FPGA a été choisie pour son respect des contraintes de volume et d'énergie électrique, ainsi que pour sa flexibilité permettant le réglage du système en fonction des applications robotiques ciblées. Les unités de traitement dédiées permettent l'étude d'un flux d'images en haute définition (1920×1080 pixels) en niveaux de gris à une cadence de plus de 50 images par seconde, ce qui dépasse les attentes. La flexibilité du circuit permet d'adapter ces unités de traitement dédiées en fonction des besoins de chaque application robotique. Les paramètres qui varient au cours du fonctionnement du système sont réglables dynamiquement. L'impact des autres paramètres sur l'utilisation du FPGA a été présenté sous forme graphique, afin de guider les choix des utilisateurs du système.

Plusieurs perspectives se présentent dans la continuité de ce travail. En premier lieu, il est essentiel de déployer le système de vision complet sur un circuit : un prototype est en cours de conception. On visera l'intégration de six demi-échelles dans le circuit, sur un flux d'images à haute résolution et forte cadence. Deux aspects plus concrets du circuit sont aussi à résoudre avec les utilisateurs du circuit : l'alimentation électrique

(charge supplémentaire sur la batterie principale du robot, ou batterie dédiée), ainsi que la caméra (choix du module, connecteur flexible pour la mobilité de la plate-forme pan/tilt). Une fois que toutes ces étapes sont atteintes, les tests du robot en symbiose avec le circuit pourront être effectués, ce qui permettra d'envisager des améliorations au système de vision.

Une unité de traitement dédiée est en cours de conception afin d'exécuter les transformations log-polaires directement sur les flux de pixels. Cette IP devrait permettre de rendre indépendantes les différentes caractéristiques locales extraites par le système. La suite plus globale du projet, est le déploiement sur FPGA des réseaux de neurones responsables des premières étapes cognitives de la perception visuelle (projet ANR SATURN). Ainsi, il sera particulièrement important de rendre les imagettes log-polaires indépendantes les unes des autres : chacune ayant sa mémoire dédiée, l'étude de plusieurs imagettes par le réseau de neurones pourra ainsi se faire en parallèle. Au delà de ces aspects, l'IP de transformation log-polaire devrait permettre l'étude d'images à une cadence plus élevée, ainsi que la génération de caractéristiques plus nombreuses et de plus haute résolution.

La dernière piste est quant à elle plus générale. L'intégration aux robots du système de vision n'est pas suffisante à rendre ces robots autonomes vis à vis d'unités de calcul externes. En effet, le réseau de neurones artificiel central à chaque robot est lui-même déporté sur des stations de travail reliées au robot via une liaison WiFi. L'intégration de ces réseaux de neurones sous un format embarqué est un défi majeur qui est en cours de discussions. Ce projet est intimement lié à celui du système de vision, et il est essentiel de gérer une certaine symbiose entre ces deux éléments. On peut envisager la conception d'un circuit complet contenant capteur, système de vision et réseau de neurones au sein d'un même circuit intégré. Ce circuit, au delà d'une rétine artificielle, s'inspirerait du cerveau des mammifères, avec en entrée les données des autres capteurs du robot, et en sortie les commandes de ses actionneurs. Les technologies 3D semblent être une piste intéressante pour la réalisation de ce circuit, le capteur vidéo étant disposé sur la couche supérieure, le système de vision utilisant une deuxième couche, le réseau de neurones pouvant être déployé dans d'autres couches. Une meilleure modélisation des mécanismes parallèles du cerveau biologique sera obtenue grâce à la répartition sur

la surface du circuit des interconnexions entre chaque imagette log-polaire et différentes zones du réseau de neurones.

Références

[ACJG+09] L. Alba, R. Domínguez Castro, F. Jiménez-Garrido, S. Espejo, S. Morillas, J. Listán, C.Utrera, A.García, Ma.D. Pardo, R. Romay, C. Mendoza, A. Jiménez et Á. Rodríguez-Vázquez : *Spatial Temporal Patterns for Action-Oriented Perception in Roving Robots*, chapitre 8. New Visual Sensors and Processors, pages 351 – 369. Springer, 2009. 38

[AKD03] Nicolas Abel, Lounis Kessal et Didier Demigny : Mise en oeuvre du lisseur de deriche sur l'architecture reconfigurable dynamiquement ardoise. *In 19è Colloque sur le traitement du signal et des images, FRA, 2003.* GRETSI, Groupe d'Etudes du Traitement du Signal et des Images, 2003. 39

[ALA05] J. D. Anderson, D. J. Lee et J. K. Archibald : Fpga implementation of vision algorithms for small autonomous robots. *In SPIE Optics East, Robotics Technologies and Architectures, Intelligent Robots and Computer Vision XVIII*, pages 23–26, 2005. 35

[Alt11] Altera : Strategic considerations for emerging soc fpgas. White paper, Altera, Febuary 2011. 44

[Bel09] Ahmed Nabil Belbachir, éditeur. *Smart Cameras*. Springer, 2009. 37

[BETG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars et Luc Van Gool : Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346 – 359, 2008. Similarity Matching in Computer Vision and Multimedia. 41

[BL98] Marc Bolduc et Martin D. Levine : A review of biologically motivated space-variant data reduction models for robotic vision. *Comput. Vis. Image Underst.*, 69:170–184, February 1998. 36

[BMC08] V. Bonato, E. Marques et G. A. Constantinides : A parallel hardware architecture for scale and rotation invariant feature detection. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(12):1703–1712, 2008. 34, 85

[Boa02] Kwabena Boahen : A retinomorphic chip with parallel pathways : Encoding increasing, on, decreasing, and off visual signals. *Analog Integrated Circuits and Signal Processing*, 30:121–135, 2002. 35

[BPS98] Arrigo Benedetti, Andrea Prati et Nello Scarabottolo : Image convolution on fpgas : The implementation of a multi-fpga fifo structure. *EUROMICRO Conference*, 1:123–130, 1998. 83

[BSP07] J. Bauer, N. Sunderhauf et P. Protzel : Comparing several implementations of two recently published feature detectors. *In Proceedings of the International Conference on Intelligent and Autonomous Systems, IAV, Toulouse, France*, 2007. 41

[BSV96] A. Bernardino et J. Santos-Victor : Vergence control for robotic heads using log-polar images. *In Intelligent Robots and Systems*, 1996. 21

[Can86] J Canny : A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8:679–698, November 1986. 13

[CB07] Pierre Chalimbaud et François Berry : Embedded active vision system based on an fpga architecture. *EURASIP Journal on Embedded Systems*, 2007:26–26, January 2007. 39

[CL00] Wanming Chu et Yamin Li : Cost/performance tradeoff of n-select square root implementations. *In Proceedings of the 5th Australasian Computer Architecture Conference*, pages 9–, Washington, DC, USA, 2000. IEEE Computer Society. 81

[CNB+08] Jeff Chase, Brent Nelson, John Bodily, Zhaoyi Wei et Dah-Jye Lee : Real-time optical flow calculations on fpga and gpu architectures : A comparison study. *In Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 173–182. IEEE Computer Society, 2008. 49

[Cop06] Ben Cope : Implementation of 2D convolution on FPGA, GPU and CPU. Rapport technique, Department of Electrical and Electronic Engineering, Imperial College, 2006. 84

[CRP02] J. Crowley, O. Riff et J. Piater : Fast computation of characteristic scale using a half-octave pyramid. *In Proceedings of the Cognitive Vision Workshop (Cogvis)*, Zurich, October 2002. 14, 41, 52

[CTC+09] V. Chandrasekhar, G. Takacs, D. Chen, S. Tsai, R. Grzeszczuk et B. Girod : Chog : Compressed histogram of gradients a low bit-rate feature descriptor. *In Computer Vision and Pattern Recognition*, 2009. 42

[Dye87] C. R. Dyer : *Multiscale image understanding*, pages 171–213. Academic Press Professional, Inc., 1987. 14

[EBD+04] A. Elouardi, S. Bouaziz, A. Dupret, J.O. Klein et R. Reynaud : On chip vision system architecture using a cmos retina. *In IEEE Intelligent Vehicle Symposium, IV'04*, pages 206 – 211, June 2004. 35

[FFC10] Francisco Fons, Mariano Fons et Enrique Cantó : Run-time self-reconfigurable 2d convolver for adaptive image processing. *Microelectronics Journal*, 42:204–217, 2010. 84

[Fia12] Laurent Fiack : Intégration d'un système de vision robotique d'inspiration biologique sur fpga. Mémoire de projet de Master 2, 2012. 111

[FKA06] J. Fauqueur, N. Kingsbury et R. Anderson : Multiscale keypoint detection using the dual-tree complex wavelet transform. *In IEEE International Conference on Image Processing*, Oct. 2006. 14, 42

[Fre04] Matthew FRENCH : A power efficient image convolution engine for field programmable gate arrays. *In 2004 MAPLD International Conference*, 2004. 91

[Gai] Aeroflex GAISLER : Leon processor, leon3 core. 28

[GDHP10] Dominique GINHAC, Jérôme DUBOIS, Barthélémy HEYRMAN et Michel PAINDAVOINE : A high speed programmable focal-plane simd vision chip. *Analog Integr. Circuits Signal Process.*, 65:389–398, December 2010. 39

[GL11] Reza GHODSSI et Pinyen LIN : *MEMS Materials and Processes Handbook*. Springer, 2011. 45

[GRRD07] Sara GRANADOS, Eduardo ROS, Rafael RODRÍGUEZ et Javier DÍAZ : Visual processing platform based on artificial retinas. *In Proceedings of the 9th international work conference on Artificial neural networks*, IWANN'07, pages 506–513, Berlin, Heidelberg, 2007. Springer-Verlag. 35

[GSB04] K. GHOSH, S. SARKAR et K. BHAUMIK : A bio-inspired model for multi-scale representation of even order gaussian derivatives. *In Intelligent Sensors, Sensor Networks and Information Processing Conference*, 2004. 14

[HMV07] Emmanuel HUCK, Benoît MIRAMOND et François VERDIER : A Modular SystemC RTOS Model for Embedded Services Exploration. *In First European Workshop on Design and Architectures for Signal and Image Processing (DASIP)*, Grenoble, France, November 2007. 27, 69

[HS88] C. HARRIS et M. STEPHENS : A combined corner and edge detection. *In Proceedings of The Fourth Alvey Vision Conference*, pages 147–151, 1988. 41

[IK01] M. ISHIKAWA et T. KOMURO : Digital vision chips and high-speed vision systems. *In VLSI Circuits, 2001*, pages 1 – 4, 2001. 38

[Int] INTEL : Intel atom processor e6x5c series. 45

[KD09] Nachiket KAPRE et André DEHON : Performance comparison of single-precision spice model-evaluation on fpga, gpu, cell, and multi-core processors. *In FPL : 2009 International Conference on Field Programmable Logic and Applications. International Conference on Field Programmable Logic and Applications*, 2009. 48

[Kel71] M.D. KELLY : Edge detection by computer in pictures using planning. *In Machine Intelligence 6*, pages 397 – 409, 1971. 41

[Kha10] Sonia KHATCHADOURIAN : *Mise en œuvre d'une architecture de reconnaissance de formes pour la détection de particules à partir d'images atmosphériques.* Thèse de doctorat, Université de Cergy-Pontoise, Sept. 2010. Direction : L. Kessal, J-C. Prévotet. 37

[KII97] Takashi KOMURO, Idaku ISHII et Masatoshi ISHIKAWA : Vision chip architecture using general-purpose processing elements for 1ms vision system. *In Proceedings of the 1997 Computer Architectures for Machine Perception (CAMP '97)*, CAMP '97, pages 276–. IEEE Computer Society, 1997. 38

[KNL+00] Hiroyuki KURINO, M. NAKAGAWA, Kang Wook LEE, Tomonori NAKAMURA, Yuusuke YAMADA, Ki Tae PARK et Mitsumasa KOYANAGI : Smart vision chip fabricated using three dimensional integration technology. *In NIPS*, pages 720–726, 2000. 39

[Koz06] L.J. KOZLOWSKI : Noise minimization via deep submicron system-on-chip integration in megapixel cmos imaging sensors. *Opto-Electronics Review*, 14:11–18, March 2006. 38

[LMY04] Miriam LEESER, Shawn MILLER et Haiqian YU : Smart camera based on reconfigurable hardware enables diverse real-time applications. *In Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 147–155. IEEE Computer Society, 2004. 37

[Low99] David G. Lowe : Object recognition from local scale-invariant features. *In Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–. IEEE Computer Society, 1999. 17, 40

[Low04] David G. Lowe : Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60:91–110, November 2004. 34, 40

[LZ07] Bing Luo et Yun Zhang : Hierarchical classification for imbalanced multiple classes in machine vision inspection. *In Proceedings of the Fourth International Conference on Image and Graphics*, ICIG '07, pages 536–541, Washington, DC, USA, 2007. IEEE Computer Society. 33

[MA06] J. Martinez et L. Altamirano : Fpga-based pipeline architecture to transform cartesian images into foveal images by using a new foveation approach. *Reconfigurable Computing and FPGA's*, 0:1–10, 2006. 43

[Mai07] Mickaël Maillard : *Formalisation de la perception comme dynamique sensori-motrice : application dans un cadre de reconnaissance d'objets par un robot autonome.* Thèse de doctorat, Université de Cergy-Pontoise, déc. 2007. Direction : Ph. Gaussier, L. Hafemeister. 6, 14, 21

[MB08] Giovanni De Micheli et Luca Benini : *On-Chip Communication Architectures : System on Chip Interconnect.* Morgan Kaufmann Publishers Inc., 2008. 45

[MDBS09] Abid Muneeb, Fabio Dias, François Berry et Jocelyn Sérot : Harnessing a multi-sensor fpga-based smart camera : a virtual processor-based approach. *In DASIP 2009*, pages 135 – 141, Sep. 2009. 38

[MGGH05] M. Maillard, O. Gapenne, Ph. Gaussier et L. Hafemeister : Perception as a dynamical sensori-motor attraction basin. *In* M. Capcarrere et al., éditeur : *Advances in Artificial Life (8th European Conference, ECAL)*, volume LNAI 3630 de *Lecture Note in Artificial Intelligence*, pages 37–46. Springer, sep 2005. 8

[Mor80] Hans Peter MORAVEC : *Obstacle avoidance and navigation in the real world by a seeing robot rover*. Thèse de doctorat, Stanford University, 1980. 40

[MRP+05] Antonio MARTÍNEZ, Leonardo M. REYNERI, Francisco J. PELAYO, Samuel F. ROMERO, Christian A. MORILLAS et Begoña PINO : Automatic generation of bio-inspired retina-like processing hardware. *In international work-conference on artificial neural networks*, volume 3512, pages 527 – 533. Springer, 2005. 36

[MS04] Krystian MIKOLAJCZYK et Cordelia SCHMID : Scale & affine invariant interest point detectors. *Int. J. Comput. Vision*, 60:63–86, Oct. 2004. 42

[MS09] Grant MARTIN et Gary SMITH : High-level synthesis : Past, present, and future. *IEEE Design and Test of Computers*, 26:18–25, July 2009. 114

[Nel00] Anthony Edward NELSON : Implementation of image processing algorithms on fpga hardware. Mémoire de Master, Vanderbilt University, May 2000. 83, 86

[Oue03] Nabil OUERHANI : *Visual Attention : From Bio-Inspired Modeling to Real-Time Implementation*. Thèse de doctorat, Institut de microtechnique de l'université de Neufchâtel, 2003. 14

[Per03] Stefania PERRI : A power efficient image convolution engine for field programmable gate arrays. *In 2003 MAPLD International Conference*, 2003. 83

[RBSE10] Nicolas ROUDEL, Francois BERRY, Jocelyn SEROT et Laurent ECK : A new high-level methodology for programming fpga-based smart camera. *In Proceedings of the 2010 13th Euromicro Conference on Digital System Design : Architectures, Methods and Tools*, DSD '10, pages 573–578, Washington, DC, USA, 2010. IEEE Computer Society. 40

[Rev97] Arnaud REVEL : *Contrôle d'un robot autonome par approche neuromimétique*. Thèse de doctorat, Université de Cergy-Pontoise, 1997. 5

[Rob] ROBOSOFT : Robulab 10 robots. 11

[RVDCF+07] A. Rodríguez-Vázquez, R. Domínguez-Castro, Jiménez-Garrido F., S. Morillas, J. Listán, L. Alba et et al : The eye-ris cmos vision system. *Analog Circuit Design*, 2007:15 – 32, 2007. 38

[RVLCC+04] A. Rodriguez-Vazquez, G. Linan-Cembrano, L. Carranza, E. Roca-Moreno, R. Carmona-Galan, F. Jimenez-Garrido, R. Dominguez-Castro et S.E. Meana : Ace16k : the third generation of mixed-signal simd-cnn ace chips toward vsocs. *Circuits and Systems*, 51:851 – 863, 2004. 38

[RWS+08] B Rinner, T. Winkler, W. Schriebl, M. Quaritsch et W. Wolf : The evolution from single to pervasive smart cameras. *In Distributed Smart Cameras. ICDSC 2008.*, pages 1–10, Sep. 2008. 40

[Sch77] E. L. Schwartz : Spatial mapping in the primate sensory projection : Analytic structure and relevance to perception. *Biological Cybernetics*, 25:181–194, 1977. 21

[SGCZ03] Shiguang Shan, Wen Gao, Bo Cao et Debin Zhao : Illumination normalization for robust face recognition against varying lighting conditions. *Analysis and Modeling of Faces and Gestures, IEEE International Workshop on*, 0:157, 2003. 21

[SM97] Cordelia Schmid et Roger Mohr : Local grayvalue invariants for image retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19:530–535, May 1997. 17

[SVK08] Sam Schauland, Joerg Velten et Anton Kummert : Implementation of hardware-optimized 3-d wave digital filters for motion-based object detection in video scenes. *In 16th European Signal Processing Conference (EUSIPCO 2008)*, August 2008. 35

[TB10] V. Javier Traver et Alexandre Bernardino : A review of log-polar imaging for visual perception in robotics. *Robotics and Autonomous Systems*, 58(4):378 – 398, 2010. 42, 43

[TP03] V. Traver et Filiberto Pla : Designing the lattice for log-polar images. *In* Ingela Nyström, Gabriella Sanniti di Baja et Stina Svensson, éditeurs : *Discrete Geometry for Computer Imagery*, volume 2886 de *Lecture Notes in Computer Science*, pages 164–173. Springer Berlin / Heidelberg, 2003. 21

[TP08] V. Javier Traver et Filiberto Pla : Log-polar mapping template design : From task-level requirements to geometry parameters. *Image Vision Comput.*, 26:1354–1370, October 2008. 42

[TS93] M. Tistarelli et G. Sandini : On the advantages of polar and log-polar mapping for direct estimation of time-to-impact from optical flow. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15:401–410, April 1993. 21

[WCLT10] Wai Kit Wong, Chee Wee Choo, Chu Kiong Loo et Joo Peng Teh : Fpga implementation of log-polar mapping. *Int. J. Comput. Appl. Technol.*, 39:12–18, August 2010. 42

[WLNM07] Zhaoyi Wei, Dah-Jye Lee, Brent Nelson et Michael Martineau : A fast and accurate tensor-based optical flow algorithm implemented in fpga. *In Proceedings of the Eighth IEEE Workshop on Applications of Computer Vision*, WACV '07, page 18. IEEE Computer Society, 2007. 49

[WLY08] Xing Wu, Peihuang Lou et Jun Yu : An embedded vehicular controller with arm and dsp for a vision-based agv. *In The 2008 International Conference on Embedded Software and Systems Symposia (ICESS2008)*, pages 398–403, 2008. 33

[Xil09] Xilinx : Ml605 faq, June 2009. 11

Annexes

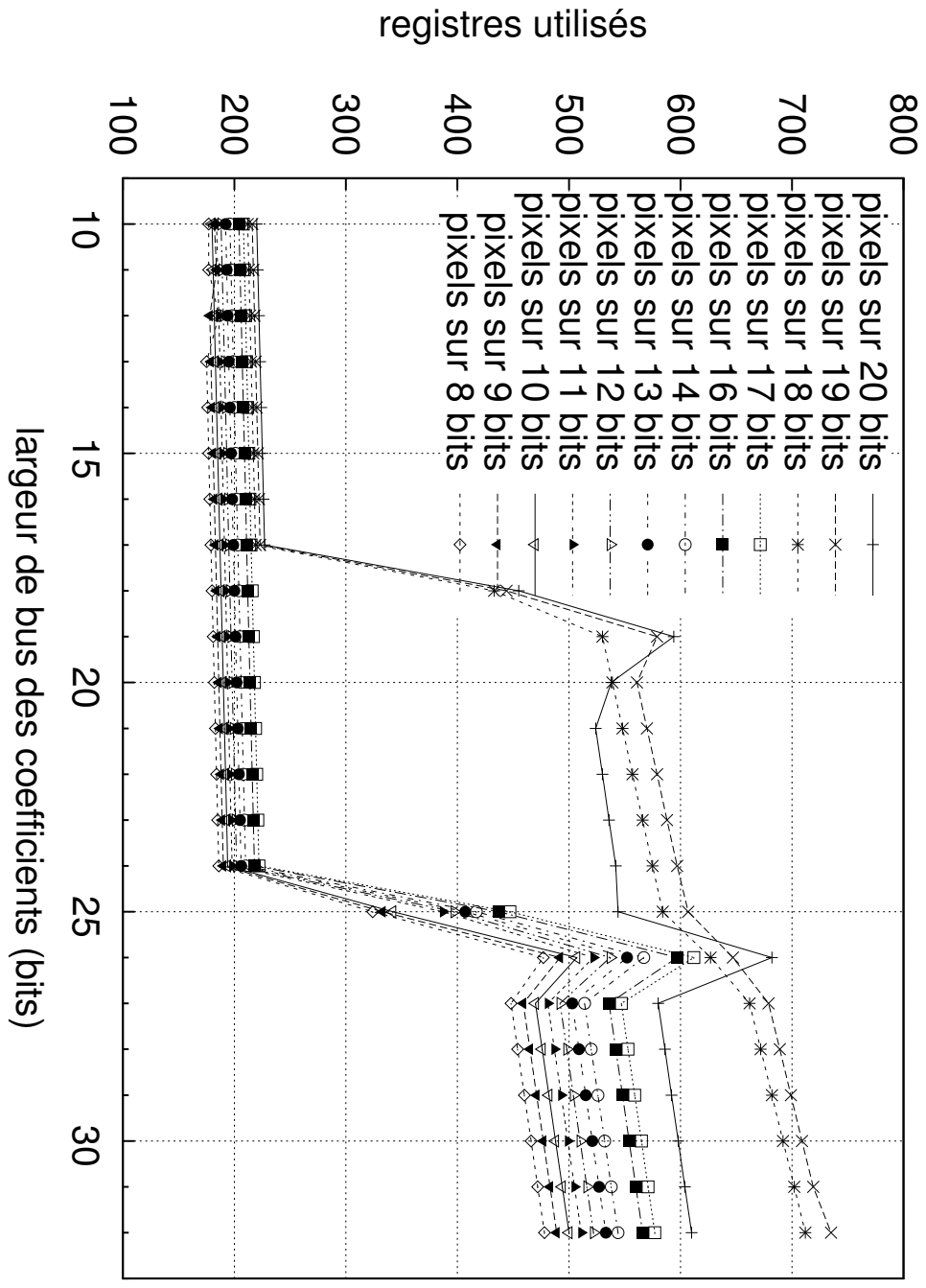
Résultats de synthèse supplémentaires

Paramétrage de l'IP de filtrage Gaussien

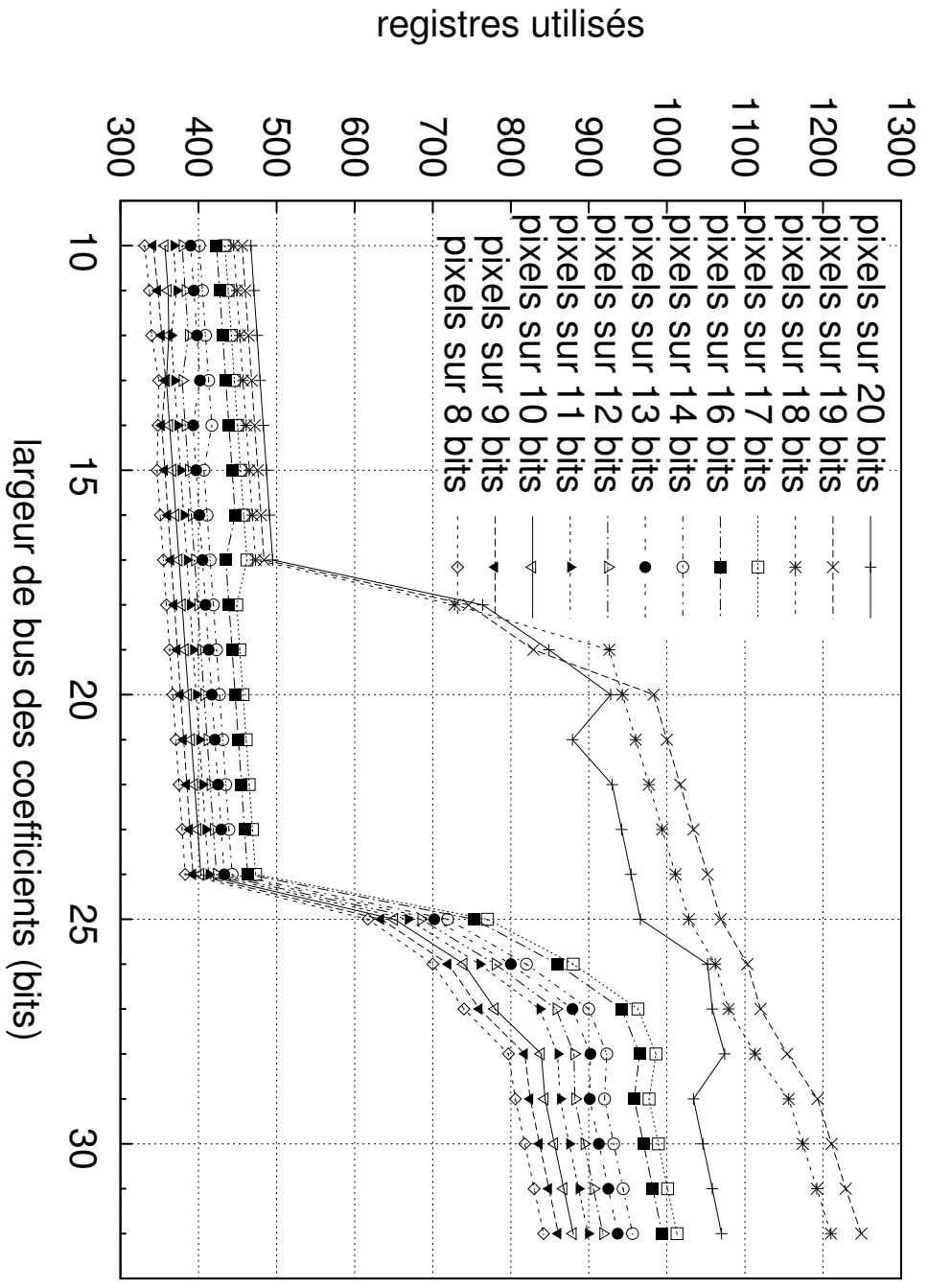
L'IP de filtrage Gaussien se paramètre selon plusieurs valeurs, qui influencent l'utilisation du FPGA :

- le coefficient σ, valant 1 ou $\sqrt{2}$ dans le SoC
- la largeur du bus de données des pixels d'entrée et de sortie
- la largeur du bus de données des résultats intermédiaires de MACC, donc des coefficients
- les dimensions du noyau de coefficients Gaussiens

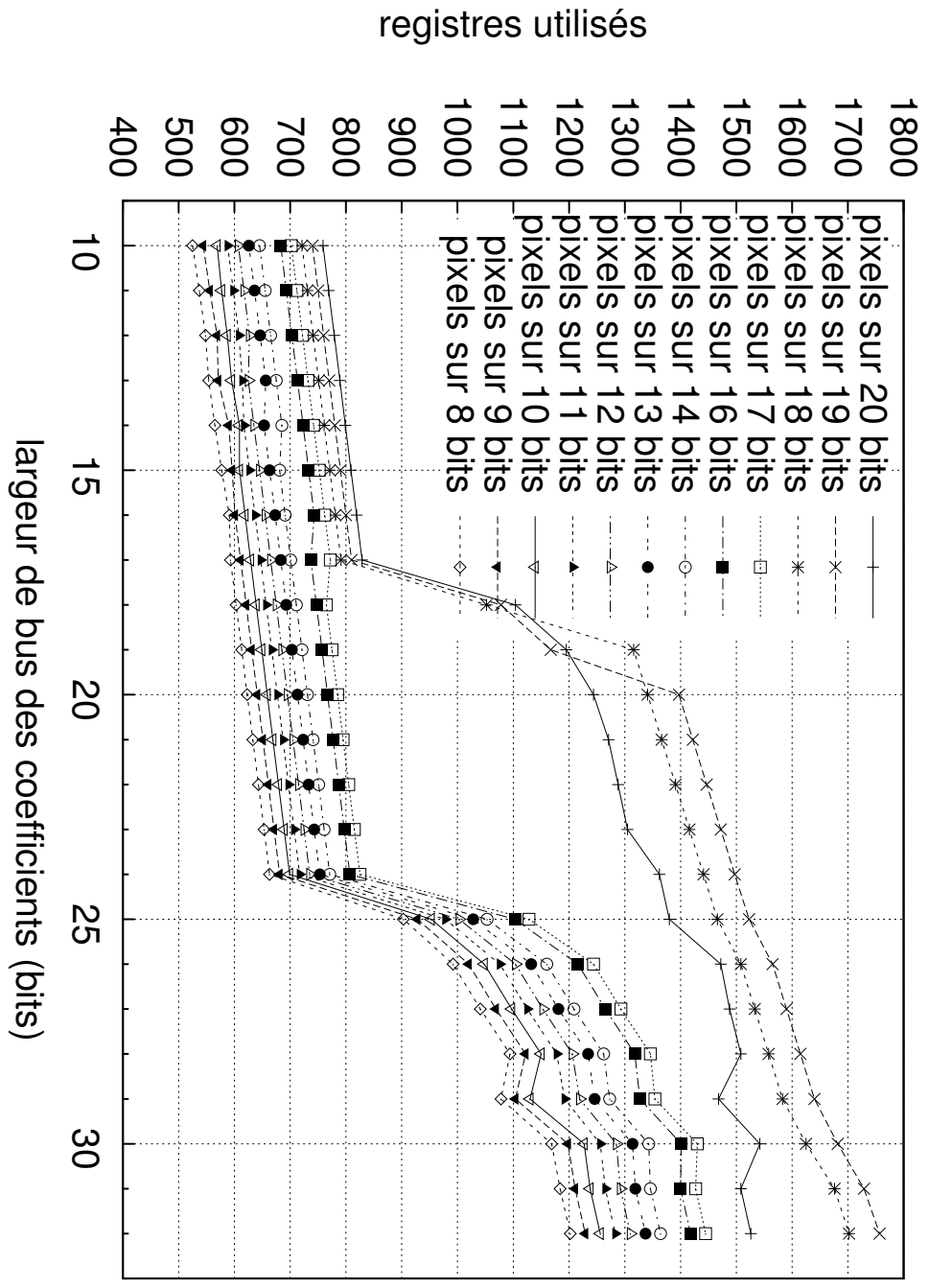
Les figures suivantes permettent de mieux appréhender l'utilisation des ressources FPGA selon l'évolution de ces paramètres. L'utilisation des ressources du FPGA (on considère surtout les registres, les LUT, les blocs mémoire BRAM et les blocs de calcul DSP48E1) évolue selon tous ces paramètres, il est donc difficile de proposer une seule figure pour chacune de ces ressources.



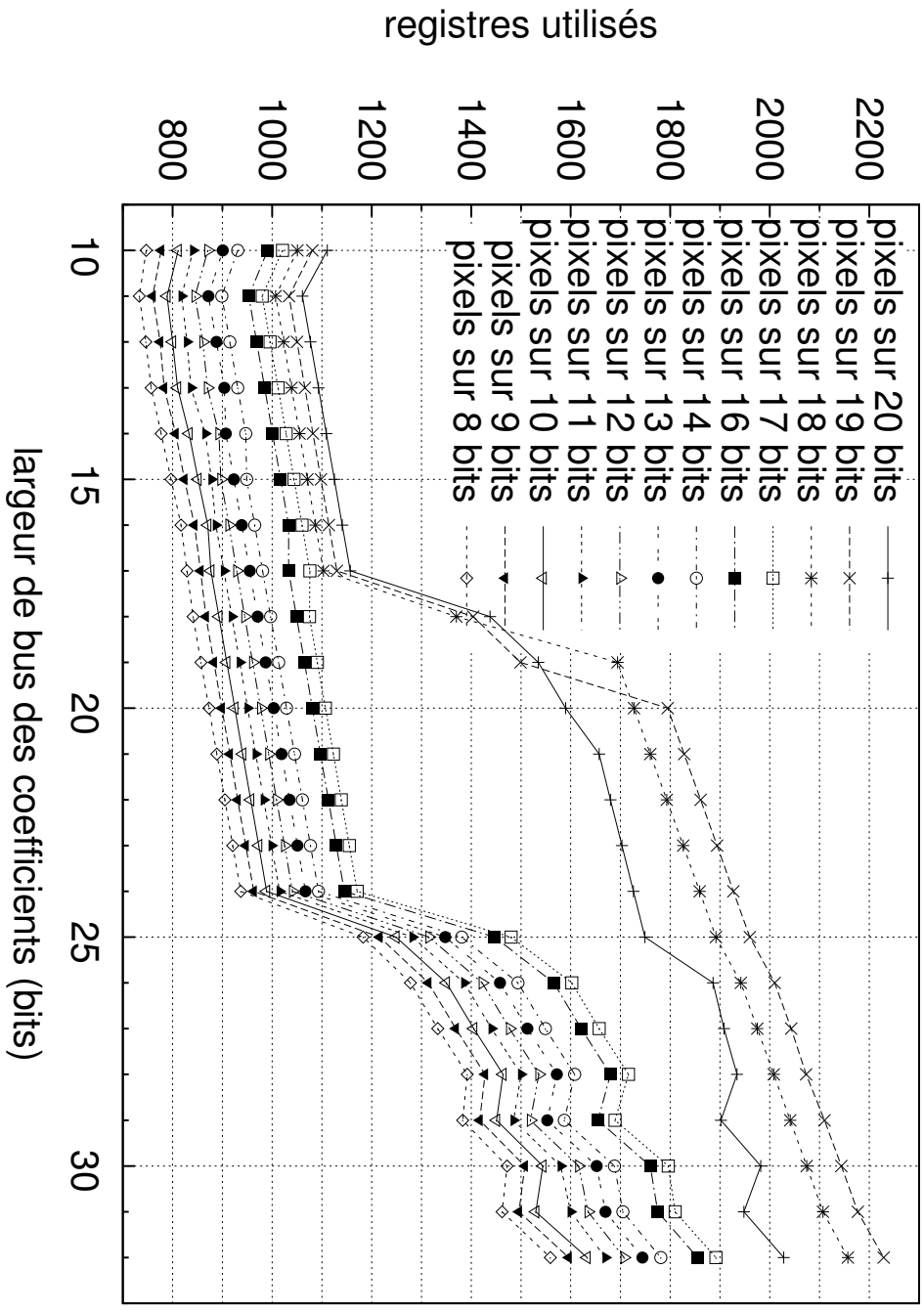
IP de filtrage Gaussien : registres utilisés 1 - Utilisation de registres d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 3



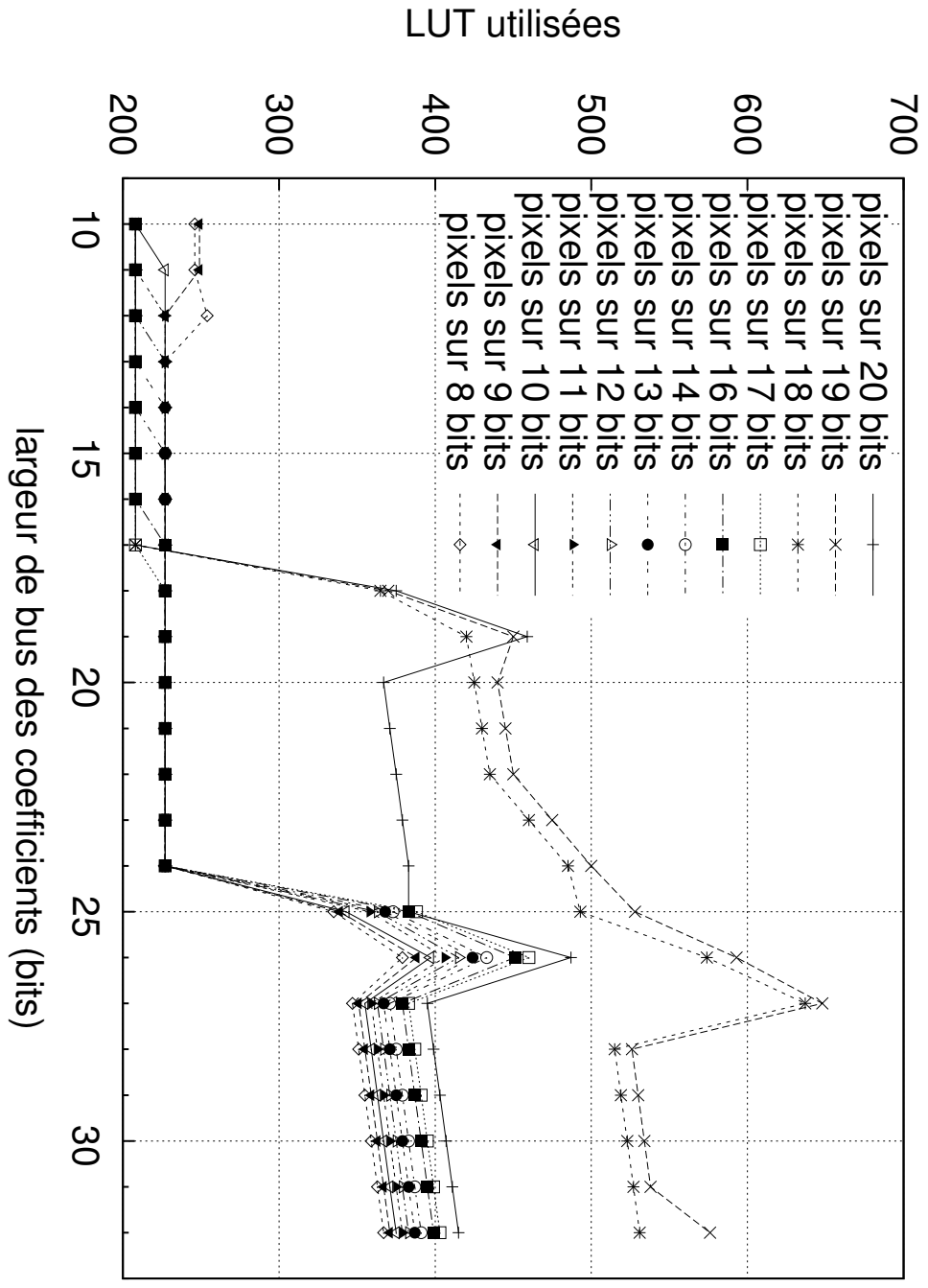
IP de filtrage Gaussien : registres utilisés 2 - Utilisation de registres d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 5



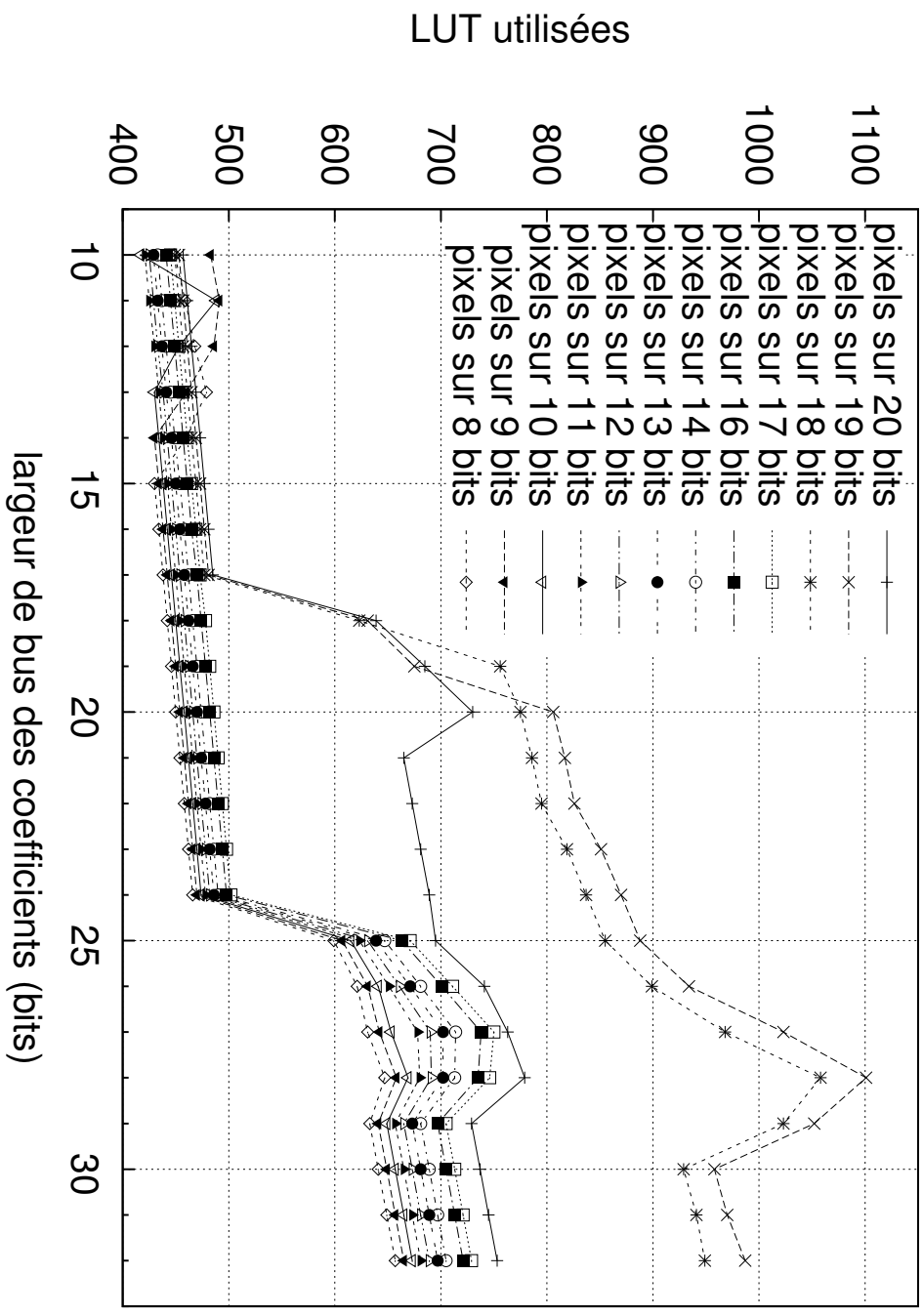
IP de filtrage Gaussien : registres utilisés 2 - Utilisation de registres d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 7



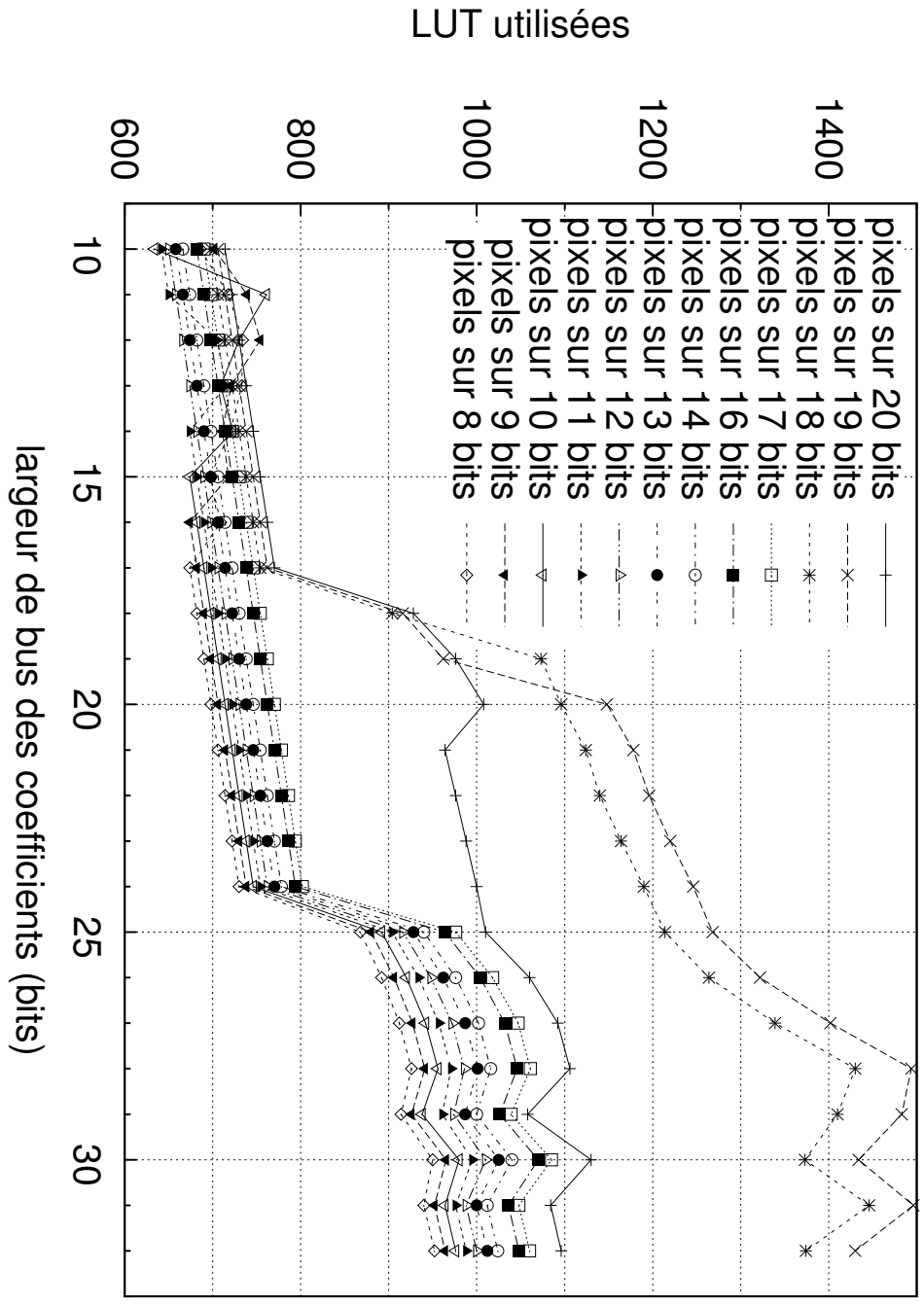
IP de filtrage Gaussien : registres utilisés 2 - Utilisation de registres d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 9



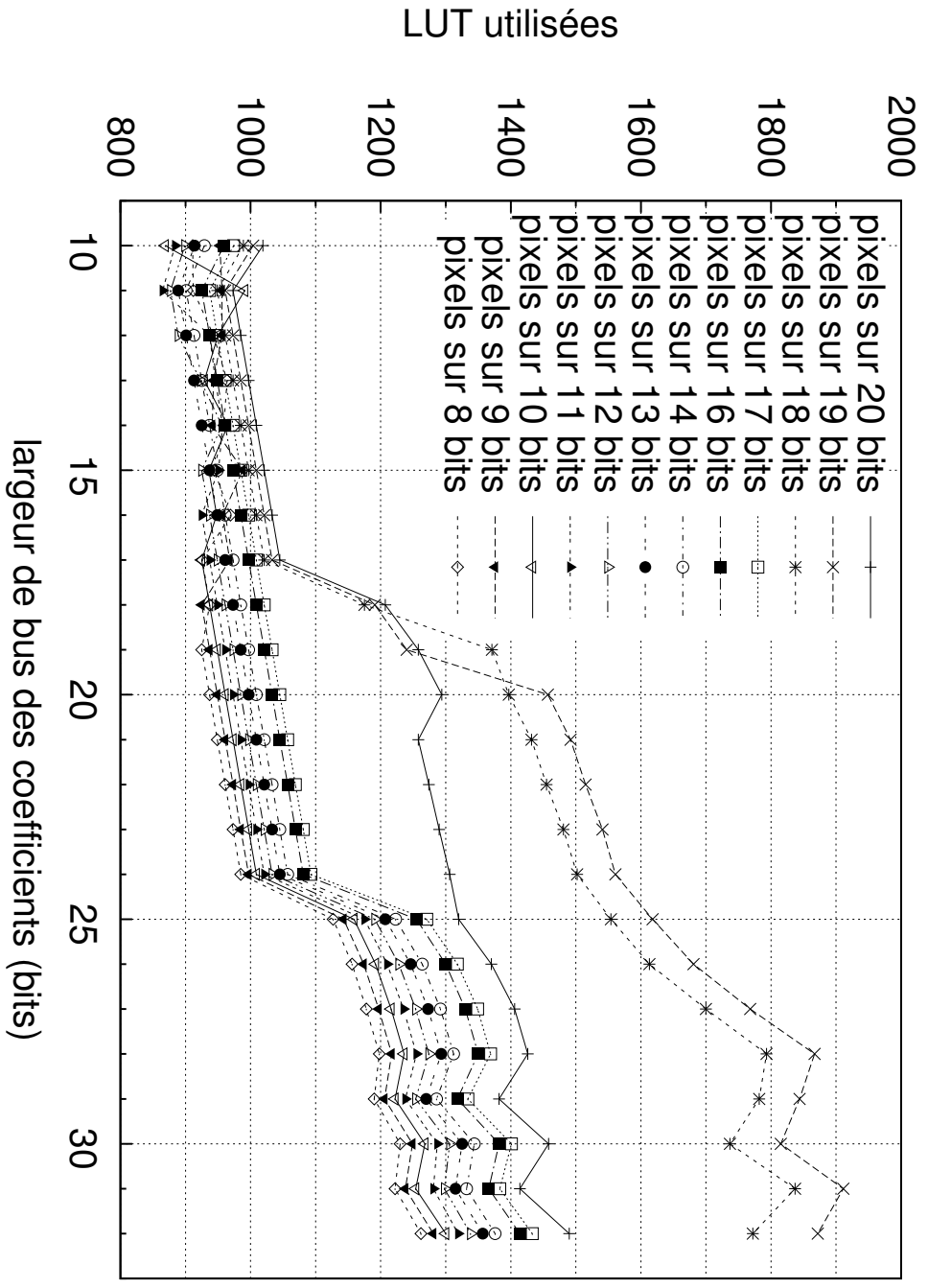
IP de filtrage Gaussien : LUT utilisées 1 - Utilisation de LUT d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 3



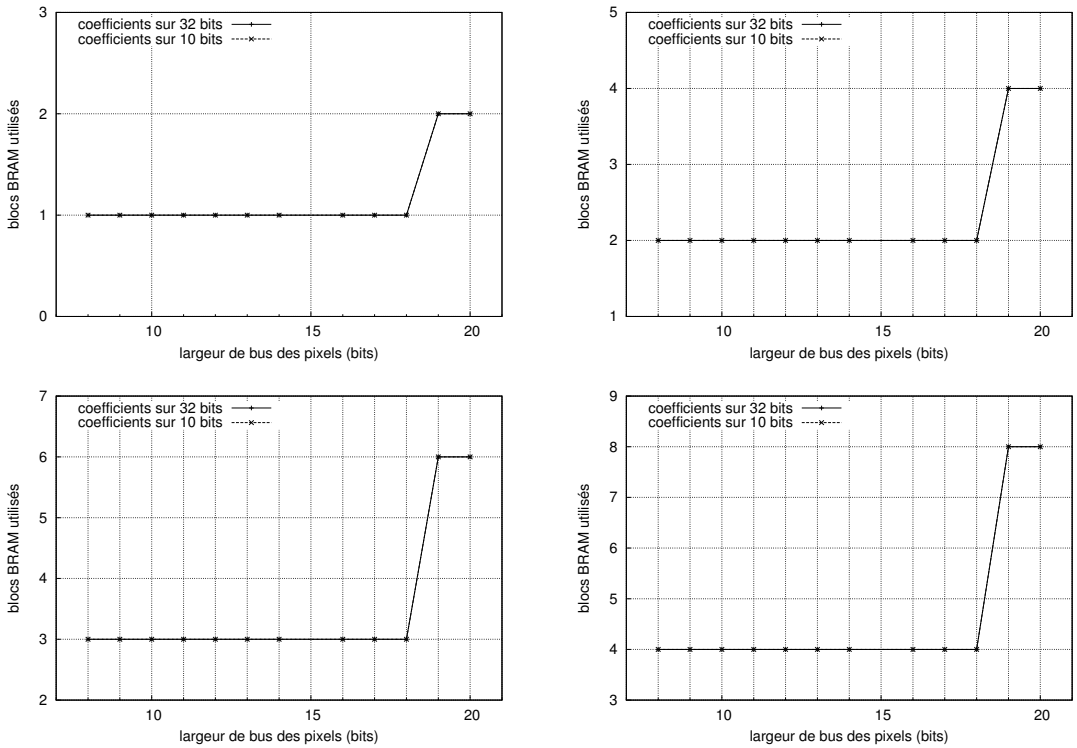
IP de filtrage Gaussien : LUT utilisées 1 - Utilisation de LUT d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 5



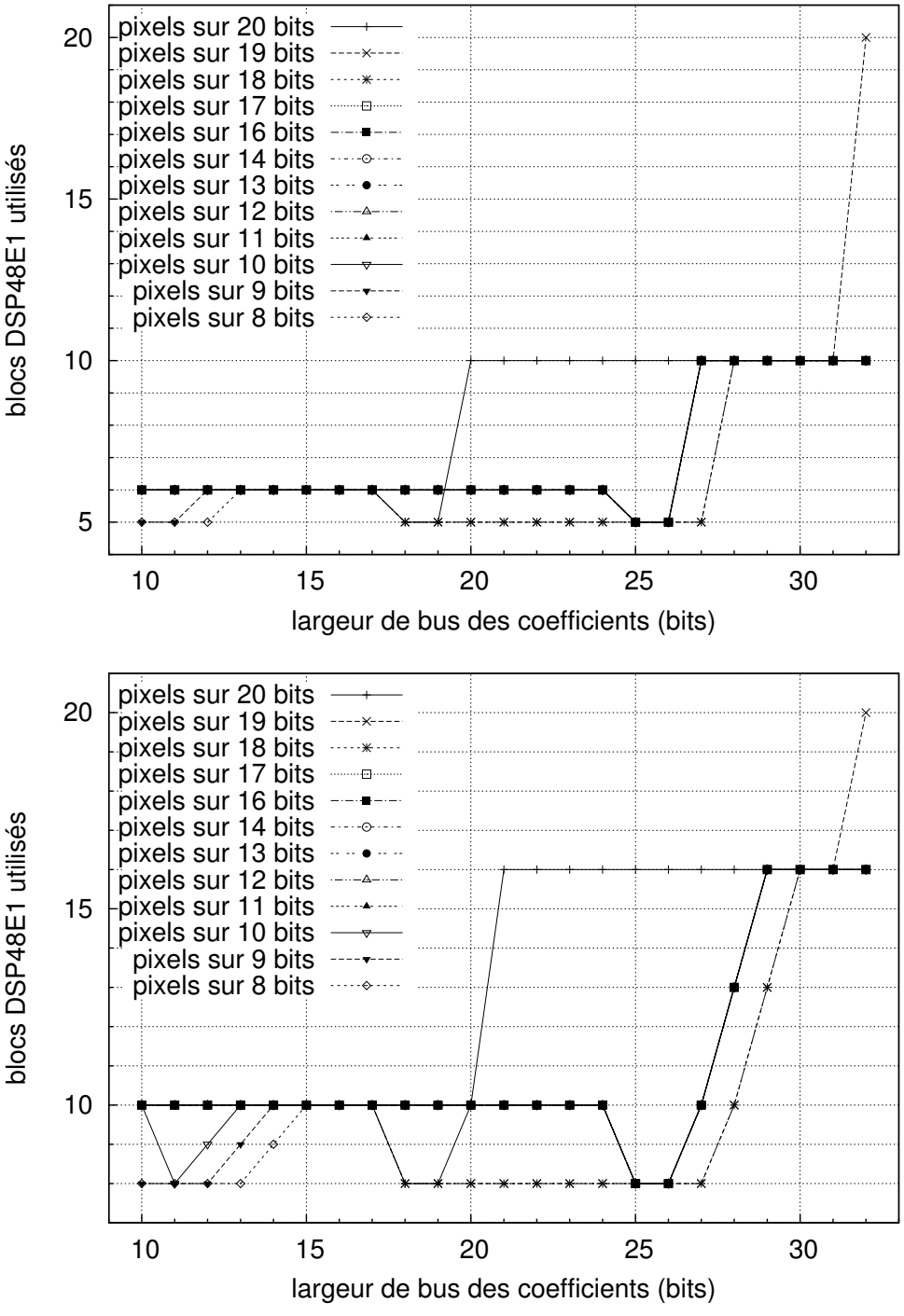
IP de filtrage Gaussien : LUT utilisées 2 - Utilisation de LUT d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 7



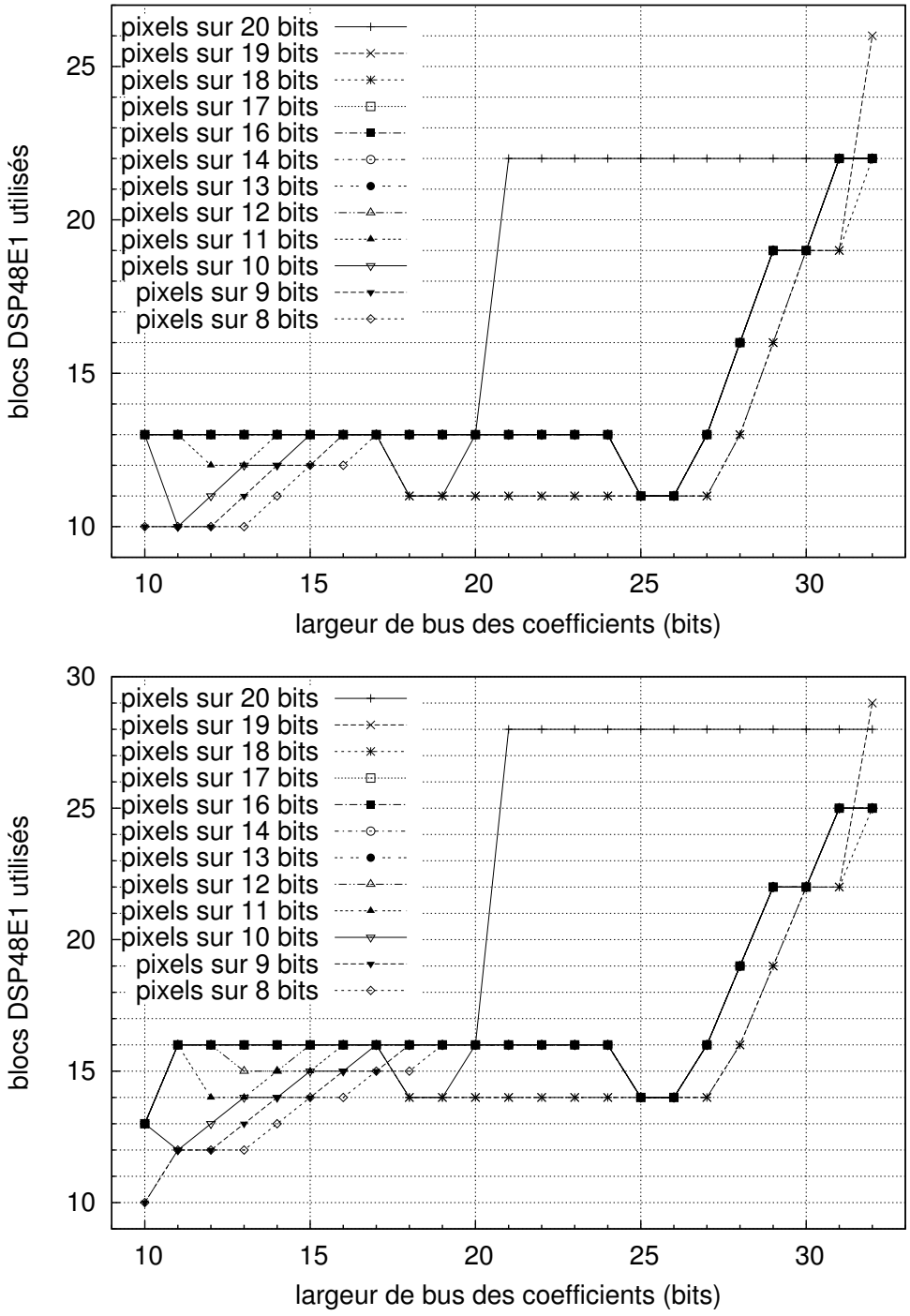
IP de filtrage Gaussien : LUT utilisées 2 - Utilisation de LUT d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et un noyau de convolution de largeur 9



IP de filtrage Gaussien : blocs BRAM utilisés 1 - Utilisation de blocs BRAM d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et des noyaux de convolution de largeur 3, 5, 7 et 9



IP de filtrage Gaussien : blocs DSP utilisés 1 - Utilisation de blocs DSP d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et des noyaux de convolution de largeur 3 et 5



IP de filtrage Gaussien : blocs DSP utilisés 2 - Utilisation de blocs DSP d'un FPGA de type Virtex 6 pour différentes largeurs de bus (pixels et coefficients), pour une image de 640×480 pixels, $\sigma = 2$, et des noyaux de convolution de largeur 7 et 9